SOURCE CODE COMMENTS UNDER GOOD AND BAD LENSES, AND THEIR ASSOCIATION WITH SOFTWARE QUALITY: AN EMPIRICAL INVESTIGATION ON OPEN SOURCE SOFTWARE

A Thesis by

Lifei Liu

Master of Business, Southwestern College, 2014

Bachelor of Management, Nantong University, 2006

Submitted to the Department of Electrical Engineering and Computer Science and the faculty of the Graduate School of Wichita State University in partial fulfillment of the requirements for the degree of

Master of Science

May 2019
The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Computer Science.

Huzefa Kagdi, Committee Chair

Zhiyong Shan, Committee Member

Krishna Krishnan, Committee Member
DEDICATION

To my family,
and my dear friends.
Here I enthusiastically express my gratitude to Dr. Huzefa Kagdi, my adviser and thesis committee chair. Dr. Kagdi’s expertise, understanding, advices, supports, and patience add precious value to my graduate experience. His deep knowledge and research skill in the area of software maintenance and evolution, and his expertise in research publications are really admirable. Without his advises and constant support this thesis would not be possible. I am very lucky to have such a knowledgeable and supportive adviser like Dr. Kagdi. I would also like to thank my thesis committee members who put their valuable time to review my thesis and provide feed backs that improved it and made it a successful one. My sincere gratitude goes to the faculty members that I came across during my entire graduate program at Wichita State University. Last but not least, I would like to thank my family, who provided enormous mental and financial support during my graduate study. I would also like to thank all my friends. They have been a most important part of my journey throughout my life and graduate studies.
Comments are ubiquitous in source code of real software systems. Software developers rely on them for comprehending and evolving software systems, i.e., to add new features and fix bugs. They imbibe the practice of commenting code from their formative years. Therein lies the critical questions for which scientific answers are largely and strikingly absent: “how prevalent are good and bad comments in production code, e.g., open source software?” or “do developers agree on their usefulness, e.g., with varied levels of experience?” Are these issues a matter of pure vanity or do they bear substance, e.g., do comments associate, if at all, with software complexity and quality, and to what extent, e.g., cohesion and coupling metrics? Answers to these questions directly influence developer communication and productivity, and software cost, reliability, and quality.

This work conducted a series of rigorous empirical studies in quest of initial answers to the above stated questions. Both quantitative and qualitative investigations were performed. Our results from six open source projects show that although 60% of source code comments are good, there are 40% bad comments (with a supermajority agreement). Not all developers always agree on the goodness or badness of specific comments. We also investigated the correlation between object-oriented complexity, cohesion, and coupling metrics with the source code comments. We did find evidence for increased levels of comments with low quality and/or high-complexity code.

Future work entails the automatic classification of source code comments into a taxonomy of good and bad comments, and to formulate approaches to prevent and eliminate bad comments (e.g., refactor code reeking with bad comment).
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Comment Categories</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Methods for Comment Classification</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Good Comments</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Legal comments</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Informative comments</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Explanation of intent</td>
<td>9</td>
</tr>
<tr>
<td>2.2.4 Clarification Comment</td>
<td>10</td>
</tr>
<tr>
<td>2.2.5 Warning of consequences</td>
<td>11</td>
</tr>
<tr>
<td>2.2.6 TODO comments</td>
<td>11</td>
</tr>
<tr>
<td>2.2.7 Amplification Comment</td>
<td>12</td>
</tr>
<tr>
<td>2.2.8 Javadoc in public APIs</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Bad Comments</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Mumbling Comment</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Redundant Comment</td>
<td>14</td>
</tr>
<tr>
<td>2.3.3 Misleading Comment</td>
<td>14</td>
</tr>
<tr>
<td>2.3.4 Mandated Comment</td>
<td>15</td>
</tr>
<tr>
<td>2.3.5 Noise Comments/ Scary Noise</td>
<td>15</td>
</tr>
<tr>
<td>2.3.6 Don’t Use a Comment When You Can Use a Function or a Variable</td>
<td>15</td>
</tr>
<tr>
<td>2.3.7 Position Markers Comment</td>
<td>15</td>
</tr>
<tr>
<td>2.3.8 Closing Brace Comments</td>
<td>15</td>
</tr>
<tr>
<td>2.3.9 Attributions and Bylines</td>
<td>15</td>
</tr>
<tr>
<td>2.3.10 Commented-Out Code</td>
<td>16</td>
</tr>
<tr>
<td>2.3.11 HTML Comment</td>
<td>16</td>
</tr>
<tr>
<td>2.3.12 Nonlocal Information</td>
<td>16</td>
</tr>
<tr>
<td>2.3.13 Too Much Information</td>
<td>17</td>
</tr>
<tr>
<td>2.3.14 Inobvious Connection</td>
<td>17</td>
</tr>
<tr>
<td>2.3.15 Function Header</td>
<td>17</td>
</tr>
<tr>
<td>2.3.16 Javadoc in Nonpublic Code</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Summary</td>
<td>18</td>
</tr>
<tr>
<td>3 A Study on the Applicability of Classification and Agreement Among Developers</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Data and Methods</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Results</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Revised categorizations</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1 Javadoc comments</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2 Non-Javadoc comments</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Threats to Validity</td>
<td>25</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>25</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>A Study on the Prevalence of Good and Bad Comments in Open Source Projects</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Data and Methodology</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Results</td>
<td>30</td>
</tr>
<tr>
<td>4.3 Threats to Validity</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Discussion</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Source Code Comments and their Association with Complexity and Quality</td>
<td>33</td>
</tr>
<tr>
<td>5.1 Data and Methodology</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Results</td>
<td>36</td>
</tr>
<tr>
<td>5.3 Threats to Validity</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Related Work</td>
<td>38</td>
</tr>
<tr>
<td>6.1 Are comments helpful for software comprehensibility?</td>
<td>38</td>
</tr>
<tr>
<td>6.2 Comments and Code Quality(Fault-proneness)</td>
<td>39</td>
</tr>
<tr>
<td>6.3 Detecting bad Javadoc comments</td>
<td>39</td>
</tr>
<tr>
<td>6.4 Comments classification</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Conclusions and Future Work</td>
<td>41</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>42</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Facts of Open source projects</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Facts of sampled source files</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Correlation coefficient of LCOM, CBO, RFC and WMC with number of comments</td>
<td>37</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Legal Comment Example</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Informative Comment 1</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Informative Comment 2</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Intent Comment 2</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Intent Comment 3</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Clarification Comment 1</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Clarification Comment 2</td>
<td>11</td>
</tr>
<tr>
<td>2.8</td>
<td>Warning Comments 1</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>Warning Comment 2</td>
<td>11</td>
</tr>
<tr>
<td>2.10</td>
<td>TODO Comment 1</td>
<td>12</td>
</tr>
<tr>
<td>2.11</td>
<td>Amplification Comment 1</td>
<td>12</td>
</tr>
<tr>
<td>2.12</td>
<td>Amplification Comment 2</td>
<td>12</td>
</tr>
<tr>
<td>2.13</td>
<td>Javadoc Comment</td>
<td>13</td>
</tr>
<tr>
<td>2.14</td>
<td>Mumbling Comment 1</td>
<td>13</td>
</tr>
<tr>
<td>2.15</td>
<td>Mumbling Comment 2</td>
<td>13</td>
</tr>
<tr>
<td>2.16</td>
<td>Redundant Comment 1</td>
<td>14</td>
</tr>
<tr>
<td>2.17</td>
<td>Redundant Comment 2</td>
<td>14</td>
</tr>
<tr>
<td>2.18</td>
<td>Misleading Comment</td>
<td>15</td>
</tr>
<tr>
<td>2.19</td>
<td>Closing Brace Comment</td>
<td>16</td>
</tr>
<tr>
<td>2.20</td>
<td>Commented out code</td>
<td>16</td>
</tr>
<tr>
<td>2.21</td>
<td>Non Local Comment</td>
<td>17</td>
</tr>
<tr>
<td>2.22</td>
<td>Non Public Javadoc</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Algorithm Summary Comment</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Algorithm Summary Comment 2</td>
<td>24</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES (continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comment Manually classify tool</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Pie Chart Categories in Javadoc Comments</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Pie Chart of categories in Non-Javadoc comments</td>
<td>31</td>
</tr>
</tbody>
</table>
CHAPTER 1
Introduction

