Mining Question and Answer Sites for Automatic Comment Generation

by

Edmund Wong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Abstract

Code comments improve software maintainability, programming productivity, and software reliability. To address the comment scarcity issue in many projects and save developers' time in writing comments, we propose a new, general automatic comment generation approach, which mines comments from a large programming Question and Answer (Q&A) site. Q&A sites allow programmers to post questions and receive solutions, which contain code segments together with their descriptions, referred to as *code-description mappings*. We develop *AutoComment* to extract such mappings, and leverage them to generate description comments automatically for similar code segments matched in open source projects.

We apply AutoComment to analyze 92,140 Java and Android tagged Q&A posts to extract 132,767 code-description mappings, which help AutoComment generate 102 comments automatically for 23 Java and Android projects. The number of generated comments is still low, but the user study results show that the majority of the participants consider the generated comments accurate, adequate, concise, and useful in helping them understand the code. One of the advantages from mining Q&A sites for automatic comment generation is that human written comments can provide information that is not explicitly in the code.

In the future, we would like to focus on improving both the yield and quality of the generated comments. To improve the yield, we can replace the token-based clone detection tool with one that can detect addition and reordering of lines to increase the number of code matches. To improve the quality, we can apply advanced natural language processing techniques such as semantic role labeling to analyze the semantics of the sentences, or typed dependencies to analyze the grammatical structure of the sentences.

Acknowledgements

I would like to thank my advisor, Professor Lin Tan, for providing me guidance throughout the two years of research. I thank my research group colleagues for being great friends with me. In addition, a thank you to Professor Reid Holmes and Professor Chrysanne DiMarco for reading my thesis and providing valuable feedback.

Dedication

This thesis is dedicated to the ones I love.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Examples and Challenges . . . 3

1.1.1 Example One . . . 3

1.1.2 Example Two . . . 5

1.1.3 Example Three . . . 6

2 AutoComment Design 8

2.1 Code-Description Mapping Extraction . . . 9

2.2 Description Refinement . . . 10

2.3 Code Clone Detection . . . 15

2.4 Code Clone Pruning . . . 15

2.5 Comment Refinement and Selection . . . 17

3 Experimental Methods 21

3.1 Evaluated Projects and Databases . . . 22

3.2 User Study . . . 22

4 Results 24

5 Limitations and Future Work **27**

5.1 Comment Yield 27

5.2 Comment Quality 28

5.3 User Study 29

6 Related Work **30**

6.1 Automatic Comment Generation 30

6.2 Mining Descriptions for Code Artifact 31

6.3 Code Clone Detection 31

6.4 Automatic Code Generation 32

7 Conclusions **33**

APPENDICES **34**

A Ethics Clearance Approval **35**

B Sample User Study Questionnaire **37**

References **39**

List of Tables

2.1 List of Terms for Sentence Filtering 11

2.2 Explanation for Equation 2.1 - **VP-NP** 13

2.3 Explanation on Equation 2.2 - **NP-VP** 14

2.4 Generic Method List 16

3.1 Evaluated Open Source Projects 22

4.1 Human Judgements on the Generated Comments.
Ac: Accuracy; Ad: Adequacy; Co: Conciseness; Us: Usefulness 25

List of Figures

1.1 Code from Java project Jajuk 3
1.2 StackOverflow Post #10854831 4
1.3 Code from Java project MegaMek 5
1.4 StackOverflow Post #11332280 5
1.5 Code from Android project Barcode Scanner 6
1.6 StackOverflow Post #4665122 7

2.1 Overview of AutoComment 8
2.2 Frequency vs. Vote Count for Java Answers 9
2.3 Parse Tree for the sentence "You can use this method to capture the stack-trace in a String". The matched Tregex patterns are labelled in bold. . . . 12
2.4 Example of a piece of template code. StackOverflow post #5035894. 17
2.5 An example to show the replacement of the variable name `spinnerArray` with `settings` in the description sentence. StackOverflow post #7173323. 18
2.6 An example to show the discarding of a sentence because the code artifact `SENDTO` does not exist in the code segment in the target software. The code uses `ACTION_VIEW` as the intent value in line 2. StackOverflow post #5802974. 19

Chapter 1

Introduction

Code commenting has been an integral part of software development. Comments improve software maintainability [1] and programming productivity through helping developers understand code and improve software reliability through assisting in detecting software defects [2]. Code commenting has been a standard practice in the industry. Despite the need and importance of commenting code, many code bases are not commented or not adequately commented [3].

In addition, it is time-consuming for developers to write comments. It would be beneficial to generate comments automatically when possible so that developers can spend their valuable time on other tasks.

Recently, researchers have proposed techniques to generate comments automatically from source code. Sridhara et al. automatically generate a summary comment for a Java method [4]. They leverage the code structure and identifier names to generate one comment sentence for each chosen statement from the method, and concatenate the comment sentences to form a summary comment for the method. In a followup project, they identify statements with similar structures and topics, and generate comments for the group of statements [5]. While these techniques are successful initial steps toward automatic comment generation, they have two main limitations. First, the techniques can only generate comments for specific code structures (e.g., one method body [4], groups of method calls [5], or groups of if-else statements [5]). Second, the performance of the previous work depends on high quality identifier names and method signatures. For example, when grouping methods calls, it requires that all method names contain the same verb [5]. If the identifiers and methods have poor names, then the approach may fail to generate accurate comments or any comments at all.

We propose a new approach to generate comments automatically [6] to address the above limitations. We observe that Question and Answer (Q&A) sites such as StackOverflow [7] naturally contain code descriptions written by developers that can be used for automatic comment generation. Specifically, StackOverflow [7] is widely used to ask questions about code development, debugging, etc. Those questions often receive high quality answers due to the large user base. For example, one asked, "how to open the find type dialog programmatically in Eclipse". The question received a Java code snippet that performs the asked task. We can use the statement form of the question "open the find type dialog programmatically in Eclipse" as an explanatory description of the code snippet. We refer to the code snippet and description as a *code-description mapping.* If the code segment in a software project is identical or similar to the above code snippet, then the corresponding description can be an explanatory comment for the code segment in the software project.

Q&A sites such as StackOverflow [7] contain a wealth of information, which makes it a feasible and valuable data source for extracting code-description mappings for automated comment generation. For example, StackOverflow contains a total number of 6,943,267 posts as of March 2014. At least 49% of the Java and Android classes in StackOverflow have at least one code example in the accepted answer [8]. Android code snippets have a mean size of 16.4 lines of code (LOC) and a median of 9 LOC [9].

The idea is to generate comments automatically by mining Q&A sites for code-description mappings. The prototype, *AutoComment*, has two main components. The first component extracts a database of code-description mappings from Q&A sites. The second component searches for similar code segments between the extracted database and given software projects. Once AutoComment finds an identical or similar code segment it presents the corresponding description as a comment to explain the matched code segment. One key benefit of AutoComment is that *the description is what a developer uses to describe the code segment*, which is likely to be accurate and useful for developers to understand (compared to descriptions generated from variable names and method names).

The thesis makes the following contributions:

- We propose a new approach, AutoComment, to generate code comments automatically by analyzing Q&A sites.
- We apply AutoComment on 23 projects (16 Java projects and 7 Android projects) to generate 102 comments automatically. We conduct a user study, which demonstrates that the majority of the participants find the generated comments accurate, adequate, concise, and useful.

- We adopt natural language processing (NLP) techniques and design heuristics to improve the code descriptions to generate high-quality comments.
- AutoComment builds databases of code-description mappings that can be leveraged for purposes other than automated comment generation such as program synthesis.

1.1 Examples and Challenges

In this section, we present three examples illustrating how AutoComment generates comments automatically. We describe the challenges, summarize our solutions, and highlight the unique benefits of AutoComment.

1.1.1 Example One

Figure 1.1 shows a code segment from the Java project Jajuk.

```
1| public String getToolTipText(MouseEvent e) {
2|   java.awt.Point p = e.getPoint();
3|   int rowIndex = rowAtPoint(p);
4|   int colIndex = columnAtPoint(p);
5|   if (rowIndex < 0 || colIndex < 0) {
6|     return null;
7|   }
8|   ...
9| }
```

Figure 1.1: Code from Java project Jajuk

AutoComment generates the following comment to explain the code segment highlighted in grey automatically:

Find on which row and column the mouse is.

