Journal of Management Practices, Humanities and Social Sciences

Vol 6 Issue 1 pp. 88-105

https://doi.org/10.33152/jmphss-6.1.9

ORIGINAL CONTRIBUTION

Code Comment Analysis–A Review Paper

Syed Zohaib Hassan[1*], Ayesha Irshad[2], Jamaluddin Mir[3], Ayesha Aslam[4], Kalsoom Ayaz[5], Muhammad Awais Bawazir[6]

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

[2, 4, 5] Department of Computer Science, Abbottabad University of Science & Technology, Abbottabad, Pakistan

[3] Faculty of Computer Science & Information Technology, Universiti Tun Hossein Onn Malaysia, Batu Pahat, Malaysia

[6] Department of Computer Science, Bahria University, Islamabad, Pakistan

Abstract— The compilers are manufactured in such a way that they ignore most comments in software systems' source code. In addition to being a key source of system documentation, code comments are essential for both establishment and improvement. System comments or just quantitative assertions regarding the quality of the program are presently the only known techniques for evaluating software quality. In software development, comments are being used as a regular practice to enhance the clarity of code and to transmit the enthusiasm of programmers in a more conveyed manner. Whereas programmers seldom bother to keep their comments current. Comments are an important source of information about how the framework works. Various disciplines have concentrated on the content of the online client comments in different contexts, utilizing manual quantitative/subjective or (semi-)automatic methods. The wide variety and disciplinary partitions make it hard to get a handle on an outline of those views which have proactively been inspected. The huge number of daily comments inundating the newsroom can be amazing, particularly when a huge chunk is unfriendly or "contaminated" in content and tone. When dealing with complex documents such as source code, it can be hard to link the dots between the practical linguistic information contained within the code as well as the corresponding textual explanation found within the code, making it unsuitable for use in program analysis and mining assignments. Analysis of code comments on software improvement is examined in this research. Studies on code comments have been summarized in this paper, which covers four main areas: relevance of code comments, quality of code comment sources, code comment analysis, as well as a research approach for code comments and difficulties. It provides more comprehensive information for future research by analyzing effective methods for this study issue.

Index Terms— Code comment, Comment analysis, Software, Code mapping

Received: 19 October 2021; **Accepted:** 11 December 2021; **Published:** 22 January 2022

Introduction

Code comments are an ordinary practice in software development. Comments describe code relationship, code evaluation, Communications with a developer's programming language, semantic behavior, annotation language, and code arrange relatively unnecessary and

*Email: syedzohaibhasan@hotmail.com

independent information for the program. Software includes a huge number of comments. Various projects are written differently, like a programming language, server, operating system, desktop applications, free BSD, and open Solaris (Mikolov, 2013). In software systems, a large portion of the source code is made up of comments, which record the implementation and aid developers in understanding the code for future modifications or reuse: Researchers have found that code with comments is more readable than code without comments. After the code itself, comments are the second most often utilized documentation item for code comprehension (Pressman, 2005).

Additionally, a system's source code must be documented to ensure its long-term viability. To keep documentation and code up to date, developers use comments in source code rather than relying on external documentation. A lack of generic documentation leads to misconceptions, and studies demonstrate that inadequate documentation considerably reduces the maintainability of software. Developers sometimes overlook commenting code because of deadlines and other time constraints throughout the development process (Vermeulen, 2000). Code comments are more clear, more descriptive, and easy to understand. Comments and code provide redundant and independent data regarding a program semantic behavior of a program. Code commands include a lot of information that can be used to improve the maintainability and dependability of a program. Because of the large number of code comments available, analytics such as Natural Language Processing machine learning approach is in high demand (Pressman, 2005).

Reliability is very important in software operating system servers and desktop applications. For example, software bugs that affect the server, like operating system, software failure, fewer users, and a huge impact on finances (Cusumano, 1995). Code comments pass on valuable data approximately the framework functionalities, so numerous approaches for computer program designing assignments take comments as a critical source for code semantic investigation (Corazza, 2011). To the best of our information, recognizing the scope of comments can contribute to the taking after the program designing task (Paul, 1994).

Most software projects do not include full comments and documentation that significantly impair the readability and maintenance of the program. Scholars have thus attempted to comment on code using human and automatic techniques. The automated approach mostly covers how to multiplex and extract comments (YU Hai, 2016). The commenting method coincides with the code segments to be commented on by other Question Answer systems first. It acquires the correlating descriptive text, a code description mapping, eventually processing the text description utilizing natural language processing techniques.

In addition, there is no comprehensive model of comment quality. There is a lack of depth and accuracy in coding rules on the issue of commenting code. So yet, no automated techniques for assessing comment quality have been created since comment analysis is a complex problem: Aside from using syntactic delimiters, comments are made entirely of natural language. Since algorithms are heuristic, their solutions are also (Corazza, 2011).

Current quality analysis approaches do not include system comments or restrict them to the comment ratio statistic. According to software quality, most source code descriptions are ignored. The current study focuses on all the data related to the Code Comment Analysis; all previously published research is included in this study. The first section of this study is relevant to the study related to code comment analysis, and the second part covers the comments classification. Different sources evaluate the importance of the third section of the study.

The quality of source code comments is also discussed in the fourth section of the study. The important section is the code analysis which is detailed discussed. Researchers have undertaken code comment studies from different angles in recent times that can approximately be split into different ways automatic generation of code comments, consistency of code and comments, classification of code comments, and quality evaluation of code comments. Furthermore, the Quality of code comments was discussed, and assessment metrics were studied well. The study summarizes key issues in code comment analysis, solutions, and applications.

Methodology

The study's goal was met by conducting a comprehensive literature review. Code comment analysis, comment categorization, and the quality of source code comments are all examined in this study's methodology, which is based on synthesis methodology, which contains a procedure and findings from all accessible research connected to the issue of interest. It's possible to roughly divide recent code comment research into four categories: automated production of code comments, consistency of code and comments, code comment categorization, major difficulties in the study of code comment solutions, applications, and assessment of the quality of code comments.

Search strategy

The comprehensive evaluation of all international scientific papers published up to July 2022 in higher education focuses on Code comment analysis. We employed Web of Knowledge, SCOPUS, and ERIC databases as our primary sources during our research. Among the keywords used to locate relevant papers were "code comment," "code comment analysis," "code analysis," "comment analysis," and "code comment categorization." Combining keywords also helped restrict the search. The search criteria were incorporated in the titles, keywords, and abstracts (based on the searches permitted by each evaluated database).