Large-scale software systems are continuously subjected to evolution, i.e., changes due to bug fixes and new features, to be relevant and sustainable. Previous research shows that software developers spend a significant portion of their time in comprehending existing source code before they can evolve it. Software developers extensively utilize and rely upon secondary constructs, such as comments and identifier names, for human-to-human communication. Comments are approximately 30% of source code in terms of code size (lines of code). General-purpose programming languages, such as C++ and Java, are generous in giving these constructs the innate power of natural languages; however, they do not enforce a strict sense of verification, i.e., they are correct, meaningful, effective or helpful, or even harmful.

The conjecture is that the commented source code would aid humans in program comprehension, whereby allowing not only for effective and efficient changes, but also prevent defects or errors, in software. There is limited to no evidence on how much commenting is necessary and sufficient to aid human developers and improve software quality. Too much of it could litter the source code and prove to be detrimental (e.g., comments not maintained along with the associated code). A common best practice suggests that comments should reveal the intent behind a unit of source code, e.g., design decisions, historical warnings, and idiosyncrasies of the problem domain. It should not be a mere re-documentation or alternative description of the implementation of the constructs used from the solution domain. On the other hand, software developers are inflicted with the mindset of erring on the side commenting more (and not less) right from their formative years in education and practice. Organizational and project policies often promote or even require such commenting standards. This collective state of affairs calls for a foray on the conventional wisdom of source code comments. That is the helpful and harmful effects of source comments need to be
revisited with an investigative lens of empirical studies. Any problems arising from the findings thereof need to be addressed with well-informed and effective solutions.

Over the last two decades or thereabout, there has been a monumental level of research efforts into applying information retrieval, natural language processing, and text analysis techniques in software engineering. Such techniques are used either individually or in conjunction with each other or in concert with traditional program analysis methods, i.e., static/structural and behavioral/dynamic analyses, in processing software artifacts to support various software engineering tasks. The ever-increasing application domain includes, but is not limited to, tasks such as program comprehension, traceability recovery, impact analysis, and concept location. One of the key feeders to these applications is source code in general, and identifiers and comments in particular. The expressiveness and effectiveness of these techniques rely on these source code units. With the exception of limited pre-processing of comments (e.g., stemming and stop word removal), they are fed non-discriminatively regardless of their quality (e.g., good or bad). This state of affairs calls for another line of investigation into the potential impact of good and bad comments on solutions driven by such techniques. In summary, the role of comments in terms of software comprehension, reliability, and automated tools needs to be reexamined. Additionally, a new dimension of automated techniques and tools to improve the quality of source code comments and their applications needs to be established should they be warranted based on an empirical foundation.

The thrust of thesis is to empirically investigate the prevalence of good (desired) and bad (undesired) comments in the source code base of real software systems, e.g., in the open source world. Furthermore, we examine the agreement among software developers on their good and bad ratings, and how the scale of comments associate with the complexity and quality of code. It is anchored in investigation of the following set of research questions:

**RQ1:** How much do developers agree on the good and bad ratings of source code comments?
RQ2: How prevalent are good and bad source code comments in real software systems from the open source domain?

RQ3: How do source code comments associate with the complexity and quality of real software systems from the open source domain?

Martin et al. [7] proposed a classification of good and bad comments, and narrative guidelines along with examples for each category. Their classification appears plausible; however, they did not offer any empirical evidence. Using their classification as a starting point, we conducted a series of rigorous empirical studies in quest of initial answers to the above stated questions. Both quantitative and qualitative investigations were performed. Our results from six open source projects show that although 60% of source code comments are good, there are 40% bad comments (with a supermajority agreement). Developers had a much higher agreement on the general categorization of good or bad comments than on the specific ones from the subject systems. We also investigated the correlation between object-oriented complexity, cohesion, and coupling metrics with the source code comments. We did find evidence for increased levels of comments with low quality and/or high-complexity code.

The overarching goal of the proposed research program is to significantly advance the state-of-the-art in research, practice, and education of authoring, maintaining, and applications of source code comments. The thrust of this proposal is to empirically investigate the prevalence of good (desired) and bad (undesired) comments in the source code base of real software systems, e.g., in the open source world. Furthermore, we will examine the factors influencing the helpful and harmful effects of source code comments on software quality, especially in terms of program comprehension, reliability, and evolvability. Should there be credible empirical evidence suggesting a wide proliferation of bad (undesired) comments, we will formulate algorithms, methods, and software tools to identify and eliminate their harmful effect. The proposed three-year investigation sets the following main three objectives:

Comments are commonly used in source code and are considered important for software
maintenance and evolution. Traditionally, comments have been believed to aid human developers in comprehending source code. Early research investigations showed that code comments have significant usefulness in improving code readability and maintenance [21][9][20]. In recent years, the function of comments has been extended, such as automation of defect detection [17] [18]. With the popularity of agile software development in industry, more and more developers claim that good code should be self documentary and explanatory. Comments should be only used when something cannot be explicitly stated in the code.

Not all comments are of the same quality. Good comments provide a loyal explanation for concepts in the code otherwise difficult to understand. They also reveal the intent of the code. Bad comments can be redundant and noisy. As time goes by, the redundant comments can become misleading, because when people make updates or corrections on code these redundant comments may not be updated concurrently especially if the programmer is under time or task pressure. Some also believe that the amount of comments in a code is positively related to the chance of bugs. That is to say, comments are used to cover for the "bad smells". Therefore, it is important to distinguish good and bad comments. Good comments are meant to be kept, while bad comments should eventually be removed. In Martin R ‘s far-reaching book Clean Code'[7], he proposed guidelines and examples of classify good and bad comments. However, the application of these criteria in real field has not been investigated. In this thesis, we applied these principles and taxonomy to several open source projects in order to test the usefulness of these criteria. We also investigated the prevalence of bad comments, and the relationship between the number of comments and complexity metrics.