Our user study results show that users consider this comment accurate in describing this piece of code and useful in helping them understand the code. The previous technique [5] would not generate a comment for this example because the three method names in Line 2–4 share no common verb.

`StackOverflow Question (Title):`

Tool tip in JPanel in JTable not working

`StackOverflow Answer:`

The problem is that you set tooltips on subcomponents of the component returned by your CellRenderer. To perform what you want, you should consider override getToolTipText(MouseEvent e) on the JTable. From the event, you can find on which ***row*** *and* ***column*** *the mouse is, using:*

```
java.awt.Point p = e.getPoint();
int rowIndex = rowAtPoint(p);
int colIndex = columnAtPoint(p);
```

Figure 1.2: StackOverflow Post #10854831

Figure 1.2 shows the StackOverflow post that AutoComment leverages to generate the comment. It shows the title of the post, the code snippet, and one paragraph immediately before the code snippet in the answer.

Challenges in Comment Selection: Figure 1.2 shows two textual descriptions that can be leveraged to describe the code segment in the answer. One is the title of the post, which describes the question. The other is the paragraph immediately before the code segment, which consists of three sentences. Among the four sentences in the title and the answer paragraph, only the last sentence in the answer paragraph describes the code snippet. AutoComment needs to select this relevant sentence from the four sentences to generate the comment automatically, which is challenging.

AutoComment uses two techniques to address this comment selection challenge. First, many sentences ask and answer how to troubleshoot their code (e.g., the title and the first sentence in the answer paragraph in Figure 1.2). These sentences often do not describe the code segment. Therefore, AutoComment removes sentences that imply troubleshooting based on keyword filtering. For example, "not" indicates that the title describes a troubleshooting problem rather than the code segment; and "problem" in the first sentence from the answer suggests the cause of the problem. Therefore, AutoComment filters out both sentences (Section 2.2). Second, AutoComment leverages the *text similarity* between each sentence and the code segment to identify the most relevant sentences (Section 2.5). In Figure 1.2, the shared words between the text and code are in bold (**row** and **column**).

1.1.2 Example Two

Figure 1.3 shows a code segment from the Java project Megamek.

```java
private String getStackTrace(Throwable throwable) {
    StringWriter swriter = new StringWriter();
    PrintWriter pwriter = new PrintWriter(swriter);
    throwable.printStackTrace(pwriter);
    pwriter.flush();
    pwriter.close();
    return swriter.toString();
}
```

Figure 1.3: Code from Java project MegaMek

AutoComment generates the following comment for the code segment highlighted in grey:

Receive a stack trace. Use this method to capture the stacktrace in a String.

Figure 1.4 shows the post in StackOverflow that AutoComment leverages to generate this comment. AutoComment detects that the two code segments had partially matched, and generates the comment by combining the two sentences from the title and the answer as these two sentences have the same text similarity (Section 2.5).

`StackOverflow Question:`
Is it possible in Java's MessageFormat to receive a stack trace?
`StackOverflow Answer:`
You can use this method to capture the stacktrace in a String

```java
public String getStackTrace(Throwable t) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    t.printStackTrace(pw);
    pw.flush();
    return sw.toString();
}
```

Figure 1.4: StackOverflow Post #11332280

Challenges in Comment Refinement: The sentences from question titles and the answers are often in a question form (e.g., "How to ...?", "Is it possible to ...?") or contain excessive information (e.g., "You can ..."). Directly using these sentences will lead to low quality comments.

To address this challenge, we deploy natural language processing (NLP) techniques to extract the core parts of a sentence. In Figure 1.4, AutoComment extracts "Receive a stack trace" from the title, and "Use this method to capture the stacktrace in a String" from the answer. AutoComment looks for a subtree that contains a verb phrase (VP) and a noun phrase (NP) from the parse tree of a sentence (Section 2.2). In addition, it removes clauses that are connected by a coordinating conjunction (i.e., "but" and "yet"), personal pronouns (e.g., "you" in the example), and code artifacts (e.g., class/method/field/constant names and primitive data types) that do not exist in the code segment.

Challenges in Code Matching: Finding similar code segments between StackOverflow and the input projects requires code clone detection techniques (Section 2.3). However, since the code segments from StackOverflow are often incomplete and uncompilable, AutoComment uses token-based clone detection instead of Abstract Syntax Tree (AST)-based clone detection.

The two code segments in Figure 1.3 and 1.4 are slightly different in terms of the variable names (e.g., `swriter` vs. `sw`). On top of that, code in Figure 1.3 has one additional line (Line 6) compared to that in Figure 1.4. However, such a small difference does not affect the semantic similarity of the two pieces of code, which should be detected as matched code. We solve this problem by extending the matching algorithm to enable line skipping.

1.1.3 Example Three

Figure 1.5 shows a code segment from the Android project Barcode Scanner.

```
private static Bitmap toBitmap(LuminanceSource source, int[] pixels) {
  int width = source.getWidth();
  int height = source.getHeight();
  Bitmap bitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
  bitmap.setPixels(pixels, 0, width, 0, 0, width, height);
  return bitmap;
}
```

Figure 1.5: Code from Android project Barcode Scanner

AutoComment generates the following comment for the lines highlighted in grey:

Create ***empty*** *bitmap with dimensions of original image and ARGB_8888 format.*

Figure 1.6 shows the post in StackOverflow that AutoComment leverages to generate the comment.

`StackOverflow Question (Title):`
Android Pass Bitmap to Native in 2.1 and lower
`StackOverflow Answer:`
Create empty bitmap with dimensions of original image and ARGB_8888 format:

```
int width = src.getWidth();
int height = src.getHeight();
Bitmap dest = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
```

Figure 1.6: StackOverflow Post #4665122

Benefits of AutoComment: AutoComment generates a comment to provide important information that is not explicitly in the code, e.g., the code is to create an "**empty**" bitmap. In addition, AutoComment can naturally group the three statements into a semantic unit for comment generation because developers have already grouped so in the StackOverflow post. Such grouping is general because it does not rely on the quality of the method names or the structure of the methods, which is different from previous work [5]. The grouping is also reliable because most of the code segments are small snippets that are meant for demonstration purposes.

Chapter 2

AutoComment Design

The overview of AutoComment is in Figure 2.1. AutoComment takes two inputs: (1) a StackOverflow database dump containing information of all posts; and (2) source code of the target projects. The output is a list of code segments and the corresponding comments generated by AutoComment.

AutoComment consists of two major components. The first component generates databases of code-description mappings (Section 2.1) and leverages natural language processing (NLP) techniques to refine the descriptions (Section 2.2). The second component generates comments for the target software. It applies code clone detection technique to identify matched code between the database and the target software (Section 2.3), prunes out the bad matches (Section 2.4), and selects the best comment for the matched code (Section 2.5).