Eligibility criteria

Research articles irrelevant to the study were discarded as a first step. Our research then focuses on articles that include "Code comment analysis" in the title or abstract. At various times in the project and development, data extraction and analysis were performed on documents that met the inclusion criteria. This collection contains a total of 16 documents.

Selection process

Searching through all three databases, the initial search turned up 100 papers that needed more investigation. Figure 1 illustrates the manuscript selection process.

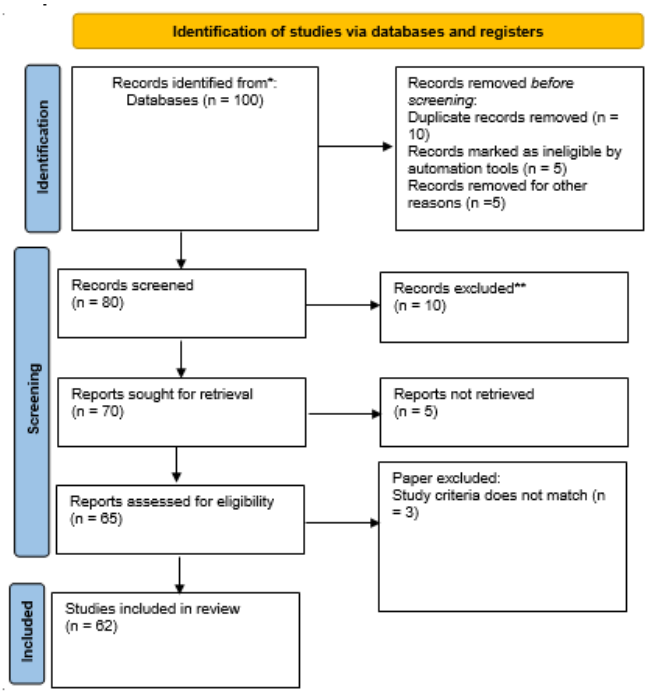

Fig. 1. Flow chart diagram of study

Related Studies

General comment analysis, comment development research, and code recognition algorithms are all subcategories of similar studies.

Comment analysis

In 2010, Khamis and colleagues created Javadoc Miner, an application that analyses Javadoc comments for quality. They use a set of simple criteria for assessing both the quality of the comments and their compliance with the code. They're curious about the same thing we are: how to gauge the quality of comments left by others. They did not investigate whether metrics like reading indexes, nouns, verb count heuristics, or acronym count heuristics can effectively estimate comment quality. ' It also doesn't discriminate among different types of comments or indicates any problems among code and comments from outside Javadoc's structural limitations."

Hu (2018, 2020) has previously studied the storage and analysis of task feedback. Task comment quality can only be assessed manually or semi-automatically. We provide a more comprehensive assessment of comment quality instead of only assessing the quality of task-specific remarks. Jiang (2006) study the feasibility and benefits of comment analysis to find code problems. They uncovered 12 vulnerabilities in the Linux kernel using this approach, two of which have been confirmed by the operating system's developers.

Freitas (2020) did study on code comments analysis using contextualized vocabulary. By studying code comments, they could build a better lexicon for identifying items of admitted technical debt. Using pattern-based code comment analysis, the researchers discovered that finding and categorizing technical debt issues might be easier. Automated comment analysis is required based on such instances.

Thus, the emphasis is on synchronization in the investigation. In contrast, we use a method that analyses comments independently of their context.

Table I
Previous publications on comment analysis

Authors	Research Methods
(Freitas 2020)	code comments analysis using contextualized vocabulary
(Khamis, 2010)	Javadoc comments on quality evaluation
(Svyatkovskiy, 2020)	Comments storage analysis
(Yang, 2019)	Comment analysis feasibility and advantages
(Tan L., 2015)	Extracting Code Segments by autonomic method
(Panichella, 2012)	Finding code explanations from developer communications

Information retrieval techniques

The comments, provided "the code is of excellent quality," offer an accurate account of the code. Information gathering approaches based on similarity measures for vector space models are used by Roy (2007) to assess the quality of functions. It's like this study, which investigates the relationship between source code and comments.

A collection of statements neglected in this research are examined in further detail by analyzing their relationship to the technique name. Code-to-documentation links may also be tracked using information retrieval techniques (Gašević, 2009; Williams, 2005). Such methods, on either hand, focus more on the fundamental paradigm of the link between the code and the documentation than on comments.

Table II
Previous publications on retrieval techniques

Authors	Research Methods
(Roy, 2007)	Cosine similarity-based vector space model retrieval techniques
(Palomba, 2018)	Detection of test smells by information extraction
(Gaševíc, 2009) & (Williams, 2005)	Retrieval algorithms to track out code-to-documentation relationships

Evolution of code and comments

Remark evolution is examined by Jiang (2006) to examine the general assertion that developers update code without changing its related comment. The research, on the other hand, shows that programmers often update the function comments. In Fluri (2007), the authors investigate how code and comments evolve independently. According to the results, research shows that 97% of all comments are changed in the same revision as the source code changes. They also look at how much work developers put into code commenting over time to see whether this ratio has an upward or downward trend. There are exceptions to this rule, such as where copyright or commented-out code should be omitted from this metric (Tan, 2007).

Table III
Previous publications on the evaluation of code and comment

Authors	Research Methods
(Jiang, 2006)	Examine the prevalent assertion that developers update code without changing its related comment
(Fluri, 2007,)	Explore how they develop code and comments
(Tan, 2009)	Copyright or commented-out code should be omitted from this metric

Source code recognition

In Section IV, source code recognition is included in comment classification. (Madani, 2010) used support-vector machines (SVMs) with a precision of 92.97 percent and a recall of 72.17 percent to detect code during email data cleaning. A recall is increased while accuracy is maintained by using comparable machine learning approaches. Email code can be identified with 94% accuracy and 86% recall using lightweight approaches (regular expressions and pattern matching). The accuracy and recall of this strategy and ours are almost the same. Code snippets appear in comments but not in emails.