The rest of the document is organized as follows: Chapter 2 discusses the categorization/taxonomy of comments along with examples. A study on the applicability of these guidelines and agreement among developers is presented in Chapter 3. Chapter 4 presents a study on the prevalence of good and bad comments in real open source systems. Chapter 5 presents a quantitative analysis on the association between source code comments and software complexity and quality.
Related work is discussed in Chapter 6. We state our conclusions and direction for future work in Chapter 7.
CHAPTER 2

Comment Categories

The use of comments in source code is common because programmers cannot always express their intents purely within code; however, non-discriminatory and injudicious use of comments can be problematic. Martin [7] considered comments as a "necessary evil", and argued that software developers should always try to express themselves in code without using comments. Comments should not be used to make up bad code, instead, fixing the code itself is a much better practice. Just as Kernighan [6] pointed out: "Don’t comment bad code — rewrite it". The reason comments can be "bad" is that code changes and evolves, but comments may not always be changed to reflect the code accurately. It imposes risks pertaining to programmers being misled, whereby, compromising the original assumptions of the system. That said, comments are often necessary or beneficial. We call them "good comments". Good comments accurately explain the intent of the code while these intents cannot be easily explained in the code itself. Inaccurate and unnecessary comments are "bad comments". They are worse than no comments at all, because they are deluding and misleading, setting unfit expectations, or laying down old rules that are not or should not be followed any longer. Therefore, it is important to distinguish good and bad comments. Good comments have their values to be kept, while bad comments should eventually be avoided and removed. In this chapter, we aim to establish rules and criteria to distinguish good and bad comments, and categorize them.

2.1 Methods for Comment Classification

We studied three methods for categorization of comments: 1) Martin’s [7], 2) Steidl et al. [14], and Pascarella et al. [11]. We compared the merits and short comings of the 3 earlier published categorizations of comments. Based on our analysis, we proposed a methodology to distinguish and categorize good and bad comments based on the classification proposed in "Clean Code". In the book "Clean Code", Martin listed 8 kinds of good comments, and 18 kinds of bad comments. He
described what those comments look like and gave examples of each in open source code. Martin’s principles on the "badness" of comments are now widely accepted. These descriptions set the basic rules to distinguish good and bad comments. This book is fundamentally important in the industrial practices and serves as the current standards. However, the categories proposed in his book to classify good and bad comments are less studied.

Our study found that these categories are basically well set and relatively objective. They have great values in classifying comments in open source code. However, there are some shortcomings in these categorizations, which can be improved. First, this book does not give a clear definition of each kind of comments. There are not definitive criteria to be used in classifying each comment. Second, there are overlaps and unclear description among some categories. E.g. the "Warning of consequences" category contains a lot of overlap with the "Amplification" category, and the "Redundant comments" category overlaps with the "Mandated comments" category. Third, some of these categories are not applicable, and are very rarely seen in open source code. E.g. the "Closing Brace Comments". Fourth, depending on the condition in which the comments are used, some comments listed as "bad" have their values and make sense in that specific scenario. E.g. “HTML comments” are listed as bad comments. However, most ”HTML comments” would make sense or are even required in the Javadoc comments which are considered good comments. Steidl et al defined 7 classes of comments, but they are not clearly classified to good or bad. Rather, the classification is based on position than content of the comments. A quality model was proposed but is only applicable to the 7 proposed classes. Thus, these categories are less helpful for us to distinguish good and bad comments. Pascarella et al improved Steidl’s categorization. In their study, both position and the content of the comments were considered in classification. There are 6 main categories and 22 sub-categories. The methodology for categorization is detailed and suitable for automerization. The categorization uses regular expression, allowing direct classifying of a comment into a sub-category based on phrases contained in that comment. However, it is not easy to define good or bad using this method, therefore limited its meaningfulness and usefulness.
In this chapter, we first established a categorization system based on Martin’s categories, then we tested and validated the categorization in Chapter 2. Modifications were made to correct the short comings we found in the original Martin’s categories. Shown as below, comments are divided into good and bad groups. Examples are given for each category using the comments we found in open source projects.

2.2 Good Comments

2.2.1 Legal comments

Legal comments usually located in the beginning of every source code file, showing copyright and authorship statements.

```java
/*
 * Copyright (C) 2006 The Android Open Source Project
 * *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 * *
 * http://www.apache.org/licenses/LICENSE-2.0
 * *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

Figure 2.1: Legal Comment Example

2.2.2 Informative comments

Provide basic information with a comment. It can be useful, but sometimes can be removed if rename the function or refactor to a new function or class. This kind of comments using short sentence or a few words explaining what the commented code is or does. This kind of comments are tricky, whether it is good or bad depend on how much information it revealed beyond the code itself. If it is basically another way of restating the code, then it would be a redundant comment. How-
ever, if it tells too much beyond the code, it might be too-much information comment or non-local
comment. These three comments provide more information about the commented code, explicat-

```java
/** Type for IActivityManager.serviceDoneExecuting: anonymous operation */
public static final int SERVICE_DONE_EXECUTING_ANON = 0;
/** Type for IActivityManager.serviceDoneExecuting: done with an onStart call */
public static final int SERVICE_DONE_EXECUTING_START = 1;
/** Type for IActivityManager.serviceDoneExecuting: done stopping (destroying) service */
public static final int SERVICE_DONE_EXECUTING_STOP = 2;
```

Figure 2.2: Informative Comment 1

ing what will they used for. In this example, the comment explained more beyond the code itself,

```java
// the platform type affects volume and silent mode behavior
private final int mPlatformType;
```

Figure 2.3: Informative Comment 2

gives us more information about the variable mPlatformType, and what it may affect.

### 2.2.3 Explanation of intent

This kind of comments provide the intent of a decision. It gives the reason of why we write
the code or why we should do it in this way. Intent Comment 1 this comment here tells us why

```java
synchronized (mResourcesManager) {
    WeakReference<LoadedApk> ref;
    if (differentUser) {
        // Caching not supported across users
        ref = null;
    } else if ((flags & Context.CONTEXT_INCLUDE_CODE) != 0) {
        ref = mPackages.get(packageName);
    } else {
        ref = mResourcePackages.get(packageName);
    }
}
```

should check differentUser, and if it returns false, ref should be null This comment goes beyond
Figure 2.4: Intent Comment 2

give simple information about the code, but explains why we should keep state of pre-Honeycomb apps. This comment here explains why there is no call onPause()

Figure 2.5: Intent Comment 3

2.2.4 Clarification Comment

This kind of comments translate the meaning of some obscure arguments or return values into something readable. This comment is a clarification, gives the meaning of value of the variable
permissions.

```java
// Allow application-generated systrace messages if we're debuggable.
boolean appTracingAllowed = (data.appInfo.flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0;
```

Figure 2.7: Clarification Comment 2

### 2.2.5 Warning of consequences

This kind of comments give warning to other programmers about certain consequences.

```java
// Merge any pending results and pending intents; don't just replace them
if (tmp.pendingResults != null) {
    if (r.pendingResults == null) {
        r.pendingResults = tmp.pendingResults;
    } else {
        r.pendingResults.addAll(tmp.pendingResults);
    }
}
if (tmp.pendingIntents != null) {
    if (r.pendingIntents == null) {
        r.pendingIntents = tmp.pendingIntents;
    } else {
        r.pendingIntents.addAll(tmp.pendingIntents);
    }
}
```

Figure 2.8: Warning Comments 1

```java
// Don't set application object here -- if the system crashes, we can't display an alert, we just want to die die die.
android.ddm.DdmHandleAppName.setAppName("system_process",
        UserHandle.myUserId());
```

Figure 2.9: Warning Comment 2

### 2.2.6 TODO comments

Comments that marked "TODO" to remind the programmer future works.
2.2.7 Amplification Comment

This kind of comments are used to amplify the importance of something that may otherwise seem inconsequential. In this case, the programmer feel it is necessary to comment.