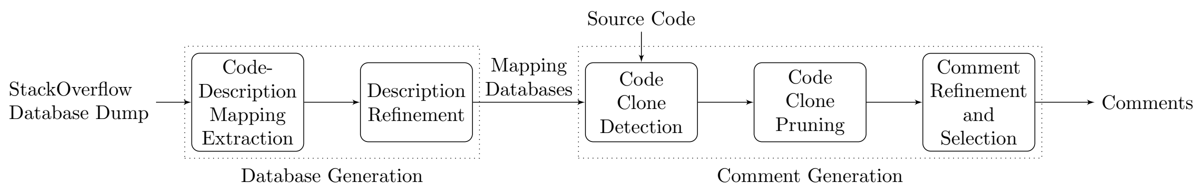

Figure 2.1: Overview of AutoComment

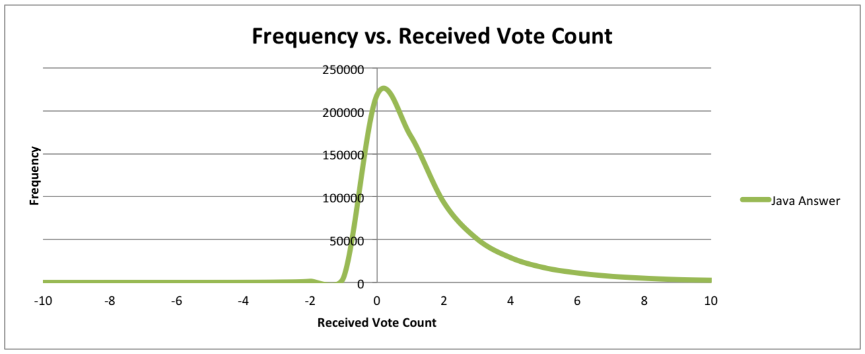

Figure 2.2: Frequency vs. Vote Count for Java Answers

2.1 Code-Description Mapping Extraction

We choose a programming Q&A site called StackOverflow [7] as the data source to build databases of code-description mappings. StackOverflow contains questions from diverse software domains (e.g., Java, Android, C++, etc.), each associated with its respective tag. Questions and answers from one software domain are unlikely to benefit software projects from a different domain. For example, an answer about how to use Android APIs is unlikely to help AutoComment generate comments for a Java desktop application. Therefore, we build and apply code-description mappings extracted from Java questions (tagged with `java`) to Java projects; and Android-related questions (tagged with `android`) to Android projects.

StackOverflow contains invalid and low-quality questions and answers. To ensure the quality of extracted code-description mappings, AutoComment selects questions and answers based on the number of votes it received from the voting system that StackOverflow deploys. Figure 2.2 shows the frequency distribution of the score count for all 621,017 Java tagged answers, which has a mean of 1.89, medium of 1 and mode of 0. AutoComment only keeps questions with a non-negative number of votes. For each kept question, it selects the answer(s) with the highest positive number of votes.

The title of a post is not the only description for the code segment (Figure 1.4). Since it is common for people to write a code description immediately before the code segment, we also extract the paragraph immediately before the code segment as a comment candidate.

Based on the ideas elaborated above, for each post in StackOverflow, AutoComment uses the following steps to build the initial databases of code-description mappings:

1. discard a post if its question receives a negative number of votes,

2. check the tags of the post and discard posts that are not relevant to the target domain (e.g., Java and Android),

3. select the answer(s) that has the highest positive number of votes (select multiple answers if they have the same highest positive number of votes), and

4. for each code segment in the answer (based on the HTML tag `<code>`), map it against the title of post and one paragraph before the code segment (based on the HTML tag `<p>`) as candidate descriptions.

We had also attempted to extract description sentences from other parts of a post. For example, instead of only extracting from the paragraph immediately before the code segment, we tried to include description sentences from 1) two paragraphs before the code segment, and 2) one paragraph after the code segment (only if there are no code segments following this paragraph). We experimented with extracting these description sentences, but the majority of them are not describing the code segment.

2.2 Description Refinement

The description sentences extracted from StackOverflow are often in a question form or contain unnecessary information. To improve the quality of descriptions, AutoComment leverages NLP techniques to perform refinements. This includes filtering of invalid descriptions and extracting core parts of the descriptions. AutoComment refines the descriptions using techniques in the following order.

Description Segmentation: Since we extracted textual paragraphs for the code segments, AutoComment needs to split the paragraphs into sentences based on the period character. To avoid incorrect split due to the dot operator (commonly used in programming languages), AutoComment only treats a period character as the end of a sentence if there is a space immediately afterwards (similar to iComment [2]).

Description Filtering: As discussed in Section 1.1.1, sentences that ask and answer how to troubleshoot code often do not describe the code segment. For example, "<u>Why</u> this code does <u>not</u> work?" and "Android: <u>problem</u> retrieving bitmap from database". AutoComment filters out such sentences based on the manually collected terms shown in Table 2.1. We use the same list of filtering terms for both Java and Android databases.

Main Sub-Tree Extraction: Sentences that are in a question form or contain personal pronouns (e.g., "you") are not suitable as comments. Therefore, we adapt NLP techniques

no, not, error, bug, difficult, difficulty, problem, problems, fix, shouldn't, doesn't, can't, couldn't, don't, isn't, aren't, wouldn't, fail, why, what, null, bad, wrong, missing, lack, probably, likely, perhaps, think, may, maybe, unfortunately, unluckily

Table 2.1: List of Terms for Sentence Filtering

with two objectives: 1) to convert the sentences in a question form to a statement form, and 2) to extract the core parts of the sentences. We achieve them by identifying and extracting the main sub-tree of a sentence.

There are three steps to extract the main sub-tree of a sentence:

1. to generate a parse tree from the input sentence,
2. to obtain all the sub-trees that match the specified patterns, and
3. to merge all the matched sub-trees together to form a refined sentence.

Step one generates a parse tree using Stanford CoreNLP[1] (v1.3.4). AutoComment first uses CoreNLP's part-of-speech (POS) tagger [10] to label the part of speech of each word of a sentence, then uses the statistical parser [11] to generate the parse tree. Figure 2.3 shows the parse tree for the sentence in Figure 1.4. The leaf nodes are the words, and the parent node of a leaf node shows its POS tag.

CoreNLP does fall short on interpreting certain technical terms because it was trained on well written text such as the Wall Street Journal. For example, it would sometimes tag the word, "file", incorrectly as a verb instead of a noun depending on the sentence structure. However, it is robust at parsing sentences and works well for our experiments.

Step two extracts the main sub-trees from the parse tree. The idea is to obtain sub-tree(s) that contains at least one verb phrase (VP) and one noun phrase (NP), which ensures each extracted phrase has a verb associated with a subject or an object.

We define two patterns, Equation 2.1 and Equation 2.2, in Stanford's Tregex[2] format to extract the main sub-tree(s) of a parse tree. Table 2.2 and Table 2.3 explain these two patterns respectively.

$$\textbf{VP-NP}: VP << (NP < /NN.?/) < /VB.?/ \tag{2.1}$$

[1]`http://nlp.stanford.edu/software/corenlp.shtml`
[2]`http://nlp.stanford.edu/software/tregex.shtml`

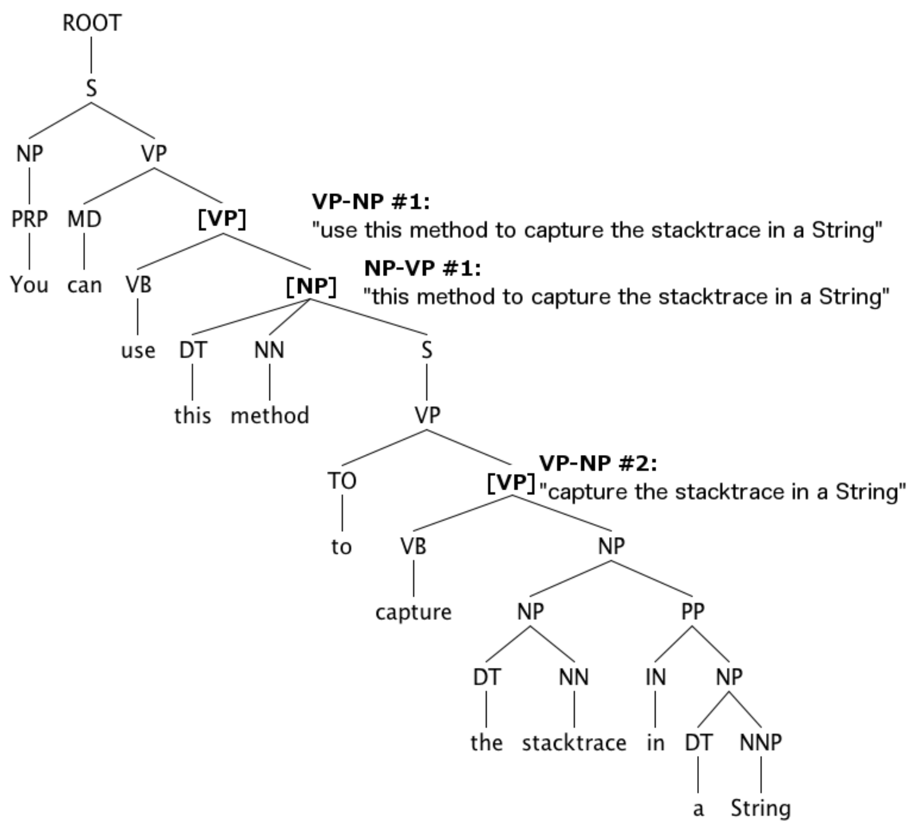
ROOT
S
NP
VP
PRP
MD
[VP]
You
can
VB
[NP]
use
DT
NN
S
this
method
VP
TO
[VP]
to
VB
NP
capture
NP
PP
DT
NN
IN
NP
the
stacktrace
in
DT
NNP
a
String
VP-NP #1:
"use this method to capture the stacktrace in a String"
NP-VP #1:
"this method to capture the stacktrace in a String"
VP-NP #2:
"capture the stacktrace in a String"