Comment classification

No parser or programmer can classify comments based on grammatical rules since all comment types share the same syntax. This problem can only be solved using heuristic approaches. Due to the large number of criteria that influence the categorization decision, we employ machine learning approaches to categorize comments automatically. We construct seven distinct comment categories. The existing machine learning library WEKA1 is used for implementation. There are seven distinct sorts of comments in the Java and C/C++ programming languages: • Copyright Comments: The source code file's copyright or licensing information is included in copyright comments. In most cases, they may be located near the top of the file. • Header Comments: In addition to providing information about the modification number, class's creator, or review status, header comments offer a summary of the class's functionality. After the importation, but before the class definition, Java headers are detected. • Member Comments: Member comments are inserted on the same line as the member declaration and explain the method/functionality. Fields help the developer with the project's API. • Inline Comments: Inline comments describe a method's execution choices in the method's body. • Section Comments: Several fields/methods relating to a single functional aspect are discussed in section comments. As an example, consider the following: // ---- Setter and Getter Methods --- And several setters and getter methods follow. • Code Comments: The compiler ignores a "commented out" piece of code. Temporarily commenting out code for troubleshooting or future use is rather frequent. • Task Comments: Todos, bugs, or implementation hiccups may all be found in task comments, including a developer message.

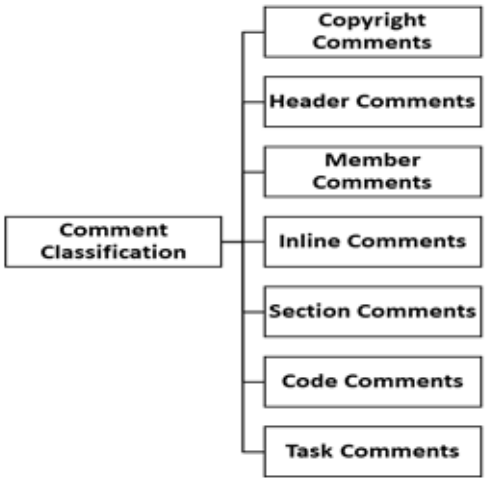

Fig. 2. Comment classification

Importance of Code Comments

Annotation is the process of inserting a human-readable description (HRD) into a computer program to explain the purpose of the code. Correct use of comments can make code maintenance easier and help detect errors faster. In addition, comments are essential when writing functions that others will use. Remember, well-documented code is just as important as well-run code (Van Roy, 2004). Many comments explain code relationships, code evolution, or how to use and meaning of integers and integer macros. Existing comment languages can express a large number of comments. Over comments express issues related to timing, but the comment language is not adequate (Kagdi, 2007).

Software repositories

Ongoing investigations are being conducted into software repository structures. Tools for analyzing software repositories have been created to help programmers understand and evolve their code more effectively and efficiently (Kagdi, 2007). While looking at the co-evolution of source code as well as the comments that accompany it in software repositories, we had to connect source code objects to the comments that appeared there (Fluri, 2007).

Learning comments

Our goal, dubbed "learning to comment," is to automate the process of making comments. As a machine learning model, learn common commenting techniques. After that, use the model to help developers make commenting judgments through the development process. The essential notion is depicted in Figure 1. Underlying our strategy entails guessing whether the present line will continue depending on the facts and an appropriate commenting site.

```
boolean hasBufferArgs = false;
if (views != null){
 for (View view : views){
  if (view != null){
    hasBufferArgs = setBuffer(true);
    break;
  }
  view.setAlwaysOnTop(false);
  view.setBuffer(false);
  buffer = jEdit.getFirstBuffer();
 }
}
```

Fig. 3. Essential notion

Code snippet

Software developers commonly comment on a functionally (or logically) independent code snippet (Mikolov, 2013). If the current line is the first line of a functionally independent code snippet, it is almost certainly a comment placement (Roy, 2007)

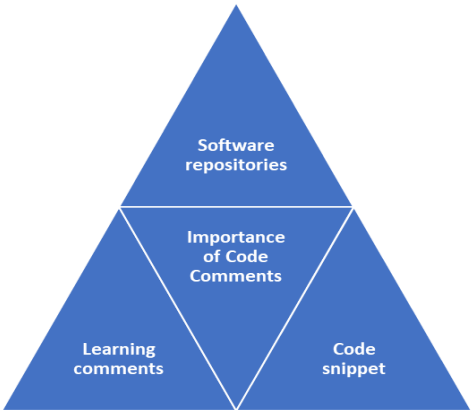

Fig. 4. Importance of code comments

Analysis Quality of Source Code Comments

Research and analyze the benefits of code comments which help in the following aspect.

Software bug reliability

You can look over the comments to detect bugs in the source code
/* This function must be
/* This function must be
/* This function must be
/* This function must be
/* This function must be

*/ The interruption context cannot be used to call this method. According to (Van Roy, 2004) Comment), just 1% of the data was used. Lock-related problems may be discovered automatically using Linux kernel comments. as well as issues with phone calls. Though it has been shown that comments may increase software stability, comment has not yet been implemented.

Programming language

Feedback can inspire new programming language extensions or the design of new programming languages. The comment in OpenSolaris shown below specifies the field name to which the value is assigned; for example, 15 is assigned to the length field. Specifying such information in comments is inconvenient and prone to errors when the structure definition changes. The field name in the code, for example, length = 15. This example shows that programming language extensions have solved some of the comments' requirements. More extensions of the programming language can be designed by studying the comments.

```
const struct st_drivetype st_drivetypes []= {. . .
"Unisys . . . .", /∗ name . . . ∗
15,          /*. length . . . * / ...
```

Mining detection specification and comment analysis code for error

This is the main content of this chapter. This section uses examples to introduce automatic annotation analysis for specification extraction and error detection. As mentioned above, there are a lot of notes in the software that contains a lot of information. Some information is only available in the code because the developer will not repeat all information in the code in the comment (Gašević, 2009). On the other hand, comments contain information that cannot be easily extracted from the source code.

For example, changes that need to be made together (/* Warning: If you change any of these definitions, make sure to change the definition in the X server file (radeon_sarea.h) */, there are still tasks to be processed (/* FIXME: We should group addresses here. * /), the unit of the variable, the reason for not selecting a specific algorithm, and the author information of the file. In addition, comments and codes also contain redundant information (Gašević, 2009). For example, the comment in OpenSolaris states that the held lock must be used to call the taskq_ent_free (Williams, 2005) function and asks to create a common () to acquire the lock before running the taskq_ent_free (Gašević, 2009) function.

There is a lot of repetitive information in the comments and excerpts. The redundancy in this example is consistent. Figure 1 shows an example where the remark doesn't fit the code, which is undesirable. Checking for errors in the comment code is made easier because of the program's redundant information on its semantic behavior. Commentary often lags behind the source code during software development (Padioleau, 2009). Code that doesn't follow commented comments is an error, whereas code that does follow correctly commented comments are both inconsistent (Tan, 2009).

Fig. 5. Analysis quality of source code comments

Analyzing Code Comments

Analysis comments can extract specifications, also known as programming rules, which can be used.