```java
// Need to ensure state is saved.
if (!r.paused) {
    performPauseActivity(r.token, false, r.isPreHoneycomb());
}
```

Figure 2.11: Amplification Comment 1

```java
// Watch for getting close to heap limit.
BinderInternal.addGcWatcher(new Runnable() {
    @Override public void run() {
        if (!mSomeActivitiesChanged) {
            return;
        }
    }
}
```

Figure 2.12: Amplification Comment 2

2.2.8 Javadoc in public APIs

Javadoc comments are very useful when it is well written.
2.3 Bad Comments

2.3.1 Mumbling Comment

This kind of comments are written not clear and difficult to understand. They may force readers to examine other parts of the system to find out what’s going on. This is an example of mumbling comment. It is confusing that there are no clues this method will tell whom that it is alive. By reading the whole function, we got the answer that it wants to tell the OS. This comment also raises the question that who will do the backup work. Is it unScheduleGcIdler() will do it or the programmer should later on write code to do it. To understand it, we have to refer other functions.

```java
// Instantiate a BackupAgent and tell it that it's alive
private void handleCreateBackupAgent(CreateBackupAgentData data) {
    if (DEBUG_BACKUP) Slog.v(TAG, "handleCreateBackupAgent: " + data);
}
```

Figure 2.14: Mumbling Comment 1

```java
// no longer idle; we have backup work to do
unscheduleGcIdler();
```

Figure 2.15: Mumbling Comment 2
2.3.2 Redundant Comment

Redundant comments show no more information than the code itself. It may take more time to read than the code itself. This kind of comments are most frequent among all bad comments in the later investigating. They often appear in the head of functions or variables.

```
// Tear down a BackupAgent
private void handleDestroyBackupAgent(CreateBackupAgentData data) {
    if (DEBUG_BACKUP) Slog.v(TAG, "handleDestroyBackupAgent: " + data);

    LoadedApk packageInfo = getPackageInfoNoCheck(data.appInfo, data.compatInfo);
    String packageName = packageInfo.mPackageName;
    BackupAgent agent = mBackupAgents.get(packageName);
    if (agent != null) {
        try {
            agent.onDestroy();
        } catch (Exception e) {
            Slog.w(TAG, "Exception thrown in onDestroy by backup agent of " + data.appInfo);
            e.printStackTrace();
        }
        mBackupAgents.remove(packageName);
    } else {
        Slog.w(TAG, "Attempt to destroy unknown backup agent. " + data);
    }
}
```

Figure 2.16: Redundant Comment 1

```
if (services != null) {

    // Setup the service cache in the ServiceManager
    ServiceManager.initServiceCache(services);
}
```

Figure 2.17: Redundant Comment 2

2.3.3 Misleading Comment

Misleading comments are those not precise enough and may cause confusing and misunderstanding.
2.3.4 Mandated Comment

obsessively give every function or variable a Javadoc or a common comment. They can be seemed as noise comment.

Journal Comment

This kind of comments are added every time the author editing the file and accumulated as a kind of journal or log.

2.3.5 Noise Comments/ Scary Noise

This kind of comments are nothing but noise, they restate the obvious and provide no new information. They are sometimes seemed as redundant comments because they all did not provide more information than the code itself.

2.3.6 Don’t Use a Comment When You Can Use a Function or a Variable

This kind of comments could be eliminated if they use a better function or variable name.

2.3.7 Position Markers Comment

They are used to mark a particular position in source file. They become noisy if they are used too often.

2.3.8 Closing Brace Comments

They are special comments that are put on closing braces.

2.3.9 Attributions and Bylines

They are used to record who added the code and when. This kind of comment should be updated by source code control system.
2.3.10 Commented-Out Code

This is an obvious and common type of bad comments. Commented-out code should be deleted when it is no longer needed.

```java
public static IPackageManager getPackageManager() {
    if (sPackageManager != null) {
        // Slog.v("PackageManager", "returning cur default = " + sPackageManager);
        return sPackageManager;
    }
    IBinder b = ServiceManager.getService("package");
    // Slog.v("PackageManager", "default service binder = " + b);
    sPackageManager = IPackageManager.Stub.asInterface(b);
    // Slog.v("PackageManager", "default service = " + sPackageManager);
    return sPackageManager;
}
```

Figure 2.20: Commented out code

2.3.11 HTML Comment

Having HTML tag in the source code comments. They usually appeared in Javadoc comments and were intent to be extracted by some tool. If it used too much, people will have difficult to read.

2.3.12 Nonlocal Information

This kind of comments describe the code that not near the comment, or they are system-wide information with little relation to the local comment. This exampled comment has little information related to local code. This comment mentioned function was not called in the commented function.
2.3.13 Too Much Information

Interesting historical discussions or irrelevant descriptions in the comments.

2.3.14 Inobvious Connection

The connection of code and comments are not obvious.

2.3.15 Function Header

Comments for short functions, usually they are unnecessary.

2.3.16 Javadoc in Nonpublic Code

```java
private static class StreamOverride
    implements AccessibilityManager.TouchExplorationStateChangeListener {

    // AudioService.getActiveStreamType() will return:
    // - STREAM_NOTIFICATION on tablets during this period after a notification stopped
    // - STREAM_MUSIC on phones during this period after music or talkback/voice search prompt
    // stopped
    private static final int DEFAULT_STREAM_TYPE_OVERRIDE_DELAY_MS = 0;
    private static final int TOUCH_EXPLORE_STREAM_TYPE_OVERRIDE_DELAY_MS = 1000;

    static int sDelayMs;