Figure 2.3: Parse Tree for the sentence "You can use this method to capture the stacktrace in a String". The matched Tregex patterns are labelled in bold.

$$\textbf{NP-VP}: NP\,! < \,PRP\,[<<\,VP\,|\,\$\,VP] \tag{2.2}$$

The two patterns would exclude personal pronoun (PRP) words. Personal pronouns typically contribute no value in a code comment, so it is safe to remove them. Penn Treebank tag guideline [12] defines personal pronouns to include personal pronouns proper ("I", "me", "you", "he", "him", etc.), reflexive pronouns ending in *-self* or *-selves*, and nominal possessive pronouns "mine", "yours", "his", "ours" and "theirs".

We show how to remove "You can" from the sentence in Figure 2.3 using the two patterns. AutoComment finds three sub-trees that are matched by the patterns, and then merges them (step three), which produces a sentence without "You can". We label the matched VP-NP and NP-VP patterns on the right hand side of the figure. The conditions

Regular Expression	**Explanation**	**Rationale**
VP << (NP < /NN.?/)	Verb phrase (VP) that is an ancestor of a noun phrase (NP)	Ensures the sentence starts with a VP that includes an NP.
NP < /NN.?/	Noun phrase (NP) that is the parent of the basic category of a noun (NN)	Ensures the NP have at least one noun that is not a personal pronoun (PRP), but the NP will be allowed to contain personal pronouns.
VP < /VB.?/	Verb phrase (VP) that is the parent of the basic category of a verb (VB)	Ensures the VP have at least one verb that is not a modal verb (e.g., can, must, should, will).

Table 2.2: Explanation for Equation 2.1 - **VP-NP**

that each matched sub-tree had satisfied are as follow:
VP-NP #1: "use this method to capture the stacktrace in a String"

1. The VP (highlighted as [VP]) is an ancestor of the NP, "this method to capture the stacktrace in a String".
2. The NP is the parent of a noun, "method".
3. The VP is the parent of a verb, "use".

NP-VP #1: "this method to capture the stacktrace in a String"

1. The NP (highlighted as [NP]) is not the parent of a personal pronoun.
2. The NP is an ancestor of the VP, "to capture the stacktrace in a String".

VP-NP #2: "capture the stacktrace in a String"

1. The VP (highlighted as [VP]) is an ancestor of the NP, "the stacktrace in a String".
2. The NP is the parent of a noun, "stacktrace".
3. The VP is the parent of a verb, "capture".

Regular Expression	**Explanation**	**Rationale**
NP !<PRP	Noun phrase (NP) that is not the parent of a personal pronoun (PRP)	NP that is a personal pronoun offers no value to the comment, thus excluded.
NP [<< VP | $ VP]	Noun phrase (NP) that is either the ancestor or sister of a verb phrase (VP)	Ensures the sentence starts with an NP followed by a VP. VP that are followed after an NP often appear on the same level in a parse tree.

Table 2.3: Explanation on Equation 2.2 - **NP-VP**

For the other sentence in the motivating example, "Is it possible in Java's MessageFormat to receive a stack trace?", AutoComment extracts the main sub-tree as "Receive a stack trace" because it matched with Equation 2.1.

Step three performs merging on the extracted sub-trees in the case where there are more than one sub-tree, and outputs a single sentence. For example, the parse tree in Figure 2.3 contains three matched sub-trees (VP-NP #1, NP-VP #2, and VP-NP #2). To generate a single sentence from the multiple matched sub-trees, AutoComment calls the method "join node" on all the sub-trees: *Given two sub-trees, locate node j such that j dominates both sub-trees, and return a tree with node j as the root of the tree.* In this example, since the first sub-tree dominates the second and third sub-trees, the "join node" operation returns the first sub-tree as the generated comment.

Clause Removal: A sentence may contain more than one clause connected by a coordinating conjunction (CC). The following sentence contains two clauses linked by the CC word "but":

How do I read in a file with buffer reader <u>but</u> skip comments with java

The seven coordinating conjunctions are "for", "and", "nor", "but", "or", "yet", and "so" [13]. The CC words "but" and "yet" imply a contrasting meaning. Therefore, AutoComment removes the clause after the CC word "but" and "yet".

However, this technique can potentially remove important information from a sentence. In the future, we would like to analyze the content of the clause prior to the removal.

Number Removal: AutoComment removes numerical numbers from a sentence to make it general by detecting the POS tag—Cardinal numbers (CD), which represents numeric words such as “three” and “3”. For example, in the sentence “Display three non-negative integers in increasing order”, we remove the word ‘three’ from the sentence.

2.3 Code Clone Detection

We extend an existing token-based clone detection tool SIM [14] to detect matched code segments between StackOverflow and input projects. SIM tokenizes the two input code and uses the longest common substring algorithm to detect code clones. It requires exact matching on method names and programming language keywords. We extended SIM with stricter matching requirements. Specifically, the value of strings and characters, class names and static/non-static fields require exact matching. In addition, we modified the matching alogrithm to allow lines from the code segment in the target software to be skipped while requiring all the lines from the StackOverflow code segment to be matched as illustrated in the example shown in Figure 1.3 and Figure 1.4. We configured the maximum number of lines that can skip to 4.

SIM utilizes the initial token run length value to build up a forward reference table for locating common substrings. We configured the initial token run length to be 20. It means that a match must contain 20 or more consecutive tokens that are the same, which equates to roughly two source code statements. A smaller token run length increases the number of detected code clones, but it also 1) increases the number false positives because less content is matched, and 2) increases the runtime because SIM hashes the tokens based on the initial run length value to avoid string comparisons.

We considered using AST-based clone detection tools. However, the partial code segments from StackOverflow are often uncompilable. A more recent work [15] resolved the issue at compiling StackOverflow code segments. It adds wrappers to the code segment to allow parsing and identifies the fully qualified names of all the code elements in the code segment, which can enable the use of more advanced clone detection tools.

2.4 Code Clone Pruning

The output of the code clone detection tool consists of pairs of code segments that have a similar syntactical structure. To generate accurate and useful comments, it is important

add, remove, put, post, get, set, read, write, delete, close, exit, hashCode

Table 2.4: Generic Method List

to ensure a high level of semantic matching. For example, matching the two generic code segments `x++; y++;` and `i++; j++;` is unlikely to help AutoComment generate useful comments for program comprehension.

Support Set Pruning: The more number of times that a StackOverflow code segment gets matched, the higher the probability that it is a generic match. This heuristic is capable of eliminating generic code. Specifically, if a StackOverflow code segment is matched five or more times with the source code within the same project, AutoComment prunes out such pairs of code segments.

Line Percentage Matching: In general, the higher proportion of matched lines in the StackOverflow code segment, the higher probability that the description sentence is applicable to the matched code segment in the target software. Therefore, AutoComment calculates the percentage matching score as a filtering metric.

