# CloCom: Mining Existing Source Code for Automatic Comment Generation

Edmund Wong, Taiyue Liu, and Lin Tan
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada
{e32wong, t67liu, lintan}@uwaterloo.ca

***Abstract*—Code comments are an integral part of software development. They improve program comprehension and software maintainability. The lack of code comments is a common problem in the software industry. Therefore, it is beneficial to generate code comments automatically. In this paper, we propose a general approach to generate code comments automatically by analyzing existing software repositories. We apply code clone detection techniques to discover similar code segments and use the comments from some code segments to describe the other similar code segments. We leverage natural language processing techniques to select relevant comment sentences.**

**In our evaluation, we analyze 42 million lines of code from 1,005 open source projects from GitHub, and use them to generate 359 code comments for 21 Java projects. We manually evaluate the generated code comments and find that only 23.7% of the generated code comments are good. We report to the developers the good code comments, whose code segments do not have an existing code comment. Amongst the reported code comments, seven have been confirmed by the developers as good and committable to the software repository while the rest await for developers' confirmation. Although our approach can generate good and committable comments, we still have to improve the yield and accuracy of the proposed approach before it can be used in practice with full automation.**

***Keywords*—comment generation; documentation; program comprehension**

## I. Introduction

Code commenting is an integral part of software development. Developers rely on code comments to help understand source code. Previous work had shown that code comments can help improve software maintainability [1]. However, many code bases do not contain adequate code comments [2]. Therefore, it is beneficial to generate code comments automatically since it can save developers' time in writing comments.

Previous work from Sridhara et al. [3], [4], [5], [6] had focused on generating code comments for Java methods [3], groups of statements [4], Java classes [5], and parameter comments [6]. Their technique synthesizes natural language sentences from the code elements directly. However, these techniques rely on high-quality identifier names and method signatures from the source code. For example, the grouping of multiple statements together requires all method names contain the same verb [4]. The technique may fail to generate accurate comments if the source code contains poorly named identifiers or method names.

Our previous work AutoComment [7] mined a large-scale question and answer (Q&A) website, Stack Overflow [8], to extract and improve human written descriptions for automatic comment generation. AutoComment extracted code segments from Stack Overflow, and identified their corresponding descriptions. AutoComment then detected code segments in software repositories that are similar to the code segments on Stack Overflow, and used the improved descriptions (modified with natural language processing techniques) as comments for the code segments in the software repositories. Although human written sentences from Stack Overflow can be used as source code comments, the technique can only generate a limited number of comments automatically. The reason is if a code segment had never been discussed on a Q&A website, then AutoComment cannot generate a comment for the detected code segments that are similar, which limits the yield. Based on our user study [7], we learnt that comments that are written for easy-to-understand code (no comment is needed to help comprehension) are less useful.

To overcome the yield issue, we propose a general approach to mine new sources—existing open source software repositories—for code comment generation. This approach is similar to AutoComment. The main difference is that this approach identifies similar code segments between two code repositories, while AutoComment detects similar code segments between a code repository and code segments in Stack Overflow. This is based on the idea that 1) software reuse is common [9], [10], and 2) millions of lines of open source projects that contain code comments are available for the generation of human written comments. For example, the 1,005 Java open source projects that we downloaded from GitHub contains 42 million lines of code and 17 million lines of comments based on CLOC [11].

Mining human written comments from software repositories have four main challenges over our previous work [7], which mines human written comments from Q&A sites:

First, since we mine code comments from a large pool of software repositories, a code segment is often similar to many other code segments that contain code comments because software reuse is common. This is a problem because a single code segment can have many comment candidates. In our previous work [7], Q&A sites often disallow duplicate questions, we typically have less than four comment candidates when mining Q&A sites. Therefore, we need new approaches to rank the multiple matched comment candidates.

Second, it is more challenging to process source code comments compared to the human written sentences from Q&A sites. The reason is that code comments are often not written

in full sentences, which means a natural language parser cannot process the code comments accurately. In contrast, since Q&A sites are a platform for users to ask questions and obtain feedback from other users, they encourage users to use full English sentences.

Third, code comments mined from source code are more likely to contain project specific information compared to human written sentences on Q&A sites. The reason is source code comments are written to be read by developers who are interested in understanding the logic of the commented code with respective to the surrounding code, whereas human written sentences on Q&A sites are written to be read by the general public. When we mine code comments from source code, we require an effective approach to determine if a code comment contains keywords that are only applicable to a specific code segment.

Forth, mining code comments from existing software repositories requires a more advanced code clone detection technique due to two reasons. First, the software repository is larger consisting of 42 million lines of code, compared to the code segments in Q&A sites, which have an average size between three to ten lines of code. The large repository brings the need for a highly scalable code clone detection technique. Secondly, many code segments in Q&A are uncompilable, which limits us from extracting fine-grained information such as type information and variable scope level. Since we extract code segments from complete source code files in the target project, we can leverage an abstract syntax tree parser to extract fine-grained information (i.e., variable type bindings and type of a statement) from source code to help improve the accuracy of the code clone detection technique.

To address these challenges, we proposed four techniques, including 1) filtering of code clones that do not have semantic similarity, 2) extraction of code comments from the source code, 3) elimination of code comments that contain project-specific information, and 4) ranking multiple code comment candidates. This paper makes the following contributions:

- We proposed a new approach, CloCom, to analyze software repositories for comment generation. The approach performs code clone detection between the input target project and the software repositories, and leverages information from all the similar code segments to eliminate code comments that contain project specific information.
- We analyzed 42 million lines of code from 1,005 Java projects and generated 359 comments. Our result shows that 23.7% of the generated code comments are good. The yield and accuracy are both low despite having a larger set of source code for mining compared to our previous work [7]. However, we analyzed the key observations in the results and reported the good code comments to developers. Amongst the reported code comments, seven have been confirmed by the developers as good and committable to the software repository while the rest await for developers' confirmation.
- We made our tool open source and our classification results publicly available on our project website (see Section Availability). Our tool is capable of extracting mappings between source code and code comments, and can act as a standalone code clone detection tool.