- To detect errors and incorrect comments

- To assist developers, prevent new problems, and better comprehend the code, enhancing the program's dependability and ease of maintenance.

In order to prevent unauthorized access to data provided inside reset hardware (), the function's caller must get a lock, as shown in Figure 5. In this case, the comment specification does not apply, and the code may violate the specification. As seen in Figure 5, this is a 2.6.11 kernel bug. Before executing the in 2000 bus reset method, the lock was not released (Tan, 2015).

We looked at the generalizability of our approach and its relevance to two specific positions in the field of software engineering. There are two possible explanations for the selection of such two. In addition, the scopes of work information were used in both jobs. Block or line comments. There are a few considerations to consider while translating code to block/line comments.

The second benefit is that their data and code are free to use and develop. We have to perform three things to do this. Research-based on facts As part of the first research, we calculated the accuracy approach to see how successful and resilient our technique was. Furthermore, the selection of features and comment scope sizes are crucial. Lastly, we evaluated the equivalent measures of our technique on projects that had just been started up. Second, we utilized our way to check for coherence in the comment code and compared it to the originally suggested method.

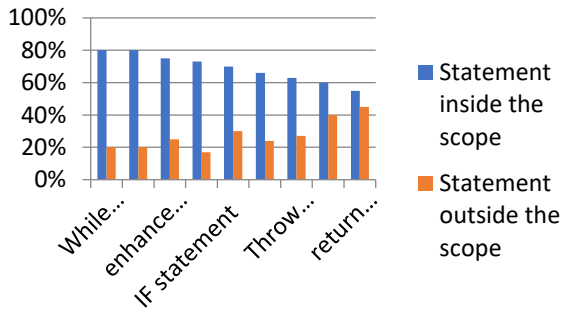

Fig. 6. Statements inside and outside of the scope

The accuracy of our approach in recognizing comment scopes is something we'd want to test. To begin, we used a classifier to separate statements into two groups: those which fit within the comment's scope and those that are outside the comment's scope.

Table IV
Comment code

C1	There is nine type of statements 1) If-statements, 2) While-statements, 3) For-statements, 4) Enhance For-statements, 5) TryCatch-statements, 6) Variable Decl
C2	The number of the sub-statements of the current statement
C3	The number of layers of statements nested in the current statement
C4	The number of lines of code in the current statement
C5	Does the current statement have the same method calls with the last statements and the following statements
C6	Same variables Does the current statement use the same variables with the last statements and the following statements
C7	Is the preceding line of the current statement blank
C8	Is the following line of the current statement blank

The second component is the ability to leave comments on posts. This data is gained via comment features. We employed the following natural language processing methods to preprocess comments.

- Word Splitting
- Stop word removal
- Word stemming

The number of words, verbs, and nouns in the remark was then tallied. Depending on our studies on posting behavior, we noticed that a comment preceded by a blank line is frequently used to summarise a block of code. as a result, I was wondering if a comment's next line is blank or not. It's now part of the commenting system.

Table V
Comment feature

No.	Feature	Description
Cml	Length of the comment	The number of words in the comment
Cm2	No. of verbs	The number of verbs in the comment
Cm3	No. of nouns	The number of nouns in the comment
Cm4	Following blank line Is	The following line of the comment blan

Research Method

Researchers have undertaken code comment studies from different angles in recent times that can approximately be split into the following ways: automatic generation of code comments, consistency of code and comments, classification of code comments, and quality evaluation of code comments.

Automatic code comment generation

Automatic Code Comments generation, i.e., code segments comparable to source code are identified in the target program as per existing code comments, and would then source code segments comment be matched by a series of processes to the code segments of the target software. Code comments enhance the maintenance of software by enabling engineers to comprehend code. Although code comments are necessary and important, several code libraries do not include adequate comments. Many academics have investigated methods to extract code comments (Table – 4).

Table VI
Previous publications on automatic code comments generation

Authors	Research Methods
(Song, 2019)	Algorithms and techniques
(Liu, 2019)	Automatic code Comments generation by pull request
(Liang, 2018)	Automatic generation of text descriptive comments for code blocks
(Chatterjee, 2017)	Extracting Code Segments by autonomic method
(Panichella, 2012)	Mining source code descriptions from developer communications[C]

Code comments may be automatically generated in the following categories.

Method based on templates

Sridhara (2010) also proposes a novel approach for automatically generating descriptive comments for Java method parameters. They utilize heuristics to produce observations that can be readily incorporated into the IDE to offer an up-to-date parameter description when a software developer edits a particular procedure. (Wong, 2013). offer a technique for automatically generating code comments by removing large-scale data from the Stack Overflow Question and Answer (Moreno, 2013). offer a technique for generating structured Java classes' natural language summaries automatically. McBurney (McMillan, 2014) proposes a novel way to generate Java method summaries automatically. This method describes the method context, not the specifics inside the method.

Table VII
Previous publications on a method based on the template

Authors	Research
(Hu, 2018)	Deep code comment generation by using the template method
(Syriani, 2018)	Template-based code generation Systematic mapping study
(Sridhara, 2010)	With the use of Java methods, generate summary comments
(Wong, 2013)	Mining question and answer sites for automatic comment generation
(Moreno, 2013)	Natural language automatic generation of
(McMillan, 2014)	source code summarization

Method based on keywords

To enhance technology, (Rodeghero, 2014) offers a programmer survey throughout the aggregation process of source code, which is an instrument to pick keywords based on the eye tracking findings of the study. Sonia (2010) are studying the use of automated text summary

methods to produce comments with the source code. Laura Moreno et al. (2013). provide a comprehensible how to create Java classes automatically.

Table VIII
Previous publications on keywords

Authors	Research
(Hu, 2020)	Code generation with hybrid lexical and syntactical information
(Svyatkovskiy, 2020)	Code generation using a transformer
(Rodeghero, 2014)	Source code summarization via a keyword
(Sonia, 2010)	text summarization techniques for summarizing source code
(Lauramoreno, 2013)	Automatic generation of natural language summaries for Java classes

Method based on machine learning and neural network Most techniques known as machine education and neural networks teach code comments, get training data and map these to uncommented code segments autonomously. The commentary on the subject model and n-gram is predicted by (Movshovitz-Attias, 2013). Allamanis (2015) offer a continuous embedding model, like a Source Code summary recommending correct methods and class names. Iyer et al. (2016) propose a novel CODE-NN model that utilizes a network of Long and Short Memory (LSTM) to create phrasing for C# code snippets and SQL queries. Adrian (2007) utilizes the ID and comments to mine the source code repository topic. Punyamurthula (2015) uses the call graph to collect metadata and source code dependency information and then utilizes this to evaluate and subject to source code.