Specifically, for each StackOverflow code segment, we exclude all source code lines that are a Java annotation, comment, method signature or return statement prior to the percentage calculation. We call the remaining lines *effective lines*. We define a *non-generic line* as a line that does not contain a generic method call in Table 2.4, because we find that a line of code that contains a generic method call contributes little to the semantic matching. AutoComment calculates the percentage matching score using the following formula with a 70% threshold, meaning that at least 70% of the effective lines has to be matched.

$$PercMatched = \frac{\text{number of matched effective non-generic lines}}{\text{number of effective lines in the StackOverflow code segment}}$$

Removal of Repetitive Method Calls: If a matched code segment in the target software only contains repetitive method calls (three or more times), it is performing a similar operation repetitively with different parameters. Since the value of the parameters may largely impact the functionality of the code segment in the target software, and AutoComment does not require the value of the parameters be exactly matched (Section 2.3), such matches are removed to guarantee the accuracy.

Removal of Template Code: Some StackOverflow answers simply provide a template with placeholders to be filled. The semantics of the filled template and the empty template can be quite different. Figure 2.4 shows a code segment that performs a generic file read operation, but the comment is too specific because the content within the curly bracket between line 3 and 5 is missing. We consider a StackOverflow code segment that contains a pair of curly braces with no statements in it as a template. AutoComment removes such StackOverflow code segments by requiring at least one statement within a pair of curly braces.

StackOverflow Question:

Fastest way to read a file line by line with 2 sets of Strings on each line?

StackOverflow Answer:

```
1 BufferedReader br = new BufferedReader(new FileReader(file));
2 String line;
3 while((line = br.readLine()) != null) {
4   // do something with line.
5 }
```

Figure 2.4: Example of a piece of template code. StackOverflow post #5035894.

Other Filters: To ensure high semantic matching, AutoComment requires the matching of at least one line that contains a method call. In addition, AutoComment filters out matches that contain the term "Exception" because exception code is inherently different from the main flow code. To explain exception code, we may combine AutoComment with the previous technique [16]. Furthermore, AutoComment prunes out long code matches (over fifteen lines of code) because they often contain multiple semantic units, and StackOverflow is unlikely to contain detailed enough descriptions.

2.5 Comment Refinement and Selection

For each remaining match, there can be one or more description sentences available as a comment candidate. If the code from the given projects matches with multiple StackOverflow code segments, AutoComment includes all of the available description sentences of each StackOverflow code segment as a candidate. AutoComment refines and selects the comment candidate(s) that best describes the matched code segment in the target software using techniques in the following order.

Variable Name Replacement: Description sentences from StackOverflow often contain variable names that appear in the code segment. In the sentence in Figure 2.5, spinnerArray refers to the variable name in the StackOverflow code segment, but the code segment in the target software uses a different name called settings. This renders the sentence invalid for the code segment in the target software. To solve this, AutoComment replaces spinnerArray with settings in the sentence. It does this by performing tokenization on both code segments to obtain two lists of tokens. Since the code clone detection tool guarantees the syntactical structure of the matched part of two segments to be the same, spinnerArray and settings can be mapped against each other.

Description Sentence(s):

Let's say your array is called spinnerArray, you can use a ArrayAdapter to talk to the spinner:

StackOverflow Code Segment:

```
ArrayAdapter<String> spinnerArrayAdapter = new ArrayAdapter<String>
    (this,android.R.layout.simple_spinner_dropdown_item,
    spinnerArray);
    spinner.setAdapter(spinnerArrayAdapter);
```

Input Code Segment:

```
ArrayAdapter<String> adapterSettings1 = new ArrayAdapter<String>(
    this, android.R.layout.simple_spinner_dropdown_item,
    settings);
    spinnerSettings1.setAdapter(adapterSettings1);
```

Figure 2.5: An example to show the replacement of the variable name spinnerArray with settings in the description sentence. StackOverflow post #7173323.

Code Artifact Matching: Code artifact matching detects code artifacts (e.g., class/method-/field/constant names and primitive data types) which exist in a description sentence, but do not exist in the method that contains the matched code in the target software. In such cases, AutoComment removes the sentence. AutoComment detects code artifacts using regular expressions combined with camel cases.

Figure 2.6 shows a code artifact mismatch example. We show the description sentence from StackOverflow and the code segment from Android OsmAnd. Since the code uses ACTION_VIEW instead of SENDTO when initializing the Intent object, the sentence is not an accurate description of the code segment. AutoComment detects constant SENDTO in the description sentence, but it cannot find this constant in the matched code segment in the target software or the surrounding code within the method. Therefore, it filters out this sentence.

Text Similarity: To select the best description sentences from the remaining sentences, AutoComment measures the *text similarity* between each remaining description sentence

`Description Sentence(s):`
Android.intent.action.SENDTO : Displays activity com.android.mms/.ui.ConversationList in Galaxy tab.
`Input Source Code [Android OsmAnd]:`

```
1 private void sendSms(String sms) {
2   Intent sendIntent = new Intent(Intent.ACTION_VIEW);
3   sendIntent.putExtra("sms_body", sms);
4   sendIntent.setType("vnd.android-dir/mms-sms");
5   mapActivity.startActivity(sendIntent);
6 }
```

Figure 2.6: An example to show the discarding of a sentence because the code artifact `SENDTO` does not exist in the code segment in the target software. The code uses `ACTION_VIEW` as the intent value in line 2. StackOverflow post #5802974.

and the code segment in the target software. After that, it selects the sentences that achieve the highest text similarity score as the comment. AutoComment follows the following steps to measure text similarity:

Step One AutoComment extracts tokens from the code and the description sentence. A token is a consecutive sequence of at least three word characters (alphabets and numbers). It also considers the dot operator as a word character. For example, it extracts `obj.function()` as `obj.function` instead of two tokens `obj` and `function`. This is because `obj.function()` is an atomic function name and should only contribute to the text similarity once.

We experimented with camel case detection for extracting tokens. For example, the technique will extract `PrintWriter` as two tokens, `print` and `writer`. However, the technique creates a bias towards selecting description sentences that contain code artifacts, because it will extract more tokens and increase the text similarity score for such sentences. Therefore, we did not deploy camel case detection.

Step Two It lemmatizes [17] the tokens from their inflected forms to the base form, e.g., converting 'takes' to 'take'. Then it removes duplicate tokens and stop words (including "new", "the", "and", "but", "for", and "you").

We also experimented with stemming instead of lemmatization, including the Porter Stemmer and the Snowball Stemmer. Stemming removes inflectional endings from a word to achieve a base form, as opposed to lemmatization, which uses a vocabulary to return the base form. They all perform equally well with negligible differences.

Step Three It calculates the text similarity as the number of overlapping token pairs between the description sentence and the code using common substring matching. For example, `BufferedImage` and `Image` are one overlapping pair. It also discards sentences that only contain a single text similarly term that is a primitive data type such as `int`, because the similarity content is insufficient.

AutoComment selects the sentences which achieve the highest text similarity. If multiple sentences have the same highest text similarity, it combines all of them as the generated comment.

Chapter 3

Experimental Methods

To evaluate the quality of the comments generated by AutoComment, we conducted a user study similar to that of Sridhara et al. [5]. The ethics clearance notification for the user study had been attached in Appendix A. We examined the following two research questions:

- **RQ1:** Are the automatically generated comments *accurate*, *adequate*, and *concise* in describing the code?
- **RQ2:** Are the automatically generated comments *useful* for developers to understand the code?

RQ1 rates a comment based on its accuracy (i.e., does not contain incorrect information), adequacy (i.e., contains a sufficient amount of information), and conciseness (i.e., expresses much in a few words).

RQ2 is a new research question that Sridhara et al. [5] did not evaluate. It is an important question to evaluate because a comment can be accurate, adequate, and concise, but does not help developers understand the code. For example, if a comment is a simple rephrase of the statement, e.g., "get the name and age of a student" for the statements `getName(student); getAge(student);` or "increment the variable x and y" for the statements `x++; y++;`, a developer may find the comments useless. Therefore, in our user study, we explicitly ask if the generated comment help in understanding the code segment.