## II. Approach

Figure 1 shows the design of CloCom. CloCom takes two inputs: 1) software projects for comment extraction, e.g., open source projects from GitHub, and 2) target software projects to be commented. The output of CloCom is a list of automatically generated comments for the target projects.

CloCom generates code comments for the target projects. It detects code clones between the database containing raw software projects and the target projects' source code (Section II-A), prunes out code clones that do not have semantic similarity (Section II-B), maps each code segment with its respective code comment (Section II-C), prunes out code comments that contains invalid information (Section II-D), and selects the code comment(s) that best describe the code segment from the list of available candidates (Section II-E).

We compared CloCom against our previous work, AutoComment [7]. The following techniques are new and improved to address the challenges described in Section I. First, we proposed a new context-sensitive text similarity measure for selecting valid code comments (Section II-D). It performs a three-way analysis between the code comment, the code segment from the database, and the code segment from the target project. Second, we proposed a new approach to rank the generated code comments based on the code comments' text similarity score and length (Section II-E).

### A. Code Clone Detection

In order to generate code comments for the target projects, we first have to locate code clones. We discover similar code segments between the database and the target projects.

*1) Algorithm:* We designed and implemented a code clone detection tool, which is token-based. Token-based code clone detection tools can achieve a time complexity of $O(n * m)$ with token-based matching, where $n$ is the number of lines of code in the input project; and $m$ is the number of lines of code in the database. The token-based matching algorithms are usually more efficient than tree-based matching algorithms (polynomial runtime [12]). The time efficiency is important for scalability purposes because our database contains 42 million LOC.

Our matching algorithm is similar to that of DuDe [13] with two differences. First, our technique is language dependent due to the usage of an abstract syntax tree (AST) parser for accurate tokenization of the code elements. The parser is different from Dude [13], which detects code elements using regular expressions. Second, we only support Type-1 and Type-2 clones whereas DuDe supports Type-1, Type-2, and some Type-3 clones. Type-2 clone is defined as "Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments" [12]; and Type-3 clone includes changed, added, removed statements [12]. We would like to explore Type-3 clone in the future because changed, added, or removed statements will impact the meaning of the code segment.

We did not leverage existing code clone detection tools because many are not scalable [14]. Existing mature code clone detection tools such as NiCad [15] and Deckard [14] both have a quadratic runtime [12], whereas our approach has

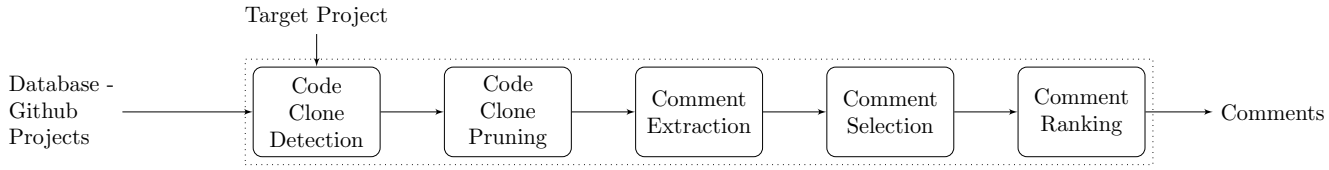


Fig. 1. Overview of CloCom

a linear runtime of $O(n * m)$. In addition, we require fine-grained tuning of the tokenization rules (i.e., require exact matching on strings and chars) and the extraction of metadata (i.e., scope level of a statement, identifier names, and the type of a statement). This brings the requirement of an open-source software.

*2) Tokenization:* Tokenization of the source code is based on the serialization of AST nodes. We utilized Eclipse AST-Parser [16] to obtain the AST of the Java source code. ASTParser cannot fully resolve variable type bindings of code elements unless we manually compile each project in the database, which is infeasible because there are 1,005 Java projects in GitHub. We overcome this problem by tracking all the variable declarations within each Java class, which gives us a non-fully qualified data type of each variable. This approach produces fewer false clones compared to SIM [17], a code clone detection tool used in our previous work [7], where variables with different declaration types are treated as the same token.

*3) Configuration:* Our tool recognizes code clones with three or more statements, which is a common threshold similar to other previous work [18]. Our tool excludes two types of statements including *return statement* and *switch statement*. Return statement is less meaningful, and switch statement can have many overlapped segments that introduce false positives [19]. In our evaluation, we observe that 59.7% of the clones have a size of three statements, 21.1% have a size of four statements, 5.7% have a size of five statements, and 13.5% have a size of six or more statements.

We require string and char literals to match exactly during the tokenization phase. The reason is that the value of string and char literals have a high impact on the semantic meaning of the code segment. The following shows a code segment, where its semantic meaning is highly dependent on the value of the string literal:

```
1 // if element is not "property" then skip
2 if ( !"property".equals( propElement.getTagName() )
      ) {
3   continue;
4 }
5 String propName = propElement.getAttribute( "name"
      ).trim();
6 String propValue = propElement.getAttribute( "value
      " ).trim();
```

## B. Code Clone Pruning

Code clone detection tools, in general, cannot distinguish the difference between a meaningful match and a meaningless match. Many tools simply report a match if two code segments are syntactically similar. For example, the following code segment contains many syntactically similar code clones, but the code clones will not be useful from a semantic standpoint.

```
1 int i = 0;
2 int j = 0;
3 int k = 0;
```

In order to identify semantically similar matches, we filter out clones that are syntactically similar, but not semantically similar using the following basic heuristics, which had been deployed in our previous work [7].