Table IX
Previous publications on methods based on image

Author	Research
(Liu Yihong, 2018)	Image Feature Extraction method
(Qin Ming, 2018)	Image Semantic Comment
(Li Hongwei et al., 2016)	Code semantic annotation method

Method based on image

The image-based approach splits the picture by extracting characteristics and then automatically combines the words of the remark to comment on the image (Liu, 2019). offer image extraction and semantic comment techniques based on visual memory utilizing human visual memory mechanisms and mechanisms to solve the issue of extraction and marking of image functions. Based on multimedia-related models (Qin Ming, 2018.) present a novel photo memantine comment technique that fuses information in the picture category and improves comment outcomes by utilizing an association rule mining algorithm.

Table X
Previous publications on machine learning and neural network

Authors	Research
(LeClair, 2020)	Improved code summarization via neural network
Xu (2019)	Commit message generation for source code changes
(Movshovitz-Attias, 2013)	Natural language models for predicting programming comments
Allamanis (2015)	text summarization techniques for summarizing source code
Iyer (2016)	Summarizing source code using the CODE-NN model
Adrian (2007)	Identifying topics in source code
Punyamurthula (2015)	Dynamic model generation

Li Hongwei (2016) proposed a technique of code labeling. This means the obtaining of code grammar information through the use of the common software reverse analysis technology and the use of IR technology to process the source code text. Then it acquires the terms for applicants' business based on them and uses a domain ontology to retrieve the concept of ontology for applicants' business terms.

Quality of Code Comments

Comment quality may be evaluated quantitatively and qualitatively using our semi-automatic technique. It is possible to analyze and evaluate the quality of comments using a semi-automatic technique. For both Java and C/C++ projects, we use machine learning to categorize the many sorts of comments in the code. One may perform a quantitative and qualitative examination of the system's comment ratio by

classifying comments. Our thorough quality methodology is based on the classification of comments. Consistency across the project, system documentation completeness, code coherence, and reader utility are the four quality factors for each comment category in the model. To analyze quality qualities, we give metrics that identify quality faults in particular comment categories. Separately, the validity and relevance of the measure are assessed: A poll of experienced software engineers is used to determine the validity of the findings. The survey results suggest that metrics may provide extra recommendations for refactoring. It turns out that comment classification provides better information about the quality of documents than just a simple comment ratio statistic and that our measurements identify quality issues in practice (Paul ,2014). Code comments' quality may be evaluated by determining what constitutes the Quality of code comments. As per the author's study (Stamelos 2002), high-level explanations of what the program has been doing must be provided in the comments. It is not necessary to reiterate the "obvious." Most development teams nowadays place a high value on quality. Even more so for those who are creating safety-related solutions. According to Steidl (2013) the quality model can be used to explain the metrics as shown in Figure 6; tasks can be categorized into hierarchical trees, with the roots at the bottom and the branches at the top.

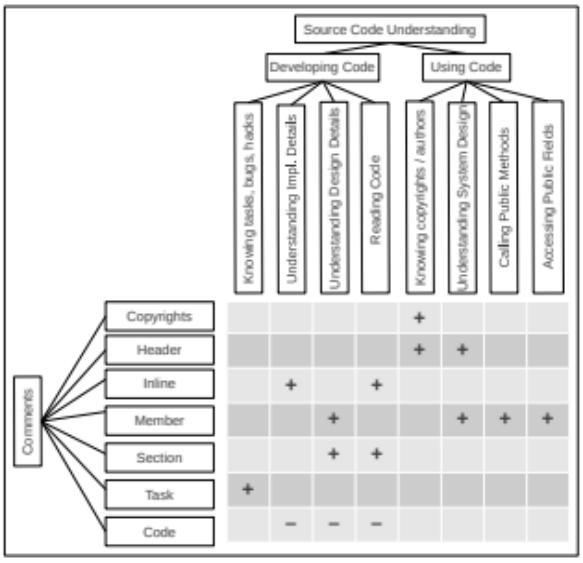

Fig. 7. Positive impacts (+) and negative impacts (-) of entities on activities

Criteria reflect the influence of a certain entity on a given activity by representing quality elements for various entities. It's clear from the study of (Steidl 2013) that comments may significantly influence developers' work. This model is based on the following four criteria:

- Coherence looks at how comment and its code work together. It also looks at how a single comment relates to the rest of the code. Because this is a strong signal that this comment is ready and impactful, member comments must be linked to the method name. This helps you call public methods and know how the system works. Developers also predict member as well as inline comments to explain these things that aren't obvious by giving information that goes further than the code to make them understand implementation as well as design details. Member comments, on the other hand, must give us more information than just having to repeat the method name (Moreno, 2013).
- Usefulness is the things that make a single comment to help people understand the system. Comments must help people understand what a piece of code is for. In general, people who read the code should find the comment useful. It wouldn't be very useful if the comment didn't make it more difficult to understand the source code. The things that make it easier to understand and use code are made easier by clear, helpful comments (Wong, 2013).
- Completeness Putting copyright at the end of each file gives you all the information you need about copyrights, and putting a header at the start of each class shows how the system is built. Writing a remark for every public method and field enables an API user to select which public methods and fields to utilize (Paige 2014).

Items that must be consistent throughout the system are known as consistency qualities. To make reading comments simpler, they must be written in the same language as the code itself (for example, English). Each file should also be under the same copyright and formatted in the same way, which will help people learn about copyrights and authors. Figure 1 shows how the four quality criteria impact each other. In general, we don't like code comments because they don't give us any information. The effect matrix illustrates that only a few factors help individuals grasp the system. Comments in the header, member, inline, and section can aid in the identification of methods and fields, as well as in the comprehension of the system's architecture and the actual operation of the various components

(Sridhara, 2010). Copyright notices and duties should also be included in the documentation. The quality model recommends using automated assessments. Just the consistency category is discussed in this study. Consistency and utility may be found in Ayala (2021).

We may better define the criteria by defining separate attributes for various things. Positive or negative, each characteristic indicates whether the object (comment type) is intended to aid or impede the action, it is linked with. Hand-analyzing the training data allowed us to discover and characterize these effects. Using the four basic criteria as a guide, we explain how each feature affects certain actions (Panichella, 2012).