Java Project	**LOC**	**Android Project**	**LOC**
Eclipse SDK	4,678,435	Firefox	180,162
FreeCol	205,471	Chrome	75,652
FreeMind	113,929	Barcode Scanner	55,121
GanttProject	164,059	FBReader	69,927
Hibernate	708,258	KeePassDroid	42,073
HSQLDB	115,829	myTracks	54,001
JabRef	153,285	osmAnd	204,253
Jajuk	126,149		
JavaHMO	39,481		
JBidWatcher	36,228		
JFtp	32,347		
JHotDraw	56,388		
MegaMek	387,739		
Planeta	33,815		
Sweet Home 3D	104,831		
Vuze	852,622		

Table 3.1: Evaluated Open Source Projects

3.1 Evaluated Projects and Databases

We apply AutoComment to analyze 92,140 StackOverflow posts to extract two databases. We extract 87,785 code-description mappings from the `java` tag and 44,982 code-description mappings from the `android` tag. We apply the Java database on 16 Java projects and the Android database on 7 Android projects for a total number of 23 open source projects. The 16 Java projects include the 15 Java projects that were evaluated by Sridhara et al. [5], and one additional Java project, Eclipse. In addition, Sridhara et al. [5] did not evaluate on Andriod projects. Table 3.1 shows the total number of lines of code for each project. AutoComment generates a total of 102 comments for the 23 projects.

3.2 User Study

We conducted a user study to answer the two research questions. The study included 15 human participants to rate the comments generated by AutoComment. We recruited student participants from University of Waterloo's Computer Science and Electrical and Computer Engineering department using their internal mailing list. The evaluator group includes 1 undergraduate and 14 graduate students, all of whom have industrial experience in Java programming. The participants have an average of 5.4 years of programming experience, ranging from 2 years to 10 years.

Questionnaire Generation: We provide each user with a questionnaire of 15 generated comments (10 from Java projects and 5 from Android projects) to evaluate. For each questionnaire, we randomly select the comments from all the comments generated by AutoComment. We continue this random sampling for each questionnaire until all the comments have been selected, and then reset the sampling basket. A code segment in the databases can be matched several times at different locations in the target project. Therefore, we enforce that each user only evaluates the same code segment once to avoid repetitive evaluations by the same user.

Evaluation Procedure: The user study evaluation has two steps: 1) show the users the code segment and ask them to write a comment for the code (users may give up on writing a comment if they find it difficult), and 2) provide users with the AutoComment generated comment and ask them to rate the comment on accuracy, adequacy, conciseness, and usefulness.

The first step is to ensure that users had spent time to understand the code segment before rating the AutoComment generated comment. For each code segment, we also show the surrounding code to help the users understand the code segment, including the method which contains the code segment and the existing comments related to the code segment. To avoid overwhelming the users with excessive information, we show at most 20 surrounding code lines per code segment.

The second step is to rate the comments with the five-point Likert scale [18]. Following Sridhara et. al's [5] recent work about rating comments of high level actions within methods, we use the following five-point Likert scale: (1) *Strongly Disagree*, (2) *Disagree*, (3) *Neutral*, (4) *Agree*, (5) *Strongly Agree*. To reduce bias, we do not present the synthesized comments to the users until they have completed step one. We do this by placing the questions of step two in a different page and requesting the participants not to start step two until they have completed step one.

Availability: The extracted code-description databases and generated comments are available at `http://asset.uwaterloo.ca/AutoComment/`. We attached a sample question of the user study questionnaire in Appendix B.

Chapter 4

Results

We show the human judgement results from the user study in Table 4.1. A total number of 102 comments are generated from the 23 Java and Android projects. Each of the 15 participants evaluated 10 Java comments and 5 Android comments, which results 150 and 75 responses for two project domains respectively.

The results show that AutoComment can generate comments for high level groups of code statements. Based on the 225 responses, majority of the users find the generated comments accurate, adequate, concise, and useful in helping them understand the code segments.

Amongst the 102 comments that AutoComment generates, 78 comments' corresponding code segments in the target projects do not already have a comment describing the code segments. For the 24 code segments that already contain comments, seven of the AutoComment-generated comments complement the existing comments (i.e., provide additional useful information), 14 are similar to the existing comments, and three are not as good (e.g., less useful) as the existing comments. For example, the AutoComment-generated comment, "Combine integer arrays. System.arraycopy is a method you can use to perform this copy." complements the existing comment "just add". As a future improvement, we can compare AutoComment-generated comments with existing comments using text similarly to filter out those that are less useful.

Below we discuss the breakdown results of Java and Android comments and the detailed results for the four evaluation metrics.

Accuracy: For both Java and Android, the majority of the participants *agree* or *strongly agree* that the generated comment is accurate, which consists of 106 of the 150 responses

	Java				Android			
Responses	**Ac**	**Ad**	**Co**	**Us**	**Ac**	**Ad**	**Co**	**Us**
1-Strongly Disagree	9	12	5	10	5	11	5	9
2-Disagree	8	17	17	17	5	13	8	6
3-Neutral	27	23	29	29	17	14	21	22
4-Agree	27	35	30	30	14	11	17	17
5-Strongly Agree	79	63	69	64	34	26	24	21
Total	150	150	150	150	75	75	75	75

Table 4.1: Human Judgements on the Generated Comments.
Ac: Accuracy; Ad: Adequacy; Co: Conciseness; Us: Usefulness

and 48 of the 75 responses respectively. Only 17 of the 150 Java responses and 10 of the 75 Android responses *disagree* or *strongly disagree.*

The main cause of the disagreement is that AutoComment fails to identify some sentences that contain an incorrect description of the code segment. AutoComment selects such sentences due to their high text similarity scores. While in general a sentence with a high text similarity score is more likely to an accurate description of the code, it is not always true. Some sentences are highly related to the code segment, but AutoComment fails to select them because the sentences use synonyms instead of the exact words used in the code.

In the future, we can improve the accuracy of the comment selection component by using *synonyms* [19] to help identify more related words between the code segment and comment, or using *term frequency-inverse document frequency* (tf-idf) to determine the importance of similar terms.

Adequacy: For Java, the majority of the participants *agree* or *strongly agree* that the generated comment contains adequate information, which consists of 98 of the 150 responses. Only 29 of the 150 Java responses *disagree* or *strongly disagree.* For Android, 37 of the 75 responses agree that the comments are adequate, which is one response short from achieving majority. A total of 24 of the 75 Android responses *disagree* or *strongly disagree.*

The main cause of disagreement on adequacy is user expectation. When we present a code segment with its surrounding code to help users understand the code, it is natural for them to think that the comment should be integrated with information from the surrounding context. In one user study question, the participant wrote, “Add path of the action event to the clipboard” and our tool generated “Use the StringSelection with the string and add it to the Clipboard.”. The participant was able to infer that the *string* is the *path of an action event* from the surrounding code.

Conciseness: For both Java and Android, the majority of the participants *agree* or *strongly agree* that the generated comment is concise and does not contain excessive information, which consists of 99 of the 150 responses and 41 of the 75 responses respectively. Only 22 of the 150 Java responses and 13 of the 75 Android responses *disagree* or *strongly disagree.*

The main cause of disagreement on conciseness is that the generated sentences contain overlapping content. AutoComment presents multiple sentences if they have the same text similarity score. Future work can address this issue by leveraging the text summarization techniques to detect overlapping content in sentences, and use advanced text similarity metrics described earlier to select the most relevant sentences.

Usefulness: For both Java and Android, the majority of the participants *agree* or *strongly agree* that the generated comment is useful, which consists of 94 of the 150 responses and 38 of the 75 responses respectively. Only 27 of the 150 Java responses and 15 of the 75 Android responses *disagree* or *strongly disagree.*

The main cause of disagreement on usefulness is that the code is easy-to-understand (so that no comment is needed to help comprehension), or the comment is too trivial. A comment is most useful if the code cannot be easily understood. To improve the usefulness of AutoComment-generated comments, we can design code and comment complexity metrics to filter out simple comments and comments for simple code segments in the future.