*1) Require at least one Method Invocation:* If a code clone does not contain at least one method invocation, it means that the code segment is performing low-level operations (e.g., simple variable declarations, switch statements, for/while/try/catch statement headers). Hence, we discard such code segments.

*2) No Repetitive Statements Allowed:* Code clones, where the matched body statements only consist of the same repeated statement, are removed. This type of code clone is particularly common in code that are related to declarations and numerical computations. The following is an example of a repetitive code segment:

```
1 hasmap.put("John", 0);
2 hasmap.put("Peter", 0);
3 hasmap.put("Mary", 0);
```

## C. Comment Extraction

Once we have a list of useful code clones, we retrieve the code comments from the AST of the database code segment. We map the code comments to the code segments from the target project. We preprocess all the extracted code comments by normalizing spaces and removing newline characters. The preprocessing helps us detect if the same code comment already exists in the code segment of the target project. In such cases, it means the code segment from the target project does not require a code comment, and hence we do not extract a duplicate code comment. Code comments can exist in one of the following three forms.

*1) Single-line Comment:* Single-line comment starts with a double slash (//). In our previous work [7], we were able to extract full sentences easily from Stack Overflow posts, which is not the case in this work with single-line comments. The problem with single-line comments is that programmers often stack multiple lines of single-line comments together to form a single comment block. It is common for programmers to omit the full stop symbol, and we observe that code comments are often not written as a full sentence. The following shows an example of the issue:

```
1 // obtain a list of files
2 // inside the directory
3 // remove the files afterwards
```

In the case where there are multiple single-line comments, we simply merge and treat them as a single comment. We merge them due to two reasons. First, the sentences that are within the same block can be describing the same subject. It will be inappropriate to remove a sentence from a block that contains multiple sentences without understanding the sentence's context. Second, there is no reliable way to break down a block into individual sentences.

A different form of single-line comment is the in-line comment, which contains code in front of the comment. It is only used to describe a single line of code as opposed to a block of code, which is often less useful, and hence we discard them.

*2) JavaDoc Comment:* JavaDoc comment is delimited by `/**...*/`. It only appears in front of a public class, or a public/protected method/variable.

*3) Block Comment:* Block comment is delimited by `/*...*/`. It is similar to the JavaDoc comment except it can appear anywhere in the source code.

### D. Comment Selection

A code segment from an input project can be matched against code segments in the database, referred to as *database clones*. However, a code segment from an input project can also be matched against code segments within its project, referred to as *local clones*. Since all these matches share the same code structure, we treat them as a single *match group* for analysis, as opposed to our previous work [7] which had analyzed them separately. The goal of comment selection is to aggregate and leverage the information between all these code clone matches, and generate a single comment that is applicable against all the code clones. Having a large number of code clones within a clone group is beneficial to this technique because more information will be available for selecting code comments.

Text similarity is a measure to calculate the association of a code comment against 1) all code segments from the target projects, and 2) all code segment from the database.

Text similarity is a global metric because it compares the comment against all the possible code clones, which allows us to prune out project specific comments. It is a three-way analysis between 1) the code comment, 2) the code segment from the database, and 3) the code segment from the target projects. The analysis differs from our previous work [7], which only performs a two-way analysis between 1) the code comment, and 2) the code segment from the target projects. The key difference is that, in this work, we want to focus on determining if the code comment contains text similarity terms that are specific to a code segment (a context-sensitive analysis). Our previous work only checks if there are text similarity terms (a context-insensitive analysis).

Consider the following database code segment that reads a text file line by line:

```
1 // read the text file
2 String textFile = "file.txt";
3 FileReader fr = new FileReader(textFile);
4 BufferedReader br = new BufferedReader(br);
5 StringBuilder sb = new StringBuilder();
6 String line = br.readLine();
7 while (line != null) {
8     sb.append(line);
```

The database code segment contains a code comment in line 1 that describes the matched code between line 2-8, and the code comment assumes that the variable, `textFile`, in line 2 is a text file. However, given a different code segment that is in a *different context* (i.e., an audio file instead of a text file), this code comment will not be applicable. Hence, if we can detect the term, `text`, is a context-sensitive term, we can discard this code clone pair.

In some cases, text similarity terms between a comment and a code segment can exist outside of the matched code statements. For example, if the variable, `textFile`, is named `file`, then there will be no way to identify that the term, `text`, is important. To tackle this problem, we extend the search to the entire method body that encapsulates the code segment during the similarity term extraction process. Similarity terms that exist within the matched statements are considered to be in the *local context*, and terms that exist outside the matched statements are considered to be in the *global context*. However, we require there to be at least one similarity term that exists in the local context, otherwise we discard the match. This is because a code comment should be directly describing the matched statements.

Similarity terms are extracted from two different types of sources, including 1) code comments, and 2) source code. For the source code, we extract all the simple name nodes from the AST. A simple name is defined as an identifier other than a keyword, boolean literal, or null literal. Each term is then broken down based on the camel case convention (i.e., `CamelCase` can be divided into two terms, `Camel` and `Case`). The similarity score between two lists of similarity terms is the number of overlapping terms. A higher similarity score gives us a metric to measure the closeness of a code comment against the matched statements.

We use the following simplified example to illustrate this technique. First, we have a code segment from the input project in Figure 2.

```
1 Pattern pattern = Pattern.compile("\\bhello\\b");
2 Matcher matcher = pattern.matcher(argument);
3 if (matcher.find()){
4   return true;
5 }
```

Fig. 2. A code segment from the input project that does not have a comment

The code segment is matched against another piece of code segment from the database in Figure 3, which has a code comment at line 1.

```
1 // Search for the term hello from the input string
2 Pattern pattern = Pattern.compile("\\bhello\\b");
3 Matcher matcher = pattern.matcher(inputString);
4 if (matcher.find()){
5   return true;
6 }
```

Fig. 3. A code segment from the database that contains a comment

The procedure for calculating the text similarity is the following:

(i) Extract a list of terms from the following sources:
- input project code segment (#1):

`pattern, compile, hello, matcher, argument, find`
- database code segment (#2):
`pattern, compile, hello, matcher, input, string, find`
- database code comment (#3):
`search, for, the, term, hello, from, the, input, string`

(ii) Obtain the list of shared terms between #2 and #3:
`hello, input, string`
This returns a list of terms that are important in the code comment.