    static void init(Context ctxt) {
    
    Figure 2.21: Non Local Comment

    Figure 2.22: Non Public Javadoc
```
2.4 Summary

Good comments are similar. Major good comments reveal information that is not shown in the code but is important to understand the code. Other good comments serve as tools (etc. Javadoc comments, TODO comments, legal comments). Bad comments have different bad traits. They can be very similar to the code that they only repeat it but do not add any value (etc. redundant comments, noise comments, mandated comments, function headers), or they can be too loose to the code that they seem lack of immediate connection to the code (etc. Inobvious connection, too much information, non local information). Some are not organized well thus raise difficulties to understand (etc. mumbling comments, misleading comments and comments could be substituted by function or variable). Still some are inherited from previous code but they are not in common anymore (etc. HTML comments, position marker, closing brace comment, journal comments, attributions and bylines).
CHAPTER 3

A Study on the Applicability of Classification and Agreement Among Developers

In Chapter, we presented the existing guidelines for classifying comments into good or bad categories. These guidelines are not specifications on which an automated tool can be build and employed. This fact raised questions as to how applicable these guidelines are for developers to help classify comments and whether developers agree on the specific category comments are classified to. That is, we investigated the following research question of the thesis:

**RQ1:** How much do developers agree on the good and bad ratings of source code comments?

### 3.1 Data and Methods

We selected 3 large sized source code files of Android system which contains a total of 284 comments. These comments were divided into 17 code pieces each contain round 20 comments. Each code piece was given to 3-14 graduate students from CS780 Advanced Software Engineering class who had just learned the comment categories described in Chapter 1. The students (evaluators) are asked to read the code and classify each comment into the categories described in Chapter 1. Based on the data gained from the student evaluations for the comments, improvements were made to the original descriptions to achieve more objective criteria.

### 3.2 Results

The responses of each evaluator to the assigned comments are shown as Supplementary Data 1. Our primary research question is that whether the evaluators agree with each other in distinguishing good versus bad comments when applying the descriptions? For each specific comment, we counted the votes for good or bad among evaluators. If 60% or more evaluators classified it to “good” category, the comment was considered good. If 60% or more evaluators agreed on “bad”, then the comment was considered bad. Of the total 284 evaluated comments, 86 comments were
categorized as good, and 140 categorized as bad. These comments (226 comments) achieved an “agreement” among the evaluators. The rest 58 comments did not achieve an “agreement” among the evaluators using the descriptions in Chapter 1. The agreement rate is 80%. In another word, by applying the descriptions in Chapter 1, 80% of the tested comments can be categorized into good or bad comments without ambiguity.

The students were familiarized with the descriptions before given the comments to evaluate. However, the levels of each student in understanding and mastering the description may vary. To control the influence of this variation, the comments were also evaluated by the author of this thesis independently. When the author’s classification was counted along with the students’ classification, there were 106 comments categorized as good by more than 60% of evaluators, and 131 categorized as bad by more than 60% of evaluators. The agreement rate was 83.5%.

If we look at whether the author’s responses agree with the students responses as a whole for each comment, then among the total 226 comments that the students achieved agreement, there were 161 comments that the author also agreed with the students on either it’s good or bad. The agreement rate was 71%. The consistency in the agreement rate was good but not ideal. There’s room for improvement. The relative consistency indicated that the evaluations were not substantially influenced by individual variations in understanding the “good” or “bad” concept based on the descriptions in Chapter 1. In another word, the “good” or “bad” nature of a comment was clearly described by the descriptions in Chapter 1. There’s some discrepancy based on the experience of the evaluator, but this does not influence “good” or “bad” classification for majority (>70%) of comments.

Taken together, the above data showed that, in general the descriptions in Chapter 1 is useful in distinguish good or bad comments, and can achieve consistent results in classifying comments into good or bad. Our second research question is that, how well do the evaluations agree on categories listed in Chapter 1? Based on all the responses including students’ and the author’s, there were only 18 comments classified into the same category by all the evaluators (100% agree
level), counting for only 6.3% of total evaluated comments. There were 72 commented classified to the same category if we lower the bar to 60% agree level (60% evaluator classified it to the same category), counting for only 25% of total evaluated comments. The data showed that when specific categories are concerned, a common agreement is lack in the categorization. It is often that one comment is classified to 3 different categories by 3 evaluators. There is seldom universal agreement. This showed that the sub-categorization based on the original descriptions as in Chapter 1 are influenced by individual judgment or bias. Our tests also pointed out a few short comings in the original category descriptions that caused the inconsistency in categorization of comments. These short comings include: 1) definitions are not clear for some categories, causing confusion in categorization; 2) There are overlaps among some categories, causing confusion in discriminating among categories.

3.3 Revised categorizations

To improve the categorization methods, we rearranged the categories by combining severely overlapped categories, remove redundant and confusing ones, as well as those rarely seen any more. In the new method, comments are first divided into two groups based on their format and purpose: Javadoc API comments and other inline comments. Javadoc API are provided as the only reliable resource other than source code itself for users of this code. Usually users would not refer to source code if there is a Javadoc API available. Inline comments are used by developers of the source code, they should be focused on implementation level of the source code. Then we decide whether a comment is a good or not in each of the two groups. Below are the definitions for good and bad comments in each group.

3.3.1 Javadoc comments

If a Javadoc comment generally satisfy the following guidelines, we consider it as a good Javadoc comment: Class/ interface level include: a. executive summary, a precise and concise description of the object. b. (optional)other information: state, security constrains, OS/hardware
dependencies. Field Javadoc comments should include one or more of following: a. what the
field models, the aspect of this field b. range of the valid value method Javadoc comments should
include one or more of following: a. expected or desired behavior of this operation. b. Range
of valid argument values and/or null argument values. c. Range of returned values d. Algorithm
defined e. Cause of Exceptions and security constrains.

Bad comment: A Javadoc comments is a bad one if it is: 1. Redundant: show no more
information than the class/interface/field name, or signature of a method. However, it is the most
frequent bad comments in Javadoc comments in practice. 2. Javadoc in private fields and methods.
3. Misleading: inconsistency in code and comments 4. Attribution: if a Javadoc comment has only
one attribute of author. 5. Noise: has no meaningful information or has no sense of appearing 6.
Other comments that do not reach the standard for a good comment.

3.3.2 Non-Javadoc comments

Good Non-Javadoc comments 1. Explanation of intent This kind of comments provide the
intent of a decision. It gives the reason of why we write the code or why we should do it in this way.
It often contains words such as ”because”, ”as”,”for the reason of”, ”since” and so on. 2. Informa-
tive or clarification This kind of comments answered the question what the code means. Usually
they translate the meaning of some obscure arguments or return values into something readable.
Or give you more background information of the commented code. 3. Amplification or warning
of consequences Amplify the importance of something that may otherwise seem inconsequential
or comments give warning to other programmers about certain consequences. 4. To do comment:
Has a “TODO”,”Fix:”, ”FixMe:” which indicates a remaining of future works. Bad non-Javadoc
comments: 1. Redundant information This kind of comments have large number of repetition word
which appeared in both comment text and commented code. The repetition rate could be calculated
by text similarity algorithms such as jaccard similarity or cosine similarity. 2. Commented out code
Piece of code that is given a comment form to avoid been compiled and execution. Commented
code can be easily detected by its “code features” such as ending with a semi-colon and/or contains
operation symbols. 3. Summary of code or algorithm: A summary of multiple lines of code that explaining what the code means or how the code works. 4. Misleading comments: There is inconsistency of code and comments. 5. Noise or mumbling comments: The meaning of comment is not clearly expressed, or there is no meaningful information exists in the comment. 6. Too much or nonlocal information comments: The comment is not explaining code that nearby. 7. Closing brace or position marker: Comment that appears followed with a closing brace, or in other position. The comment contains nothing but a position. 8. Other comments that do not reach the standard for a good comment.

All the new categories are based on the original categories described in Chapter 1, except for the bad Non-Javadoc comment 3: Algorithm. The is a new type of comment we found in addition to the original description. They are summaries of several lines of code or describe the algorithm of the code. They are not redundant because they are abstract of the meaning of code which increase the readability, but not informative of intent reveal because they do not go beyond the code. We term this type of comments as “Summary of code or Algorithm”. Because this kind of comments do not add any value to the code, they can be regarded as “bad”. We give two examples below.