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

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

Fig. 8. Positive impacts (+) and negative impacts (-) of entities on activities

(Rani 2021) shows the best practices for writing code comments in their study. Firstly, comments should not be the same as the code. They write too many comments because they were taught to do so by their first teachers. Computer science students in the upper levels often add a comment to each closed brace to show which block is coming to an end:

```
if (x > 3)
…
// if
```

I've also heard of teachers making their students write a comment for every line of code they write down. This might be a good idea if you're an extreme beginner, but you should take off your training wheels while biking with the big kids. It takes time to write and read comments that don't add information, which can become out-of-date (Chatterjee, 2017). The canonical bad example is:

```
i = i + 1; // Add one to i
```

It doesn't add any information as well as costs money to keep up (Sonia 2010).

If you write bad code, don't leave a comment about it. You should write new code instead. The third thing to remember is that the code you're developing could have an issue if you can't make an explicit remark about it. If you look at the code for the Unix operating system, you're not expected to comprehend this statement. When it was created, there wasn't a lot of code to allow for context switching among various sorts of contexts. Afterward, he explained that he intended it as "This won't be on the test," not as an "impolite" question (Miltiadis, 2015). Trying to find out what went wrong is twice as hard as developing the code itself, which gets us to Kernighan's Law (Movshovitz-Attias, 2013). If you create the code as intelligently as feasible, you'll never be capable of figuring out how to repair it. " When you turn on your lights and siren in your automobile, you're admitting that you're doing something illegal. Alternatively, rewrite the code in a way that you can understand or that is easy to understand (Tan, 2015).

Lastly, comments are meant to clarify rather than complicate an issue. To illustrate this point, Steven Levy cites the story of an anonymous blogger who was hacked after posting an offensive remark. Because of this, [Peter Samson] was so difficult to comprehend. He didn't explain what he does in his source code. There have been hundreds of commands in Samson's well-known program, and only one remark adjacent to instruction with the number 1750 was only there because it was 1750 (Srinivasan Iyer, 2016). Since 1750 was the year Bach died, and Samson wrote RIPJSB as an acronym for Rest In Peace Johann Sebastian Bach, most people wouldn't know what RIPJSB meant. Since I'm A big fan of a good hack, this was not one of them. You should delete your remark if you're causing more misunderstanding than solving the problem rather than trying to help (Tan L., 2009).

The fifth rule states that comments must be used to clarify any code that isn't clear. A remark such as this should be included in any code that somebody else thinks is unneeded or repetitive:

Though in the absence of explanation, some may conclude that the code has been "refined" or that it is powerful but enigmatic magic. To what purpose did you devote yourself to coding the program? It would help if you didn't have to bother about it for potential readers. Not whether a code explanation is necessary (Adrian et al., 2007).

As with most programmers, many programmers employ code developed by other programmers (Padioleau et al., 2009). The equation is the first place anybody who wants to learn this code will look. A lot of time is saved by copying and pasting the URL rather than searching for the reference afterward. Reusing code is a great idea because it can save you time and put your work with so many more people. Some programmers could not indicate that they didn't develop the code themselves. Nobody should write code that they have been unsure

about. People always steal Stack Overflow solutions to inquiries and paste them into their own codebases. Creative Commons licenses allow you to credit the author of the code. This condition may be met with a reference comment (Williams, 2005).

Linking to standards and other documentation may make it simpler to understand what your code is meant to perform. A well-placed remark helps readers access this information when and where they need it the most. RFC 4180 has been updated by RFC 7111, which is helpful if you click on the link. However, A comment should be included whenever a bug is fixed (Scalabrino, 2019). Sometimes you need to make changes to your code, such as when you patch a problem and find yourself needing to add comments. This remark is critical for several reasons. It aided in comprehending the code in the present and referred to methods by the reader. Additionally, it aids them in determining whether or not the code is still necessary and how to test it (Ming et al., 2018).

If the attributes don't have a title, use the name instead. The last time a phrase or line was added or altered may be found using git blame. Since commit messages tended to be brief, the most significant change may well not have been included in the previous commit that's been made (Li Hongwei, 2016). Use the comments and let others realize if their ideas aren't complete. The need to verify in code with known bugs isn't always a bad thing. Make your code problems obvious by adding an "At Do" not to the end of each line of code (Chatterjee, 2017).

Assessment Metrics

It's possible to use git blame to look up the last time a certain word or line was added or changed. But because commit messages tend to be short, the most important change (e.g., fixing issue #1425) may not be part of the last commit that was made (e.g., moving a method from one file to another). Use comments to let people know that their ideas aren't done. Sometimes, it's important to check in code despite its known flaws. It would help if you clarified your code flaws by adding a "To Do" note (Chatterjee, 2017).

Coherence between code and comments

The coherence coefficient (c coeff) is a way to measure how well member comments, as well as method names, go together. It looks at the attributes of member comments that aren't obvious and are linked to the method name. Illustrates an example where the comment helps to explain what is obvious, which is why it isn't needed in this case. Due to a noninformative comment or a noninformative ID, the comments in Figures 3 and 4 are not related to the method name, which is why they are there. Calc Eigenvalue Decomposition should be the new name of the method in this case. Our new metric will be able to find all three of them. Metric. Comment: We look for words in the comment and compare them to words in the method name. A camel casing method is used to get the words in the method name. Words in the comment are assumed to be separated by white spaces. This is how many words from one set match words from another set. They are similar if the distance between two words is less than 2. It is called the number of similar words divided by the total number of comment words. The coefficient c of the comment is 0.75. Based on some tests with manual evaluation, we set two thresholds and looked at member comments with c coeff = 0 and c coeff > 0.5 (Wen, 2019).

Length of comments

As the second metric, we look at the length of inline comments to see how well they fit into the rest of the code. This shows how well they fit into the rest of the code. This looks at the quality attribute that says that inline comments aren't obvious. Scientists haven't looked into the role of very short or very long inline comments.

Key Issues and Solutions

Comments for automated creation, consistency modification, categorization, and quality assessment are now extensively utilized. It may make it simpler for engineers to comprehend the source code. Comments may also be used as a pseudo code to express purpose before the real code is written. In this instance, the reasoning behind the code should be explained rather than the code itself. There are, however, many shortcomings and limitations regarding the related literature, including several aspects. The automatic creation of code comments is less accurate. Researchers have conducted many studies on code observations in recent years. However, there are still some difficulties in generating code comments, such as less automation and irregular comments for automated creation and decision-making of code comments, for instance, which codes must be remarked on and which comments must be followed. Since the standards are not yet recognized, the accuracy of the final results is not great. Limited situations are available to use the code comments (Antoniol, 2000).