Execution Time: We executed AutoComment on an Intel Core i7-2600 CPU. The database generation for Java took 261 minutes, and the comment generation for the 16 Java projects took 612 minutes (10–384 minutes per project). The database generation of the Andriod database took 121 minutes, and the comment generation for the 7 Andriod projects took 51 minutes (4–22 minutes per project).

Chapter 5

Limitations and Future Work

Here we discuss the issues of AutoComment and propose solutions that can address the issues.

5.1 Comment Yield

AutoComment cannot generate a comment for a code segment if a Q&A site does not discuss it, or if the technique cannot detect the code-description mapping. AutoComment had generated a low number of comments for the evaluated projects even though we are able to extract 132,767 code-description mappings. Here are some of the main limitations:

First, the current implementation only accepts post answers that have the highest vote and only considers the description sentences immediately before the code segment, which limits the size of the databases. In the future, we can increase the size of the code-description mapping databases by including StackOverflow answers that do not have the highest vote count.

Second, the code clone detection (1) is not tolerant of statement reordering, and (2) cannot find clones that contain line additions in the StackOverflow code segment because we only allow line skipping on the target software. In the future, we can replace the code clone detection tool with one that can detect addition and reordering of lines to increase the number of code matches. For example, we can utilize an AST-based clone detection tool. Since the partial code segments are not compilable into a fully qualified AST tree, we can leverage previous work [15] to resolve the fully qualified names for the code elements.

Third, in order to expand the code-description mapping databases, it is possible to extract comments from data sources other than StackOverflow, such as the source code or existing API documentation. Extracting comments from the source code requires analysis to determine the lines of code that the comment is describing. Extracting comments from existing API documentation such as JavaDoc requires natural language processing techniques to combine the extracted sentences for a concise sentence.

Forth, AutoComment extracts databases of code-description mappings based on the tag of each post. It is possible for the Java database to miss some code-description mappings of Java if they are not tagged with `java` in StackOverflow. However, given the fact that StackOverflow is well maintained, the Java database should contain most of the Java code-description mappings from StackOverflow. We can potentially leverage previous techniques [20, 21, 22] to help classify the posts in StackOverflow to locate more relevant code-description mappings.

5.2 Comment Quality

Some generated comments are incorrect, contain overlapping information, or are too trivial at describing the code. It is challenging to analyze natural language sentences because they are highly unstructured. We can apply advanced NLP techniques such as semantic role labeling to analyze the semantics of the sentences, or typed dependencies to analyze the grammatical structure of the sentences. One of the possible directions is to design templates to extract certain sentence structures based on the grammatical structure. It allows one to restrict the types of sentences that we extract. For example, "Convert array of strings into a string". The sentence has a verb phrase "Convert ..." with a direct object "array", and a prepositional phrase "into ..." with an object "string". One can generalize this grammar structure to extract sentences that utilize different prepositional phrases. For example, the template can match against a wide variety of sentences such as "Convert array of strings **towards** a string", or "Convert array of strings **to** a string".

We can leverage information such as the author's post count to determine the quality of the extracted code-description mappings. It is also feasible to use advanced text similarity measures by incorporating synonyms and the importance of the similar words (e.g., tf-idf scores) to identify relevant sentences more accurately to generate more accurate comments.

5.3 User Study

Our current user study requests the participants to rank the quality of the comments based on the accuracy, adequacy, conciseness, and usefulness criteria. However, this is subjective and each participant has a different standard based on their programming background. In order to create a baseline, we need to show two comments to the participants and ask them to compare their relative quality. This can be achieved by asking the participants to compare between 1) a comment that is synthesized using AutoComment, and 2) a comment that is written by another participant. This way we can understand if the automatically generated comments are better compared to human written comments.

Chapter 6

Related Work

Much work for comment generation generates comments from the source code. Our work takes a different approach and attempts to mine comments from Question and Answer sites.

6.1 Automatic Comment Generation

Automatic comment generation generates comments automatically for certain code structures, such as failed test cases [23], exceptions [16], APIs [24], code changes [25] and function parameters [26]. Sridhara et al. proposed an approach to generate comments automatically from code for Java methods [4], high level actions within methods [5], and Java classes [27]. Other work generates comments for software concerns [28] and MPI programs [29]. However, these techniques do not solve the problem of grouping statements that perform *different* sub-actions into a high level action. Recently, Wang et al. [30] proposed a grouping strategy that segments method code into meaningful blocks. The grouping strategy can potentially improve the previous technique [5]. However, it is still difficult to generate comments for a group of statements with *different* sub-actions.

Different from previous work, AutoComment leverages a Q&A site—StackOverflow for automatic comment generation instead of source code. Our approach is not limited to the grouping strategy (as discussed in Section 1 and 1.1.1) because AutoComment can naturally group the statements based on the code segments from StackOverflow written by developers. Previous work by Sridhara et al. automatically generates high level actions within methods [5], but their technique works on statement sequences that are conditional blocks, perform similar actions, or follow specific templates. Our work can generate a high level comment for multiple statements that perform different actions.

6.2 Mining Descriptions for Code Artifact

One contribution of AutoComment is the databases of code-description mappings. Many studies mine descriptions or documentations for code artifacts from developers' communications, such as bug reports, forum posts and emails [31, 32, 33]. These studies focus on project specific descriptions. For example, they extract descriptions for Eclipse code artifacts from the mailing list and bug tracking system of Eclipse. Thus, such descriptions are more likely to benefit Eclipse.

Different from previous work, the code-description mappings discovered by AutoComment are general for each domain, e.g., the Java database should benefit all Java projects. In addition, these studies focus on method level descriptions, and they aim at documentations instead of more concise comments. Our work is not limited to methods, and we adapt NLP techniques to improve the comment quality. In addition, previous work leverages heuristics (e.g., text similarity) to link descriptions and code. Our approach combines clone code detection and heuristics to improve the accuracy. Some work helps improve code extraction from unstructured data, such as emails and documents [34, 35]. In the future, AutoComment can leverage these techniques to extract more code-description mappings from emails and documents, not only from Q&A sites.

6.3 Code Clone Detection

There are three kinds of code clone detection techniques: token-based [14, 36, 37], AST-based [38, 39], and semantics-based [40]. As most of the code segments from StackOverflow are uncompilable, AST-based and semantics-based techniques are not suitable for use. AutoComment leverages a token-based tool, SIM [14], to detect matched code segments because it is an open source tool and can be extended to support other programming languages.

6.4 Automatic Code Generation

Some researchers work on generating code from natural language descriptions [41, 42, 43]. These studies point out a valuable application of our techniques and results. Our code-description mapping databases can potentially be used to assist in automatically generating code from natural language descriptions.

Chapter 7

Conclusions

We proposed a new, general approach to mine Question and Answer site for automatic comment generation. Our tool generated 102 comments from the 23 evaluated projects. The number of generated comments is still low, but we had identified several directions to tackle the issue. We leverage descriptions that developers use to describe code segments in StackOverflow. The generated comments are accurate and useful in helping developers understand the code segments as confirmed by the conducted user study. Also, the generated comments can contain information that is not explicitly in the code segment, which is a significant advantage of the proposed approach over the previous techniques on automated comment generation.

In the future, we want to focus on improving both the yield and quality of the generated comment. To improve the yield, we can expand the size of the code-description mapping database by including StackOverflow answers that do not have the highest vote count. Another way is to use a code clone detection tool that can detect addition and reordering of lines to increase the number of code matches. To improve the quality, we can apply advanced NLP techniques such as semantic role labeling to analyze the semantics of the sentences, or typed dependencies to analyze the grammatical structure of the sentences. In addition, the code-description mapping databases can be leveraged for other purposes such as program synthesis, which generates code automatically from natural language descriptions.