(iii) We require all the shared terms between (#2 and #3) to exist in #1. Since the intersection between (#2 and #3) and #1 is missing two text similarity terms, `input` and `string`, we discard this code comment. However, if the two missing text similarity terms exist in the method that encapsulates the input project code segment, then we can consider this as a match. The reason is that 1) all the shared terms between (#2 and #3) exists in #1, and 2) at least one of the terms, `hello`, exists in the local context.

In our design, we require at least one text similarity term to exist in the local context. We show the frequency distribution of the text similarity terms for both the local and global context in Table I. It is possible to configure the matching to be more strict by increasing the text similarity score threshold. However, we observed that 80.6% of the sentences contain one local text similarity term only, which means a text similarity score of one is a good threshold. The same applies to the global text similarity score, where we did not configure a threshold because 93.5% of the code comments do not contain a global text similarity term.

TABLE I. TEXT SIMILARITY SCORE DISTRIBUTION OF ALL THE GENERATED CODE COMMENTS

| Score | Local Frequency | Global Frequency |
|---|---|---|
| 0 | not applicable | 93.5% (638) |
| 1 | 80.6% (550) | 4.8% (33) |
| 2 | 9.4% (64) | 1.3% (9) |
| 3 | 4.4% (30) | 0.1% (1) |
| $\geq 4$ | 5.6% (38) | 0.1% (1) |

Lastly, code comments commonly contain code artifacts that represent the important elements within a sentence. We require code artifacts, which exist inside a code comment, exist inside all the matched code segments. For example, the code comment, "Invoke arraycopy() to copy the numbers to the new array," contains the method name, `arraycopy`. If `arraycopy` does not appear in the code segment, it is very likely that the code comment is not applicable. We utilize the following regular expressions, similar to our previous work [7], to detect code artifacts:

- Quoted text surrounded by a single or double quote.
- Method invocations and field access.
- Camel cases including standard `CamelCase`, interior `camelCase`, and capital `CAMELCASE`.
- Remove code comments that contain any of the commonly used terms in defect prediction [20].
`bug, fix, error, issue, crash, problem, fail, defect, patch`

TABLE II. 21 EVALUATED OPEN SOURCE JAVA PROJECTS
LOC - LINES OF CODE
ET - EXECUTION TIME IN MINUTES

| Project | LOC | ET | Project | LOC | ET |
|---|---|---|---|---|---|
| Java JDK | 964,143 | 141 | ArgoUML | 195,363 | 26 |
| DNSJava | 63,071 | 3 | Ant | 135,407 | 25 |
| ANTLR | 42,078 | 6 | Carol | 11,694 | 4 |
| GanttProject | 54,461 | 4 | Hibernate | 528,662 | 46 |
| HSQLDB | 169,178 | 19 | JabRef | 96,663 | 18 |
| Jajuk | 68,998 | 15 | JavaHMO | 25,631 | 6 |
| JBidWatcher | 30,219 | 5 | JFtp | 77,195 | 5 |
| JHotDraw | 233,991 | 5 | MegaMek | 289,864 | 52 |
| Planeta | 11,125 | 3 | SweetHome | 87,547 | 16 |
| Vuze | 574,566 | 66 | FreeMind | 67,287 | 8 |
| FreeCol | 130,308 | 22 | | | |

### E. *Comment Ranking*

Since each code segment can have a large number of code comment candidates, we rank our results and only present the best code comments to the user.

We rank the code comments based on two simple heuristics, including 1) the closeness of a code comment against the code segment (text similarity score), and 2) the conciseness of the code comment (number of words in the code comment). The calculation of the text similarity score is shown in Section II-D. A high text similarity score means the code comment is closely associated against the code segment. However, since code comments can obtain the same text similarity score, we further rank the code comments based on the conciseness of the code comments by counting the number of words in the code comment.

In this paper, we display a maximum number of two code comments for each code segment. The reason is that 95.4% of the code clone groups have a maximum number of two code comments. If multiple comments contain the same text similarity score and have the same length, we display them all to the user, even if there are more than two code comments.

## III. EXPERIMENTAL METHODS

We evaluate CloCom on open-source software projects.

### A. *Data Source*

We apply CloCom to generate comments for 21 Java open-source projects in our evaluation. The evaluated projects includes 6 commonly evaluated projects in code clone detection [21], i.e., Java JDK, ArgoUML, DNSJava, Ant, ANTLR, and Carol, along with 15 projects that were previously evaluated for automatic comment synthesis by Sridhara et al [4]. The total number lines of code for each project calculated using CLOC [11] is shown in Table II, along with the single-threaded execution time on an Intel Core i5-3470 CPU with a 3GB Java heap.

Since we require a software repository for mining code comments, we collected open source Java projects from a repository host, GitHub, to generate the database. Our script queries the repository hosts' search API and fetches all the hosted software. We only collected Java projects because our code clone detection tool (Section II-A) is currently limited to the Java programming language. As of November 2014, our database contains 1,005 projects from GitHub for a total of 42 million LOC.