```
// Sanity check the requested target package's uid against ours
try {
    PackageInfo requestedPackage = getPackageManager().getPackageInfo(
        data.appInfo.packageName, 0, UserHandle.myUserId());
    if (requestedPackage.applicationInfo.uid != Process.myUid()) {
        Slog.w(TAG, "Asked to instantiate non-matching package " + data.appInfo.packageName);
        return;
    }
} catch (RemoteException e) {
    Slog.e(TAG, "Can't reach package manager", e);
    return;
}
```

Figure 3.1: Algorithm Summary Comment
// instantiate the BackupAgent class named in the manifest
LoadedApk packageInfo = getPackageInfoNoCheck(data.appInfo, data.compatInfo);
String packageName = packageInfo.mPackageName;
if (packageName == null) {
    Slog.d(TAG, "Asked to create backup agent for nonexistent package");
    return;
}

String classname = data.appInfo.backupAgentName;
// full backup operation but no app-supplied agent? use the default implementation
if (classname == null ||
    (data.backupMode == IApplicationThread.BACKUP_MODE_FULL
     || data.backupMode == IApplicationThread.BACKUP_MODE_RESTORE_FULL)) {
    classname = "android.app.backup.FullBackupAgent";
}

try {
    IBinder binder = null;
    BackupAgent agent = mBackupAgents.get(packageName);
    if (agent != null) {
        // reusing the existing instance
        if (DEBUG_BACKUP) {
            Slog.v(TAG, "Reusing existing agent instance");
        }
        binder = agent.onBind();
    } else {
        try {
            if (DEBUG_BACKUP) Slog.v(TAG, "Initializing agent class " + classname);

            java.lang.ClassLoader cl = packageInfo.getClassLoader();
            agent = (BackupAgent) cl.loadClass(classname).newInstance();

            // set up the agent's context
            ContextImpl context = ContextImpl.createAppContext(this, packageInfo);
            context.setOuterContext(agent);
            agent.attach(context);

            agent.onCreate();
            binder = agent.onBind();
            mBackupAgents.put(packageName, agent);
        } catch (Exception e) {
            // If this is during restore, fail silently; otherwise go ahead and let the user see the crash.
            Slog.e(TAG, "Agent threw during creation: " + e);
        }
    }
}
The comment “`instantiate the BackupAgent class named in the manifest`” abstracts and explains the following 60 lines of code.

### 3.4 Threats to Validity

**Insufficient Dataset:** Both number of evaluators who manually classified same comments and the amount of comments are relatively small. this may cause bias on defining the category of the comment. **Unrepresentative Dataset:** All the comments came from same project. on one hand, it could let the evaluators more familiar with the context of comments they categorized. On the other hand, it may cause unrepresentative. **Misclassification:** the evaluators’ experiences on Java and Android are vary and untested. This may cause misunderstanding of the intent of code and comments.Also, their comprehension of categories are also unknow.

### 3.5 Discussion

Application of the descriptions in Chapter 1 provided consistent results in distinguishing good or bad for majority of comments. They also provide conceptual standards clear enough even for inexperienced evaluators to justify whether a comment is good or bad. For majority of comments (70%), these descriptions are good enough to provide general guidelines to define whether it is good or bad. The data of this study also exposed some problems in the application of these original descriptions. First, there are 20% of comments that cannot be clearly distinguished to good or bad based on these descriptions. This suggested certain degree of ambiguity exists in these descriptions. More significantly, there lacks consistency in categorizing a comment. This suggested problems in the definition of each category, causing inconsistent understanding in these descriptions by different individuals. We developed a new categorization based on the values of the original descriptions. The advantages of the new methods are: 1) Definition for good or bad comments is clearer, providing unambiguous criteria; 2) The criteria provide clear-cut guidelines that can be easily followed by either people or machine learning. This creates a basis for future development of automation classifier; 3) Several overlapping or similar categories in the original
descriptions are combined. This could potentially decrease confusion and increase consistency while classifying a relative comment.
CHAPTER 4
A Study on the Prevalence of Good and Bad Comments in Open Source Projects

In the proceeding chapter, we established that developers can achieve a high level of agreement in classifying comments. That lead to another question as to how prevalent the different types of comments were in real open source systems. That is, we investigated the following research question of the thesis:

**RQ2:** How prevalent are good and bad source code comments in real software systems from the open source domain?

4.1 Data and Methodology

We selected 257 source files from 6 open source projects as our sample to manually classify each comment in them. Sample file was selected by its size, we choose some of small sized files(<10kb) medium sized files (10kb 30kb) and some big sized files(>30kb). The facts of open source projects and our samples are shown in table 1 and table 2. We first developed a tool to extract the comments from 6 open source projects. The code for the tool is enclosed as Supplementary Data 2. Then using the improved categorization definition that we defined in Chapter 2, we classified these comments into categories. Based on the category, it was determined as good or bad, and calculated the percentage of good or bad comments in each circumstance.

Figure 1 Comments Manually classify tool Section 1: we can choose a source code java file from preset sample folder. Section 2: showed a list of all comments in this file, start with line number of the comment Section 3: showed the attributions of the comment, including comment position(class top, class field, method top, inner method, undefined other), class name(the class name, if it’s belong a class) and method name(the method name, if its belong to this method). Section 4: comment details (including comment text and commented code) Section 5: manually choose the category of the comment and save it. All comments in a file was saved into a json file for later analyze and statistics.
Table 1 showed the numeric facts of the 6 open source projects we used for this study. As shown in the table, there were 1,384,474 lines of pure code including blank lines but excluding comment lines. A total of 126,742 comments were found in these 6 open source projects. The lines of the comments make up to 29% of total code lines. This is consistent with other reports [?] [16], and showing significant amount of comments.

loc: lines of pure code and blank lines, excluding comments
cloc: lines of comments
cloc/(loc+cloc): the ratio of comments in total lines

Our sampled 257 files contained 368 classes, 2,036 methods, and 2,850 comments, out of the total comments 1440 of them are Javadoc comments. We then manually reviewed and categorized the 2,850 comments using the new categorization established in Chapter 2. The number of sampled files and comments are shown in Table 2.
### Table 4.1: Facts of Open source projects

<table>
<thead>
<tr>
<th>Name of project</th>
<th>#class</th>
<th>#method</th>
<th>#comments</th>
<th>#javadoc</th>
<th>Files</th>
<th>loc</th>
<th>cloc</th>
<th>cloc/(loc+cloc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crawler4j</td>
<td>57</td>
<td>403</td>
<td>388</td>
<td>251</td>
<td>73</td>
<td>5386</td>
<td>2519</td>
<td>32</td>
</tr>
<tr>
<td>Android-base-core</td>
<td>4228</td>
<td>43682</td>
<td>62983</td>
<td>44891</td>
<td>2148</td>
<td>503087</td>
<td>378045</td>
<td>43</td>
</tr>
<tr>
<td>maven</td>
<td>1058</td>
<td>6619</td>
<td>4704</td>
<td>2145</td>
<td>961</td>
<td>81128</td>
<td>30869</td>
<td>28</td>
</tr>
<tr>
<td>flink</td>
<td>10471</td>
<td>51673</td>
<td>51623</td>
<td>23599</td>
<td>6255</td>
<td>683590</td>
<td>274792</td>
<td>29</td>
</tr>
<tr>
<td>spark</td>
<td>1638</td>
<td>9994</td>
<td>7044</td>
<td>2959</td>
<td>918</td>
<td>111283</td>
<td>30193</td>
<td>21</td>
</tr>
<tr>
<td>Subtotal</td>
<td>17452</td>
<td>112371</td>
<td>126742</td>
<td>73845</td>
<td>10355</td>
<td>1384474</td>
<td>716418</td>
<td>30.6</td>
</tr>
</tbody>
</table>

### Table 4.2: Facts of sampled source files

<table>
<thead>
<tr>
<th>Name of project</th>
<th>Classes</th>
<th>methods</th>
<th>comments</th>
<th>javadoc</th>
<th>files</th>
<th>LOC</th>
<th>CLOC</th>