APPENDICES

Appendix A

Ethics Clearance Approval

From:	ORE Ethics Application System <OHRAC@uwaterloo.ca>
To:	lintan@uwaterloo.ca
CC:	e32wong@uwaterloo.ca, j223yang@uwaterloo.ca
Date:	Mon, Feb 11, 2013 at 3:21 PM
Subject:	Ethics Clearance (ORE # 18754)

Dear Researcher:
The recommended revisions/additional information requested in the ethics review of your ORE application:

Title: AutoComment: Mining Developer Forums for Automatic Comment Generation
ORE #: 18754
Faculty Supervisor: Lin Tan (lintan@uwaterloo.ca)
Student Investigator: Edmund Wong (e32wong@uwaterloo.ca)
Student Investigator: Jinqiu Yang (j223yang@uwaterloo.ca)

have been reviewed and are considered acceptable. As a result, your application now has received full ethics clearance.

A signed copy of the Notification of Full Ethics Clearance will be sent to the Principal Investigator or Faculty Supervisor in the case of student research.

****
Note 1: This ethics clearance from the Office of Research Ethics (ORE) is valid for one year from the date shown on the certificate and is renewable annually, for four consecutive years. Renewal is through completion and ethics clearance of the Annual Progress Report for Continuing Research (ORE Form 105). A new ORE Form 101 application must be submitted for a project continuing beyond five years.

Note 2: This project must be conducted according to the application description and revised materials for which ethics clearance has been granted. All subsequent modifications to the project also must receive prior ethics clearance (i.e., Request for Ethics Clearance of a Modification, ORE Form 104) through the Office of Research Ethics and must not begin until notification has been received by the investigators.

Note 3: Researchers must submit a Progress Report on Continuing Human Research Projects (ORE Form 105) annually for all ongoing research projects or on the completion of the project. The Office of Research Ethics sends the ORE Form 105 for a project to the Principal Investigator or Faculty Supervisor for completion. If ethics clearance of an ongoing project is not renewed and consequently expires, the Office of Research Ethics may be obliged to notify Research Finance for their action in accordance with university and funding agency regulations.

Note 4: Any unanticipated event involving a participant that adversely affected the participant(s) must be reported immediately (i.e., within 1 business day of becoming aware of the event) to the ORE using ORE Form 106.

Best wishes for success with this study.

Susanne Santi, M. Math.,
Senior Manager
Office of Research Ethics
NH 1027
519.888.4567 x 37163
ssanti@uwaterloo.ca

Appendix B

Sample User Study Questionnaire

```
=================================================================
Q1: Please read the marked code that is highlighted between ####
=================================================================
  public String getToolTipText(MouseEvent e) {
############### Marked code starts here ###############
    java.awt.Point p = e.getPoint();
    int rowIndex = rowAtPoint(p);
    int colIndex = columnAtPoint(p);
############### Marked code ends here   ###############
    if (rowIndex < 0 || colIndex < 0) {
      return null;
    }
    Object o = getModel().getValueAt(convertRowIndexToModel(rowIndex),
        convertColumnIndexToModel(colIndex));
    if (o == null) {
      return null;
    } else if (o instanceof IconLabel) {
      return ((IconLabel) o).getTooltip();
    } else if (o instanceof Date) {
      return FORMATTER.format((Date) o);
    } else {
      return o.toString();
    }
  }

Q1.1: Please write your comment that describes the functionality of the marked
   code segment:

       [  ]  I do not have an answer
```

```
Q1.2: Please read this comment:
"Find on which row and column the mouse is."

  (a): The comment above is accurate in describing the marked code:

       (Strongly Disagree)    1    2    3    4    5    (Strongly Agree)

  (b): The comment above is adequate (i.e., is not missing information)
       in describing the marked code:

       (Strongly Disagree)    1    2    3    4    5    (Strongly Agree)

  (c): The comment above is concise in describing the marked code:

       (Strongly Disagree)    1    2    3    4    5    (Strongly Agree)

  (d): The comment above helps me understand the highlighted code:

       (Strongly Disagree)    1    2    3    4    5    (Strongly Agree)
```

References

[1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Proceedings of annual Reliability and Maintainability Symposium*, pages 235–241, 2002.

[2] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*iComment: Bugs or Bad Comments?*/. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[5] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically Detecting and Describing High Level Actions within Methods. In *Proceedings of 33rd International Conference on Software Engineering*, pages 101–110, 2011.

[6] E. Wong, Jinqiu Yang, and Lin Tan. AutoComment: Mining Question and Answer Sites for Automatic Comment Generation. In *Proceedings of the 28th International Conference on Automated Software Engineering*, pages 562–567, 2013.

[7] StackOverflow http://stackoverflow.com, 2014.

[8] C. Parnin, C. Treude, G. Lars, and M. D. Storey. Crowd documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[9] S. Subramanian and R. Holmes. Making Sense of Online Code Snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 85–88, 2013.

[10] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, pages 173–180, 2003.

[11] D. Klein and C. D. Manning. Accurate Unlexicalized Parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, pages 423–430, 2003.

[12] B. Santorini. Part-Of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision, 2nd Printing). Technical report, Department of Linguistics, University of Pennsylvania, 1990.

[13] A. Curzan and M.P. Adams. *How English Works: A Linguistic Introduction.* Pearson Education/Longman, 2012.

[14] Dick Grune. The software and text similarity tester SIM. `http://dickgrune.com/Programs/similarity_tester/`, 2014.

[15] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of the International Conference on Software Engineering*, 2014.

[16] R. P. L. Buse and W. R. Weimer. Automatic Documentation Inference for Exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[17] G. A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, November 1995.

[18] R. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[19] J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 161–170, 2012.

[20] S. Gottipati, D. Lo, and J. Jiang. Finding Relevant Answers in Software Forums. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 323–332, 2011.

[21] M. Allamanis and C. Sutton. Why, When, and What: Analyzing Stack Overflow Questions by Topic, Type, and Code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 53–56, 2013.

[22] S. Wang, D. Lo, and L. Jiang. An Empirical Study on Developer Interactions in Stack-Overflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024, 2013.

[23] S. Zhang, C. Zhang, and M. D. Ernst. Automated Documentation Inference to Explain Failed Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 63–72, 2011.

[24] F. Long, X. Wang, and Y. Cai. API Hyperlinking via Structural Overlap. In *International Symposium on the Foundations of Software Engineering*, pages 203–212, 2009.

[25] R. P. L. Buse and W. Weimer. Automatically Documenting Program Changes. In *International Conference on Automated Software Engineering*, pages 33–42, 2010.

[26] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating Parameter Comments and Integrating with Method Summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80, 2011.

[27] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 23–32, 2013.

[28] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating Natural Language Summaries for Crosscutting Source Code Concerns. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pages 103–112, 2011.

[29] Suparna Gundagathi Manjunath. Towards Comment Generation for MPI Programs. Master's thesis, University of Delaware, 2011.

[30] X. Wang, L. Pollock, and K. Vijay-Shanker. Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, pages 35 –44, 2011.

[31] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora. Mining Source Code Descriptions from Developer Communications. In *Proceedings of the International Conference on Program Comprehension*, pages 63–72, 2012.

[32] B. Dagenais and M. P. Robillard. Recovering Traceability Links between an API and its Learning Resources. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 47–57, 2012.

[33] J. Kim, S. Lee, S. Hwang, and S. Kim. Enriching Documents with Examples: A Corpus Mining Approach. *Proceedings of the ACM Transactions on Information Systems*, 31(1):1:1–1:27, January 2013.

[34] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting Source Code from E-Mails. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, pages 24 –33, 2010.

[35] Nicolas Bettenburg, Bram Adams, Ahmed E. Hassan, and Michel Smidt. A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 185–188, 2011.

[36] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *Proceedings of IEEE Transactions on Software Engineering*, 32(3):176 – 192, 2006.

[37] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Proceedings of the IEEE Transactions on Software Engineering*, 28(7):654 – 670, 2002.

[38] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105, 2007.

[39] I.D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, 2004.

[40] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001.

[41] Samuel T. King Anthony Cozzie. Macho: Writing Programs from Natural Language and Examples. Technical report, 2012.

[42] Anthony Cozzie, Murph Finnicum, and Samuel T. King. Macho: Programming with Man Pages. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, pages 7–7, 2011.

[43] Anthony Cozzie. *Detecting and Combining Programming Patterns*. PhD thesis, University of Illinois, 2011.