### B. Evaluation Criteria

We performed a manual verification to evaluate the quality of the automatically generated code comments. Our previous work [7] evaluated the quality of each comment based on the accuracy, adequacy, conciseness and usefulness criteria. Recent work by Moreno et al. [5] evaluated the comments based on the content adequacy, conciseness, and expressiveness criteria. These approaches have a common problem. There are multiple criteria for each code comment, which can be too fine-grained for determining if a comment will be committed and used by developers. For example, a code comment with a high accuracy score and a low conciseness score does not tell us whether a developer would consider it committable to the code repository. Therefore, we propose the following ranking criteria.

Under the assumption that the code comment has to be committed in the software repository, the code comment is:

- **Good**: The generated comment is accurate, adequate, concise and useful at describing the source code, i.e., it can be committed. If there is an existing comment, the generated comment must have a major difference in sentence structure, or offer a similar or better quality compared to the existing comment.
- **Fix**: The generated comment is not accurate, adequate, concise, or useful, at describing the source code but can be fixed with minor modifications, i.e., it can be committed with a fix.
- **Bad**: The generated comment is not accurate, adequate, concise, or useful at describing the source code, i.e., it cannot be committed.

In the case where a comment is not accurate, adequate, concise, or useful, we classify the main types of problem for further analysis.

### C. Research Questions

We would like to answer the following research questions in our evaluation:

**RQ1:** What are the yield and quality of the automatically generated code comments?
**RQ2:** For a comment that contains partially incorrect information, what has to be done (i.e., types of modifications that are needed) in order to fix the comment?
**RQ3:** What are the reasons for a code comment to be inapplicable to a different code segment (despite the two comments having the same syntax)?
**RQ4:** How does the quality of the existing code comments compare to the quality of the automatically generated code comments?

## IV. Results

### A. RQ1

We apply our tool to 21 projects to evaluate the yield and quality of the generated code comments. Table III shows the detailed breakdown.

TABLE III. YIELD AND QUALITY RESULTS OF THE COMMENTS
NC - NUMBER OF GENERATED CODE COMMENTS
CG - NUMBER OF UNIQUE CODE SEGMENTS (CLONE GROUPS)
LC - AVERAGE NUMBER OF CLONES IN THE INPUT PROJECT
DC - AVERAGE NUMBER OF CLONES IN THE DATABASE
GD (GOOD) - CODE COMMENTS IN NC IS ACCURATE, ADEQUATE, CONCISE AND USEFUL
FX (FIX) - CODE COMMENTS IN NC IS NOT ACCURATE, ADEQUATE, CONCISE, OR USEFUL BUT CAN BE FIXED WITH MINOR MODIFICATIONS
BD (BAD) - CODE COMMENTS IN NC IS NOT ACCURATE, ADEQUATE, CONCISE, OR USEFUL
CM - CODE SEGMENT IN LC HAS AN EXISTING CODE COMMENT
* - AVERAGE VALUE

| | Yield | | | | Quality | | | |
|---|---|---|---|---|---|---|---|---|
| Project | NC | CG | LC | DC | GD | FX | BD | CM |
| Java JDK | 96 | 84 | 2.1 | 1.8 | 23 | 5 | 68 | 51 |
| ArgoUML | 22 | 18 | 1.7 | 1.4 | 9 | 1 | 12 | 9 |
| DNSJava | 12 | 10 | 1.4 | 1.7 | 2 | 4 | 6 | 7 |
| Ant | 34 | 27 | 1.9 | 2.1 | 6 | 2 | 26 | 14 |
| ANTLR | 11 | 8 | 1.1 | 1 | 2 | 2 | 7 | 11 |
| Carol | 3 | 3 | 1 | 1 | 0 | 0 | 3 | 1 |
| GanttProject | 4 | 2 | 1 | 9 | 0 | 0 | 4 | 4 |
| Hibernate | 12 | 9 | 1.4 | 3.1 | 3 | 2 | 7 | 4 |
| HSQLDB | 35 | 27 | 1.5 | 1.9 | 8 | 3 | 24 | 15 |
| JabRef | 20 | 18 | 1.2 | 2.6 | 8 | 0 | 12 | 3 |
| Jajuk | 8 | 5 | 1.8 | 4.6 | 3 | 0 | 5 | 7 |
| JavaHMO | 5 | 3 | 3 | 7 | 0 | 1 | 4 | 3 |
| JBidWatcher | 16 | 11 | 1 | 3.4 | 1 | 1 | 14 | 7 |
| JFtp | 2 | 2 | 4 | 1 | 0 | 0 | 2 | 0 |
| JHotDraw | 7 | 6 | 1 | 1.2 | 6 | 0 | 1 | 7 |
| MegaMek | 13 | 8 | 2.5 | 3.9 | 1 | 0 | 12 | 0 |
| Planeta | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| SweetHome | 15 | 9 | 2.1 | 3.7 | 4 | 0 | 11 | 4 |
| Vuze | 19 | 15 | 2.3 | 2.3 | 1 | 0 | 18 | 6 |
| FreeMind | 9 | 6 | 1 | 1 | 3 | 0 | 6 | 6 |
| FreeCol | 15 | 9 | 1.2 | 3.4 | 5 | 0 | 10 | 1 |
| ALL | 359 | 281 | 1.8* | 2.3* | 85 | 21 | 253 | 161 |

*1) Yield of the generated code comments:* We generated a total number of 359 comments for 281 unique code segments. We show the number of generated code comments (NC) per project. In average, our tool generated an average of 17 comments for each project. The yield is still low considering that the 21 evaluated projects contain over 3.8 million LOC.

Since a unique code segment can exist in multiple locations within a single project, we only count it once and refer this as a *clone group* (CG). Each clone group consists of code segments from the following sources, including 1) the input project, and 2) the database. We show the average number of code clones within each clone group that comes from the input project, referred to as local clones (LC), and the software repositories, referred to as database clones (DC). In the case where the input project code clone already contains an existing code comment (CM), we compare the existing code comment against the automatically generated code comment in RQ4.