<th>CLOC/(LOC+CLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crawler4j</td>
<td>57</td>
<td>403</td>
<td>388</td>
<td>251</td>
<td>69</td>
<td>5291</td>
<td>2176</td>
<td>29</td>
</tr>
<tr>
<td>Android</td>
<td>29</td>
<td>604</td>
<td>708</td>
<td>256</td>
<td>7</td>
<td>10357</td>
<td>3568</td>
<td>26</td>
</tr>
<tr>
<td>maven</td>
<td>59</td>
<td>471</td>
<td>287</td>
<td>118</td>
<td>54</td>
<td>4720</td>
<td>1883</td>
<td>29</td>
</tr>
<tr>
<td>flink</td>
<td>49</td>
<td>587</td>
<td>757</td>
<td>401</td>
<td>37</td>
<td>6448</td>
<td>3864</td>
<td>37</td>
</tr>
<tr>
<td>spark</td>
<td>81</td>
<td>399</td>
<td>518</td>
<td>255</td>
<td>63</td>
<td>4740</td>
<td>2856</td>
<td>38</td>
</tr>
<tr>
<td>cayenne</td>
<td>81</td>
<td>1637</td>
<td>1621</td>
<td>1132</td>
<td>64</td>
<td>18865</td>
<td>7961</td>
<td>30</td>
</tr>
<tr>
<td>Subtotal</td>
<td>316</td>
<td>3036</td>
<td>2850</td>
<td>1440</td>
<td>257</td>
<td>40086</td>
<td>18002</td>
<td>31</td>
</tr>
</tbody>
</table>
4.2 Results

The prevalence for total good or bad comments are as below: Total comments: 2850

GOOD COMMNETS in all: 1723 — 60.45%
Bad COMMNETS in all: 784 — 27.51%
Uncategorized in all: 343 — 12.04%

If group into Javadoc comments or Non-Javadoc comments: Javadoc: 1440

GOOD COMMNETS in Javadoc: 976 — 67.78%
bad COMMNETS in Javadoc: 447 — 31.04%
Uncategorized in Javadoc: 17 — 1.18%

category statistics of Javadoc comments:
private-javadoc=77, Noise=13, Misleading=4, Attrributions=33, OtherNotGood=17, Redundent=320,
JavaDoc=976

Non-Javadoc comments: 866
GOOD COMMNETS: 489 — 56.47%
BAD COMMNETS: 134 — 15.48%
Uncategorized: 243 — 28.06%
category statiscs of inner method comments
InformativeAndClarification=210, Noise=35, CommentedOut=10, Intent=208, Misleading=4, Journal=1, TooMuchInfo=6, AlgorithmSummary=241,BetterNameNeeded=7, Redundent=60, Amplification=57, ClosingBracandposition=23, TODO=14

4.3 Threats to Validity

Insufficient Dataset: The amount of comments we manually classified was limited. This could cause bias of the study. Misclassification: These classification are all made by a single
person. They are not verified by author of the code or other experienced software engineers. The classifications may have bias.
4.4 Discussion

Summary: The lines of comments in the studied projects and in sampled comments are 30% of total lines of code, consistent with reports by others. The prevalence of clearly defined bad comments is common and makes up to 30% of total number of comments. Here the uncategorized comments are usually the Algorithm Summary type, and they can be regarded as bad comments (as discussed in Chapter 2). That brought the prevalence of bad comments to 40%. Good comments make up to 60% of total number of comments.

The Javadoc comments have a higher ratio of good comments (68%) and a lower ratio of bad comments (32%). In the contract, the inner method comments have a lower ratio of good comments (56%) and a higher ratio of bad comments (44%). Algorithm comments composed a significant portion of bad comments in the inner method comments: 64.5% of bad comments in the inner methods, and 28% of total inner method comments.

The major type of bad comments in Javadoc comments is the redundant comments (70% of bad comments in Javadoc), whereas the major type of bad comments in inner method comments is Algorithm Summary comments (64.5%).

Overall, the prevalence of bad comments is very common, which can be a problem for code evolution and maintenance. Based on data shown here, it is reasonable to suggest that by avoiding redundant comments and algorithm summary comments, a major portion of bad comments can be avoided in coding.
CHAPTER 5

Source Code Comments and their Association with Complexity and Quality

Our studies above showed that non-Javadoc comments are more prone to be bad than the others. Nearly 44% of them can be considered bad. That raised another question as to whether these comments are emblematic of the quality and complexity of underlying code that they associate with. This type of exact correlation has not been studied before. Also, Because Javadoc comments are primarily targeted for application programmers, who care about how to (re)use the public APIs rather than implementation of the code. Therefore, we wanted to study how the quality and complexity of code impact the intensity, i.e., number, of comments. That is, we investigated the following research question of the thesis:

*RQ3:* How do source code comments associate with the complexity and quality of real software systems from the open source domain?

The quality metrics are calculated on implementation details. Therefore, In this chapter, we only used non-Javadoc comments to investigate the correlation between quality metrics and number of non-Javadoc comments. A high quality code is defined here as having high cohesion inside the classes, low coupling among different classes, and low complexity. Therefore a high quality code is expected to better explain itself and may have a less need for comments. On the other hand, a low quality code has low cohesion inside the classes, high coupling among different classes, and high complexity, and therefore may require more comments. Here, we hypothesized that more comments are correlated to low quality of code, and less comments in high quality code.

5.1 Data and Methodology

The quality of code was assessed in this context at class level and was measured by 4 metrics using an automation open source tool Chidamber and Kemerer Jave Metrics 1.9 (ckjm 1.9) [13]. The number of comments in a project was correlated to each of the 4 metrics using Pearson
correlation ecofﬁciency. Five open source projects were studied. Pearson product-moment correlation coefficient (correlation coefficient for short). Correlation coefficient is a measure of the strength of the linear relationship between two variables. The range of correlation coefficient value is between -1 to 1. 1 means perfect positive linear relationship, -1 means perfect negative linear relationship and 0 means no linear relationship. correlation coefficient was calculated as:

Given a pair of random variables \((X, Y)\),

\[ r = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \]

Where:

- \(\text{cov}(X, Y)\) is the covariance
- \(\sigma_X\) is the standard deviation of \(X\)
- \(\sigma_Y\) is the standard deviation of \(Y\)

P value is used in hypothesis tests to determine whether should reject or accept the null hypothesis.

In our correlation coefficient test H0: there is no linear relation between each quality metric and the number of non-Javadoc comments H1: there is a linear relation between each quality metric and the number of non-Javadoc comments A small p-value (0.05) is an indication that the null hypothesis is false. Thus, If the p-value is smaller than 0.05, we can say that metric was correlated with the number of non-Javadoc comments.

Number of comments are counted instead of lines of comment. Because several in-line (one line) comments (commented by “//”) are commonly seen in the source code, they were combined into one integrity comment and counted as one comment if: 1) they are in-line comments 2) comments are in succession position, one comment is immediately after another.

Code quality was assessed in this context at class level, and was measured in 3 aspects:

**Cohesion:** Cohesion is the degree of how much the methods within one class are related to each other. It is used to make sure the class is serving single, well-focused responsibility. Higher cohe-
Cohesion is desirable because it promotes encapsulation. Cohesion is measured here by Lack of Cohesion of Methods (LCOM) metrics [5]. Here, we used the original version. Consider each pair of methods in one class, investigate the variables used in each method. Let $P$ = the number of method pairs in which there is null intersection of variables. $Q$ = the number of method pairs in which there is non-empty intersection of variables. $\text{LCOM} = P - Q$, if $P > Q = 0$, otherwise. $\text{LCOM} = 0$ indicates a cohesion class. High LCOM value means poor code quality, increases complexity and thereby increasing the likelihood of errors during the development process.

**Coupling:** Coupling is the degree of one class knowing about another class. Two classes are coupled when method defined in one class use methods or instance variables of defined by the other class. It indicates the influence on one class if another class changes. Low coupling is desirable. We use Coupling Between Object (CBO) [5] and Response For a Class (RFC) [5] to measure coupling. CBO counting the number of classes that the one class coupled. The high CBO value indicates high coupling of this class. Another metric used to measure coupling is Response For a Class (RFC). The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class.

Let $P$ = set of methods in the class. $Q$ = set of methods directly called by methods in $P$. $\text{RFC} = |P| + |Q|$. same method only be counted once.