Although the study of code comments in recent years has received significant attention, there are several difficulties in the present research, including unclear methodologies, unambiguous or inefficient findings, and a lack of appropriate tools to suggest and manage comments. The previous study has mainly assessed and forecasted by collecting certain metrics of code comments attributes that are

seldom linked to real software code modifications. There are very few studies on code comment evolution. It takes a lot of time and works for the experiment to investigate and solve these shortcomings that have a major effect on software development and maintenance; since there are no recognized standards, automation and control are not high, and the quality of the results produced is comparatively poor. Therefore, time and labor must be reduced appropriately (Howard, 2013).

Solution

Following the questions posed above, this section offers possible answers and future research directions in response.

- Machine learning, natural language processing, and other cutting-edge technologies may all be used in conjunction with a single algorithm to revolutionize code comment research.
- More specific ways for commenting on code should be adopted or recommended to enhance understanding & reading of code in light of multi-source software products and practical demands.
- Coding in other fields will be encouraged by its broad usage and role in resolving real-world problems. For instance, code comments are often used in the open-source and artificial intelligence domains (Padioleau, 2009).

Application

The study presented here is the first step toward a more thorough examination of code comments. Here, we examine both our strengths and faults. To evaluate the quality of comments, we conducted a quantitative and qualitative analysis: Compared to the prior comment ratio, our comment categorization provides more quantifiable data about how individuals are commenting on the system. Besides a survey and a case study, we provide two measures for identifying quality issues in comments and making refactoring recommendations. A one-time quality audit or ongoing quality checks to monitor how well the system comments over time may use the comment categorization and metrics (Lemos, 2020). C coeff and length indicators may be used throughout development to provide warnings about missing comments or shoddy code in the IDE. It is a frequent use scenario for quality control, although the recommended metrics are semi-automatic and need human intervention. In our study, you can't use this approach to rank the quality of a large number of projects' comments. Also, it doesn't provide a clear view of the quality of the feedback. As a basis for thorough quality analysis, this work provides comment kinds and an evaluation methodology for the quality of comments. As with code quality measures, ours simply highlight issues and cannot immediately identify whether anything is excellent (Lemos, 2020)

Conclusion

Commenting on a program is essential to its development and understanding. Tracking comments and source code items in an automated fashion are seldom simple. The ranges of source code comments may be automatically recognized using a machine learning-based approach shown in this study. Our approach for determining the breadth of a remark was effective because it employed three features. Two software engineering projects were also implemented using our technique. Improved performance and efficiency were achieved in both activities. It would be simpler for other autonomous software strategies to use our comment scope detection approach as a generic approach. In the future, we want to use the information in comments and code snippets to integrate the technology with more automated software artifacts (e.g., code search). Further natural language processing techniques will be investigated to enhance the assessment of semantic similarities between comments and assertions. Using a learning-based method will also improve the precise positioning of comment scopes.

The use of code comments is essential to understanding the program and the software's maintenance. Automated code comment generation, consistency in code comment creation, classification in code comment creation, and code quality assessment comments are all covered in this article. Software system development and maintenance benefit greatly from the inclusion of code comments. It is incredibly beneficial for programmers to read and understand the code. The academic community has been studying code for a long time, yet numerous challenges and responsibilities remain. Future scholars may use this paper as a guide. To aid code inspectors in selecting suitable tools and procedures, the study's results may point to residual open-line research difficulties and potential future projects, among other things.

This study was the first to develop a method for examining and evaluating code comments thoroughly. It is the cornerstone of a model of comment quality to use machine learning to categorize comments. Quality aspects like coherence, consistency, completeness, and usefulness are discussed in detail. We came up with two approaches to gauge the model's quality: the coherence coefficient and the length indicator.

A poll of developers who had worked on comparable projects was then conducted. According to the case study results, our approach can give a much more in-depth analysis than previous approaches: Just counting how many individuals have commented on a system

document does not provide as much quantitative information as categorizing the comments into categories. This metrics collection reveals difficulties with code commenting that are frequent in practice for a qualitative examination. Reorganization ideas may also be made using metrics. You can see this by looking at the length indicator when there are inline comments with more than two words. The c coeff metric looks for both comments that don't provide enough data and methods with incorrect method IDs. It will be used in the future to evaluate the quality of comments. We still need to determine exactly how many additional areas of comment quality could be analyzed entirely or semi-automatically and just how many must be done manually.

REFERENCES

Allamanis, M. B. (2015). Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint meeting on Foundations of Software Engineering*, Bergamo, Italy. https://doi.org/10.1145/2786805.2786849

Antoniol, G. C. (2000). Tracing object-oriented code into functional requirements. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, Limerick, Ireland.

Ayala, C. T. (2021). Use and misuse of the term experiment in mining software repositories research.*IEEE Transactions on Software Engineering* (Ahead of print). https://doi.org/10.1109/TSE.2021.3113558

Chatterjee, P. G. (2017). Extracting code segments and their descriptions from research articles. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, Buenos Aires, Argentina. https://doi.org/10.1109/MSR.2017.10

Corazza, A. D. (2011). Investigating the use of lexical information for software system clustering. In *15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany. https://doi.org/10.1109/CSMR.2011.8

de Freitas Farias, M. A. (2020). Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology, 121*, 106270. https://doi.org/10.1016/j.infsof.2020.106270

Fluri, B. W. (2007). Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, Vancouver, BC. IEEE.https://doi.org/10.1109/WCRE.2007.21

Fluri, B. W. (2009). Analyzing the co-evolution of comments and source code. *Software Quality Journal, 17*(4), 367-394. https://doi.org/10.1007/s11219-009-9075-x

Gašević, D. K. (2009). Ontologies and software engineering. In *Handbook on Ontologies*. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-92673-3_27

Haav, H. M. (2001). In *Proceedings of the 5th East-European Conference ADBIS*, Vilnius, Lithuania.