Each clone group contains an average of 1.8 local clones and 2.3 database clones. Do note that NC represents the total number of unique code comments for each clone group (CG). For example, if we generated a single code comment for a CG that contains two LC, we only count NC as one instead of two. We show the distribution of the clone group size in Table IV. A large LC or DC size is advantageous to our technique because it helps improve the accuracy of the text similarity and code artifact pruning. Based on the data, we see that 58.4% of the clone groups contain a size of two. In the future, we can increase the clone group size by expanding the database with more projects, or leverage more advanced code clone detection techniques.

TABLE IV. FREQUENCY DISTRIBUTION OF THE NUMBER OF CODE CLONES PER CLONE GROUP (CG)

| Size | Frequency |
|---|---|
| 2 | 58.4% (5596) |
| 3 | 15.0% (1432) |
| 4 | 6.4% (610) |
| 5 | 4.4% (426) |
| $\geq 6$ | 15.8% (1513) |

*2) Quality of the generated code comments:* We perform a manual evaluation on all the generated code comments for each project, and show the classification results in Table III. There are a total number of 359 generated comments in total. Amongst them, we observe that 85 (23.7%) of the code comments are good, 21 (5.8%) require modifications, and 253 (70.5%) are bad. It is interesting to observe that only a minor number of code comments require a fix to become applicable. We further analyze the cause for incorrect code comments in RQ2 and RQ3.

*CloCom generated code comments with a low yield and accuracy. It only generated 359 code comments for the 21 evaluated projects, and only 23.7% of all the generated code comments can be directly applied to the code segments. We report to the developers the good code comments, whose code segments do not have an existing code comment. Amongst the reported code comments, seven have been confirmed by the developers as good and committable to the software repository while the rest await for developers' confirmation.* The commit rate of the code comments are available on our website (see Section Availability).

## B. RQ2

RQ1 showed that 5.8% of the code comments require minor modifications prior from being applicable against a different code segment. We classified the major types of modifications that are required in Table V.

TABLE V. THE MAJOR TYPES OF MODIFICATIONS THAT HAVE TO BE APPLIED TO THE FIXABLE CODE COMMENTS

| Class | Cov. | Explanation |
|---|---|---|
| 1 | 80.9% (17) | The subject is wrong. For example, "Append the path", has the subject, "path", which is incorrect because the code can be appending a "string". |
| 2 | 14.3% (3) | The modifier of a head noun is incorrect. For example, "text file" has a modifier "text" and a head noun "file", which is incorrect because the code can be dealing with a "music file". |
| 3 | 4.8% (1) | The comment contains multiple clauses, one of which needs to be removed to be a valid comment for the code segment. For example, "Give the concurrent thread time so it will try to acquire locks" contains a clause, "so it will try to acquire locks". However, the code is not trying to acquire locks, which renders the entire sentence to be incorrect. |

Result shows that the majority of the problems come from comments that contain context-sensitive information (class 1 and class 2). For example, a piece of code segment that is responsible for reading a file can be reading different types of file (e.g., text file, music file). Such case requires advanced natural language analysis techniques to detect important terms within a sentence.

Another source of the problem is that generated code comments might contain multiple clauses (class 3), where one of them may contain irrelevant information. For example, "Retrieve the files and close the connection" contains two clauses connected by "and". One would have to detect and separate the analysis of the two clauses in order to be able to eliminate the second clause. However, it is very rare for this type of code comment to appear. A possible extension of the code comment analysis to a more fine-grained model (per clause analysis) will be beneficial.

In summary, *majority of the fixable code comments are due to incorrect context, where the action (verb) is correct but the subject (noun) is incorrect*. It may be possible to leverage natural language processing techniques to analyze the verb phrases and noun phrases in the sentence. However, since code comments are not necessarily written in full sentences, and that there are many technical terms in the source code, it may be challenging to obtain the parse tree of the sentence. We believe that the usage of a statistical parser, coupled with a list of common technical phrases in software engineering, can help obtain a correct parse tree of the sentence.

## C. RQ3

RQ1 showed that 70.5% of the code comments are bad comments. We classified the major types of reasons that make a code comment not applicable in Table VI.

TABLE VI. THE MAJOR TYPES OF REASONS FOR AN INAPPLICABLE CODE COMMENT

| Class | Cov. | Explanation |
|---|---|---|
| 1 | 2.0% (5) | Code comment contains invalid information. |
| 2 | 77.5% (196) | Code comment is not useful or too trivial. |
| 3 | 20.5% (52) | The target project code segment already has a similar comment (minor differences in grammar and spelling). |

Result show that the most common reason for an invalid code comment is due to not useful or trivial comments (class 2). These comments are not incorrect. For example, "create a factory class," "close previous connection," are trivial code comments that are correct, but not necessarily useful for a programmer. In order to filter such trivial comments, we have to apply techniques to analyze for code comments that are a simple rephrase of the code element. This is very interesting because the trivial code comment exists, which means some developers had considered the code comments to be useful. Since this is a subjective matter, we recommend readers to check out the evaluation details on our website.

The second most common reason for an invalid code comment is that there exists a similar code comment in the input project's source code (class 3). Although our tool already detects and filters away code comments that are similar, we notice that a lot of code comments contain grammatical, spelling, or one to two word modifications to the sentence. Hence, we were not able to filter them out automatically.

Another reason for an invalid code comment is that the code comment contains invalid information. However, the number of code comments that contain invalid information is very low, which suggests that 1) code comments from software repositories have a good quality, and 2) our text similarity technique is successful at selecting valid code comments.

*The majority of code comments that are incorrect contain information that is trivial or not useful. This is interesting because even though some developers wrote the code comments,*

*it does not necessarily mean the code comments are useful. They may be pressured to write comments for the purpose of writing comments.* In order to detect invalid information in code comments, we may be able to leverage natural language processing techniques such as the dependency parser to extract grammar relationships of the words in a sentence.