Low RFC value is desirable. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. [3]

**Complexity:** Weighted Methods per Class (WMC) is a count of methods implemented in a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity)[5]. WMC is a predictor of how much time and effort is required to develop and maintain the class. A large WMC also means a greater potential impact on derived classes, since the derived classes inherit (some of) the methods of the base class.
5.2 Results

We asked the research question that: Is low quality code (high LCOM, CBO, RFC, or WMC) commented more? The intuition answer of this question is yes. One reason to write comments is to improve the readability of the source code. If the code itself is more complicated, high coupling among classes, and low cohesion within a class, we tend to write more comments to clarify or reveal the intent in natural languages. We investigated 5 open source projects, calculated the cohesion, coupling and complexity metrics and the number of comments to reveal their relationships.

The data in table 5.1 demonstrate that there are linear relation existing in each of these four quality metrics. However, the ecoefficiency level vary in different projects and different metrics. RFC(Coupling Between Object) and WMC(Weighted Methods per Class) have higher overall high correlation coefficient values, and LCOM(Lack of Cohesion of Methods) has a low correlation coefficient in each of five projects.

5.3 Threats to Validity

Insufficient Dataset: with the time limit, we could not expand our test on vast open source projects. Non-Linear correlative: we could not apply non-linear test on our correlative study.
Table 5.1: Correlation coefficient of LCOM, CBO, RFC and WMC with number of comments

<table>
<thead>
<tr>
<th>Project</th>
<th>CBO</th>
<th>WMC</th>
<th>RFC</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hibernate</td>
<td>0.45</td>
<td>0.66</td>
<td>0.53</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td>2.2e-12</td>
<td>4.1e-29</td>
<td>8.2e-18</td>
<td>1.9e-06</td>
</tr>
<tr>
<td>Cayenne</td>
<td>0.53</td>
<td>0.44</td>
<td>0.68</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>8.9e-254</td>
<td>1.8e-167</td>
<td>0</td>
<td>2.5e-91</td>
</tr>
<tr>
<td>Maven</td>
<td>0.52</td>
<td>0.75</td>
<td>0.59</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>2e-67</td>
<td>9.3e-173</td>
<td>8.2e-92</td>
<td>2.2e-18</td>
</tr>
<tr>
<td>Flink</td>
<td>0.42</td>
<td>0.75</td>
<td>0.78</td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td>6.4e-63</td>
<td>5.3e-175</td>
<td>6.3e-196</td>
<td>9.6e-53</td>
</tr>
<tr>
<td>Spark</td>
<td>0.47</td>
<td>0.54</td>
<td>0.54</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>6.2e-47</td>
<td>1.9e-52</td>
<td>9.5e-64</td>
<td>1.3e-39</td>
</tr>
</tbody>
</table>
CHAPTER 6

Related Work

6.1 Are comments helpful for software comprehensibility?

Early empirical researches in 80’s and 90’s showed that code comments could help to improve the comprehensibility of source code. Woodfield[21], Norcio [9], Tenny [20] and Takang[15] designed different experiments that let students read source code with and without comments, and then they are asked to answer questionnaires or perform certain tasks on the source code. Code comprehensibility was measured by the accuracy of students’ answers which they were asked after they read the code. All these studies have reported that comments have a significant positive effect on program comprehension. Some recent studies focused on the different types of comments or investigated on different types of participants. They found that comments or some comments are less important for source code understanding than it was believed before. Nurvitadhi[10]’s experiment investigated the effects of class and method comments on Java program understanding among students. The results showed only method comments helped students answering low-level questions of the program while class comments did not increase high-level understanding. Salvíulo[12] did experiment among groups of students and professional programmers to understand the role of source code comments and identifiers in source code comprehensibility and maintainability. The result showed that professional programmers preferred to refer identifier names rather than comments, while only the students emphasized the importance of comments. Buse and Weimer[4] proposed a metric of code readability. In this study, comments as well as other local code features were associated with human labeled code readability. The study found that comments are moderately (33% relative power) correlated with readability.
6.2 Comments and Code Quality(Fault-proneness)

Aman studied correlations in comment lines and fault-proneness. He believes some well-written comments are a kind of “air freshener” that used to cover smelled code [1][2]. The appearance of well-written code might reflect a lack of confidence about the program clearness. He did empirical experiments on both file(module) level and method level. He found that in file level, well commented modules (line of comments is greater in the module) are more faulty (has bug fix after release) than none commented modules. In method level, methods have inner comments (comment appears inside method body) are more faulty than methods have none inner comments. and more-commented methods are more likely to be fixed later. Miyake el.[8] Replicated this study to find out the relationship of comments and fault-proneness of code (including self-admitted technical debt (SATD)comments which is defined as the kind of comments that confessing a technical debt such that the commented code need to be fixed later and excluding this SATD comments). the purpose of this study is to find out whether this positive correlative of comments and fault-proneness is due to SATD comments. the result showed that even without SATD comments, the presence of comments in both file level and method level, is related to more code fixing.

6.3 Detecting bad Javadoc comments

Tan et al.[19]analyzed Javadoc comments to identify inconsistencies between the comment and the code. They extracted properties of the method with its corresponding Javadoc comment. It then generated random test cases for the code to identify inconsistencies.

6.4 Comments classification

Steidl et al.[14]proposed model of assessing the quality of comment based on different comment categories. They developed a classifier based on machine learning technique and tested on Java and C/C++ programs. Despite the quality of the work, their classification was based on comment syntax and position in source code, i.e., inline comments, section separator comments, task comments. Pascarella and Bacchelli[11] proposed a new categorization. The comments were
annotated into 6 major categories: Purpose (explain the functionality of the code), Notice (warnings, alerts, and information about usage), Under development (todo and incomplete comments), Style and IDE (IDE directives and formatting text), Metadata (license, ownership etc), and Discarded (noisy comments). Each category is further divided into finer sub-categories. However, their categorization is in line level.
CHAPTER 7

Conclusions and Future Work

This thesis evaluated Martin’s principles as for good or bad comments, and confirmed the usefulness of these principles in classifying good and bad comments. The thesis also discovered a common prevalence of bad comments in open source projects, which can be a potential problem in code evolution and maintenance. This thesis also found a correlation between code complexity and the number of non-Javadoc comments in the code, particularly in high complexity code. Whether these comments are good or bad needs to be further classified, and the number of good or bad comments in correlation to complexity metrics is worth further investigation. If such correlation exists, then the amount of comments (or amount of good or bad comments) can be used as a marker for quality of the code. Our study in the categorization of 2,850 comments can be used as a training set for development of an automation classifier by machine learning, which can conveniently distinguish good or bad comments. Our study here can serve as a basis for development of automation classifier for good and bad comments, and for the future research in the relationship of quality of comments and complexity metrics.
LIST OF REFERENCES


LIST OF REFERENCES (continued)