Haiduc, S. A. (2010). On the use of automated text summarization techniques for summarizing source code. In *17th Working Conference on Reverse Engineering*, Beverly, MA. https://doi.org/10.1109/WCRE.2010.13

Howard, M. J. S. (2013). Automatically mining software-based, semantically-similar words from comment-code mappings. In *10th working conference on mining software repositories (MSR)*, San Francisco, CA. https://doi.org/10.1109/MSR.2013.6624052

Hu, X. L. (2018). Deep code comment generation. In *IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. Gothenburg, Sweden. https://doi.org/10.1145/3196321.3196334

Hu, X. L. (2020). Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering, 25*(3), 2179-2217. https://doi.org/10.1007/s10664-019-09730-9

Jiang, Z. M. (2006). Examining the evolution of code comments in Postgre SQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, Trier, Germany. https://doi.org/10.1145/1137983.1138030

Kagdi, H. C. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice, 19*(2), 77-131. https://doi.org/10.1002/smr.344

Khamis, N. W. (2010). Automatic quality assessment of source code comments: The JavadocMiner. In *International Conference on Application of Natural Language to Information Systems*, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-13881-2_7

Kuhn, A. D. (2007). Semantic clustering: Identifying topics in source code. *Information and software technology, 49*(3), 230-243. https://doi.org/10.1016/j.infsof.2006.10.017

Lalband, N. &. (2019). Software engineering for smart healthcare applications. *International Journal of Innovative Technology and Exploring Engineering, 8*(6S4), 325-331. https://doi.org/10.35940/ijitee.F1066.0486S419

LeClair, A. H. (2020). Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, Seoul, Republic of Korea. https://doi.org/10.1145/3387904.3389268

Lemos, O. A. (2020). Comparing identifiers and comments in engineered and non-engineered code: A large-scale empirical study. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, New York, NY. https://doi.org/10.1145/3341105.3373972

Liang, Y. &. (2018). Automatic generation of text descriptive comments for code blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Washington, DC. https://doi.org/10.1609/aaai.v32i1.12233

Liu, Z. X. (2019). Automatic generation of pull request descriptions. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA. https://doi.org/10.1109/ASE.2019.00026

Madani, N. G. (2010). Recognizing words from source code identifiers using speech recognition techniques. In *14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain. https://doi.org/10.1109/CSMR.2010.31

McBurney, P. W., & McMillan, C. (2014). Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, New York, NY.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. Retrieved from https://bit.ly/3CVn8de.

Moreno, L. A. S. (2013). Automatic generation of natural language summaries for java classes. In *21st International Conference on Program Comprehension (ICPC)*, San Francisco, California. https://doi.org/10.1109/ICPC.2013.6613830

Movshovitz-Attias, D., & Cohen, W. (2013, August). Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria.

Nolan, D. &. (2014). XPath, XPointer, and XInclude. In *XML and Web Technologies for Data Sciences with RSpringer*, New York, NY. https://doi.org/10.1007/978-1-4614-7900-0_4

Padioleau, Y. T. (2009). Listening to programmers-taxonomies and characteristics of comments in operating system code. In *IEEE 31st International Conference on Software Engineering*, Washington, DC. https://doi.org/10.1109/ICSE.2009.5070533

Palomba, F. Z. (2018). Automatic test smell detection using information retrieval techniques. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Limassol, Cyprus. https://doi.org/10.1109/ICSME.2018.00040

Panichella, S. A. (2012). Mining source code descriptions from developer communications. In *20th IEEE International Conference on Program Comprehension (ICPC)*, Passau, Germany. https://doi.org/10.1109/ICPC.2012.6240510

Paul, S. &. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering, 20*(6), 463-475. https://doi.org/10.1109/32.295894

Pressman, R. S. (2005). *Software engineering: A practitioner's approach*. London, UK: Palgrave macmillan.

Punyamurthula, S. (2015). *Dynamic model generation and semantic search for open source projects using big data analytics* (Master thesis). University of Missouri, Kansas City, Missouri.

Qin Ming, C. M. (2018). Image Semantic Comment Based on Classification Fusion and Association Rules Mining. *Computer Engineering and Science*.

Rodeghero, P. M. (2014). Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India. https://doi.org/10.1145/2568225.2568247

Roy, C. K. (2007). A survey on software clone detection research. *Queen's School of Computing TR, 541*(115), 64-68.

Scalabrino, S. B.V. (2019). Automatically assessing code understandability. *IEEE Transactions on Software Engineering, 47*(3), 595-613. https://doi.org/10.1109/TSE.2019.2901468

Song, X. S. (2019). A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access, 7*, , 111411-111428. https://doi.org/10.1109/ACCESS.2019.2931579

Sridhara, G. H. S. (2010). Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Montpellier, France. https://doi.org/10.1145/1858996.1859006

Srinivasan I. K. (2016). Summarizing source code using a neural attention model. In *ACL*, Berlin, Germany.

Svyatkovskiy, A. D. (2020). Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece. https://doi.org/10.1145/3368089.3417058

Syriani, E. L. (2018). Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures, 52*, 43-62. https://doi.org/10.1016/j.cl.2017.11.003

Tan, L. (2009). *Leveraging code comments to improve software reliability* (PhD thesis). University of Illinois at Urbana-Champaign, Champaign, IL.

Tan, L. (2015). Code comment analysis for improving software quality. In *The art and science of analyzing software data*. Burlington, MA: Morgan Kaufmann.

Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007, October). /* iComment: Bugs or bad comments?*. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY. https://doi.org/10.1145/1323293.1294276

Tan, L., Yuan, D., & Zhou, Y. (2007, May). Hotcomments: How to make program comments more useful? In *HotOS*, San Daiego, CA.

Van Roy, P. (2004). *Concepts, techniques, and models of computer programming*. Cambridge, MA: MIT press.

Vermeulen, A. A. (2000). *The elements of Java (tm) style.* Cambridge University Press, Cambridge, MA. https://doi.org/10.1017/CBO9780511585852

Wen, F. N. (2019). A large-scale empirical study on code-comment inconsistencies. In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, Montreal, QC. https://doi.org/10.1109/ICPC.2019.00019

Williams, C. C. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering, 31*(6), 466-480. https://doi.org/10.1109/TSE.2005.63

Wong, E. Y. (2013). Autocomment: Mining question and answer sites for automatic comment generation. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Silicon Valley, CA. https://doi.org/10.1109/ASE.2013.6693113

Xu, S. Y. (2019). Commit message generation for source code changes. In *International Joint Conferences on Artificial Intelligence*, Freiburg, Germany. https://doi.org/10.24963/ijcai.2019/552

Yang, B. L. (2019). A survey on research of code comment. In *Proceedings of the 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, New York, NY. https://doi.org/10.1145/3312662.3312710

Yihong, L. (2018). An Image Feature Extraction and Semantic Annotation Method Based on Visual Memory[J]. *Computer Knowledge and Technology, 2018*(15).

Ying, A. T. (2005). Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. *ACM SIGSOFT Software Engineering Notes, 30*(4), 1-5. https://doi.org/10.1145/1082983.1083152

Yu Hai, L. B. X. (2016). Source code annotation quality assessment method based on combined classification algorithm. *Journal of Computer Applications, 36*(12), 3448-3453.