### D. RQ4

One question is whether the code segment from the input project contains an existing code comment. In our design, if the code segment from the input project contains a code comment that is identical to the generated code comment, we automatically discard the code comment (Section II-C). Among the remaining code segments from the input project, 44.8% (161/359 from Table III) of them contain an existing code comment that is different from the generated comments. We compare the automatically generated comments against the existing code comments (CM in Table III) in Table VII.

TABLE VII. AUTOMATICALLY GENERATED CODE COMMENTS VS. EXISTING CODE COMMENTS
BETTER - GENERATED COMMENT IS BETTER IN QUALITY
SIMILAR - THE TWO COMMENTS ARE SIMILAR IN QUALITY
WORSE - GENERATED COMMENT IS WORSE IN QUALITY

| Classification | Better | Similar | Worse |
|---|---|---|---|
| Good | 2.5% (4) | 16.8% (27) | 9.9% (16) |
| Fix | 0% (0) | 0% (0) | 9.3% (15) |
| Bad | 0% (0) | 32.9% (53) | 28.6% (46) |

Our results show that existing code comments are superior to automatically generated code comments, For the automatically generated code comments that are good, only 2.5% of them are better than the existing code comments, but 16.8% of them are similar compared to the existing code comment. *However, our result from RQ1 shows that 55.2% (1 - 44.8%) of the code segments contain no existing code comment that is different from the generated comment. This means although automatically generated code comments are generally not as good as existing code comments; it may still be beneficial to generate a code comment for the code segments that do not have an existing code comment.*

### E. Reporting Comments to Developers

We had reported to the developers all code comments that are rated as good, provided the code comments' corresponding code segments do not have an existing comment, which consists of 37 out of the 85 of the good comments. Developers were asked to rate the comments using the same evaluation criteria in Section III, which is based on whether a code comment can be committed. As of December 2014, two of the projects' developer replied and gave a rating of *GOOD* to all seven comments that they were asked to evaluate. Figure 4 shows an example of one of the reported comments (line 6), which the developer had rated it as good for commit:

## V. LIMITATIONS AND FUTURE WORK

Most of the matched code segments contain only 3–5 lines of statements (largest contains 39 lines of statements), and the code segments may contain simple code that does not require a code comment.

```
1  try {
2    if (new File(jarEntryURL.toURI()).canWrite()) {
3      connection.setUseCaches(false);
4    }
5  } catch (URISyntaxException ex) {
6    // Wrap the exception and re-throw
7    IOException ex2 = new IOException();
8    ex2.initCause(ex);
9    throw ex2;
10 }
```

Fig. 4. The developer had rated the code comment on line 6 as committable.

There are two possible explanations to the yield and quality issue. First, this can be a limitation of our code clone detection tool, which only detects Type-1 and Type-2 clones. Type-3 clones refer to clones that contain changed, added, or removed statements, which we did not consider in this work. The reason is that Type-3 clones have a higher chance of being not applicable against a cross-project code comment. One can try to leverage these advanced code clone detection techniques and tools to help detect more sophisticated types of clones.

Second, our technique is not eliminating simple code segments effectively. Our code clone detection technique accepts a code segment if it contains three or more statements. However, it is clear that in some cases, not all three of the statements are truly useful code. For example, the following code segment currently counts as three statements. One possible method to fix this is to not consider or count the "try" token on line 1 as a valid statement from the AST node.

```
1 try {
2   Thread.sleep(100);
3 } catch (InterruptedException inte) {
```

## VI. THREATS TO VALIDITY

In our evaluation, a main threat to validity is that we evaluated the generated code comments manually, and our evaluation criteria is different compared from those of the previous work [5], [7]. We tried our best to be objective in our evaluation, made the results publicly available, and reported the generated code comments to the developers. Our evaluation criteria is based on whether a code comment could be committed in the software repository, which we rank them using the *good*, *fix*, and *bad* criteria (Section III). These criteria are different compared to the previous work [5], [7], which rate comments based on the *accuracy*, *adequacy*, *conciseness*, and *usefulness* criteria. The reason for this change is that we do not believe that it is beneficial to break down the evaluation into multiple criteria. For example, a code comment that has a high accuracy score and a low conciseness score does not tell us the applicability of a code comment.

## VII. RELATED WORK

### A. Automatic Comment Generation

There is much work that focuses on automatic comment generation for specific code structures. Our previous work [7] took a unique approach towards automatic comment generation, which mines human written code comments from Q&A sites. Recent work from Sridhara et al. automatically synthesize natural language sentences for specific types of code segments directly, including Java methods [3], groups of

statements [4], Java classes [5], and parameter comments [6]. Others focus on generating documents for exception [22], failed test cases [23], and code changes [24]. A few previous work focus on mining descriptions or documentation from developer communications, such as bug reports, forum posts and emails [25], [26], [27].

### B. Code Clone Detection

In the area of code clone detection, there are token-based [19], [28], AST-based [14], [29], and semantics-based techniques [30]. Previous work [12], [31] performed comparisons between the existing code clone detection techniques. Roy et al. [12] performed a comprehensive comparison between a large number of code clone detection tools based on their attributes, and their capability at handling Type-1, Type-2, Type-3, and Type-4 clones. Svajlenko et al. [31] developed a framework for making code clone detection tools scalable by partitioning the dataset, and based their analysis on tradition tools including Deckard [14], NiCad [15], iClones [32], Simian [33], SimCad [34], and CCFinderX [28].

Our code clone detection implementation is similar to DuDe [13], which is a text-based tool. DuDe relies on combining non-gapped clones into a single large clone to support Type-3 clones. There are two main differences between CloCom and DuDe. First, DuDe applies regular expressions to perform tokenization of the source code, whereas we utilized an Eclipse ASTParser [16] for accurate tokenization. Second, Dude is capable of handling Type-3 clones, whereas we do not support it to ensure the high scalability of the tool.

## VIII. Conclusion

We proposed a new general approach to mine software repositories for automatic comment generation. Our tool generated 359 code comments for 281 unique code segments on the 21 evaluated software projects. Amongst all the generated code comments, 23.7% of the code comments can be directly committed into the software repository as a good comment. However, the yield and accuracy are still considered to be relatively low, which suggests the approach needs to be improved before it can be used in practice with full automation.

Compared to our previous work [7], our technique is capable of generating more code comments (181 in CloCom vs. 102 in AutoComment) on the same 15 projects [4]. We proposed a new context-sensitive text similarity technique to tackle the issue with code comments that contain information specific to the code segment, which is a challenge that did not exist in our previous work.

In our evaluation, we proposed a simplified metric to rank the evaluated code comments based on the assumption that the code segment has to be committed in the software repository. We answered four research questions, including 1) the yield and quality of the generated code comments, 2) the reasons for a code comment to be fixable for commit, 3) the reasons for a code comment to be unsuitable for commit, and 4) a comparison of existing code comments against automatically generated code comments.

There are three observations in the results. First, the majority of the partially incorrect code segments are due to incorrect context, where the subject of the sentence is incorrect. Second, the majority of the code comments that we classify as not committable are because the code comments are not useful or too trivial. Developers often modify and reuse code comments, where the only differences are due to grammatical, spelling, or one to two word modifications to the sentence. Third, automatically generated code comments are poor replacements for existing code comments.

Our observations from the research questions pointed out several key areas that can be improved. We do admit our evaluation is subjective. We tried to mitigate this problem by being as objective as possible during the evaluation, and released all the evaluation data on our website.

## Availability

The source code and evaluation results of CloCom are publicly available on our project website, http://asset.uwaterloo.ca/clocom/. It contains the output of CloCom on each project, and the classification detail of all the generated code comments.

## Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada.

## References

[1] K. Aggarwal, Y. Singh, and J. Chhabra, "An Integrated Measure of Software Maintainability," in *Reliability and Maintainability Symposium*, 2002, pp. 235–241.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A Study of the Documentation Essential to Software Maintenance," in *International Conference on Design of Communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[3] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *Automated Software Engineering*, 2010, pp. 43–52.

[4] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically Detecting and Describing High Level Actions within Methods," in *International Conference on Software Engineering*, 2011, pp. 101–110.

[5] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," in *International Conference on Program Comprehension*, 2013, pp. 23–32.

[6] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating Parameter Comments and Integrating with Method Summaries," in *International Conference on Program Comprehension*, 2011, pp. 71–80.

[7] E. Wong, J. Yang, and L. Tan, "AutoComment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering*, 2013, pp. 562–567.

[8] Stack overflow. [Online]. Available: http://stackoverflow.com/

[9] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Working Conference on Mining Software Repositories*, 2013, pp. 319–328.

[10] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[11] Count lines of code. [Online]. Available: http://cloc.sourceforge.net/

[12] C. Roy, J. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[13] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments," in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2005.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *International Conference on Software Engineering*, 2007, pp. 96–105.

[15] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *International Conference on Program Comprehension*, 2008, pp. 172–181.

[16] Eclipse java development tools. [Online]. Available: https://eclipse.org/jdt/

[17] The software and text similarity tester sim. [Online]. Available: http://dickgrune.com/Programs/similarity_tester/

[18] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "XIAO: Tuning Code Clones at Hands of Engineers in Practice," in *Annual Computer Security Applications Conference*, 2012, pp. 369–378.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," in *Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004, pp. 20–20.

[20] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards Building a Universal Defect Prediction Model," in *Working Conference on Mining Software Repositories*, 2014, pp. 182–191.

[21] C. Roy and J. Cordy, "A Survey on Software Clone Detection Research," *Queen's University School of Computing TR*, vol. 115, 2007.

[22] R. P. Buse and W. R. Weimer, "Automatic Documentation Inference for Exceptions," in *International Symposium on Software Testing and Analysis*, 2008, pp. 273–282.

[23] S. Zhang, C. Zhang, and M. Ernst, "Automated Documentation Inference to Explain Failed Tests," in *International Conference on Automated Software Engineering*, 2011, pp. 63–72.

[24] R. P. Buse and W. R. Weimer, "Automatically Documenting Program Changes," in *International Conference on Automated Software Engineering*, 2010, pp. 33–42.

[25] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining Source Code Descriptions from Developer Communications," in *International Conference on Program Comprehension*, 2012, pp. 63–72.

[26] B. Dagenais and M. Robillard, "Recovering Traceability Links Between an API and Its Learning Resources," in *International Conference on Software Engineering*, 2012, pp. 47–57.

[27] J. Kim, S. Lee, S. Hwang, and S. Kim, "Enriching Documents with Examples: A Corpus Mining Approach," *ACM Transactions on Information Systems*, vol. 31, no. 1, pp. 1:1–1:27, 2013.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[29] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *International Workshop on Source Code Analysis and Manipulation*, 2004, pp. 128–135.

[30] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *International Symposium on Static Analysis*, 2001, pp. 40–56.

[31] J. Svajlenko, I. Keivanloo, and C. Roy, "Scaling classical clone detection tools for ultra-large datasets: An exploratory study," in *International Workshop on Software Clones*, 2013, pp. 16–22.

[32] N. Göde and R. Koschke, "Incremental Clone Detection," in *European Conference on Software Maintenance and Reengineering*, 2009, pp. 219–228.

[33] Simian - similarity analyser. [Online]. Available: http://www.harukizaemon.com/simian/

[34] M. Uddin, C. Roy, K. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems," in *Working Conference on Reverse Engineering*, 2011, pp. 13–22.