AutoComment: Mining Question and Answer Sites for Automatic Comment Generation

Edmund Wong, Jinqiu Yang, and Lin Tan
University of Waterloo, Waterloo, Ontario, Canada
{e32wong, j223yang, lintan}@uwaterloo.ca

***Abstract*—Code comments improve software maintainability. To address the comment scarcity issue, we propose a new 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 Java and Android tagged Q&A posts to extract 132,767 code-description mappings, which help AutoComment to generate 102 comments automatically for 23 Java and Android projects. 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.**

***Index Terms*—automated comment generation; documentation; program comprehension; natural language processing for software engineering**

I. Introduction

Code commenting has been an integral part of software development. It has been a standard practice in the industry. Comments improve software maintainability [1] through helping developers understand code. Despite the need and importance of commenting code, many code bases do not contain adequate comments [2].

Sridhara et al. automatically generate comments for Java methods [3] and groups of statements [4] from source code. 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 [3], or groups of method calls [4]). Second, performance depends upon high-quality identifier names and method signatures. For example, when grouping method calls, this technique requires that all method names contain the same verb [4]. If identifiers and methods are poorly named, the approach may fail to generate accurate comments or any comments at all.

We propose a new approach to generate comments automatically to address the above limitations. We observe that Question and Answer (Q&A) sites such as StackOverflow [5] contain code descriptions written by developers that can be used for automatic comment generation. Specifically, StackOverflow 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 question asked "how to open the find type dialog programmatically in Eclipse". In the answer, a user provided a Java code segment that performs the 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 segment. We refer to the code segment and description as a *code-description mapping*. If a similar piece of code appears in a software project, then the corresponding description of the mapping can be an explanatory comment for the code in the software project.

StackOverflow [5] contains a wealth of information, which makes it a feasible source for extracting code-description mappings for automated comment generation. It contains 5,509,302 posts as of August 2013. In addition, at least 49% of the Java and Android classes in StackOverflow have at least one code example in the accepted answer [6]. Android code segments have a mean size of 16.4 lines of code (LOC) and a median of 9 LOC [7].

The idea is to generate comments automatically by mining Q&A sites for code-description mappings. One key benefit of this approach 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 and method names).

This paper makes the following contributions:

- We propose a new approach, *AutoComment*, to generate code comments automatically by analyzing Q&A sites.
- We conducted a preliminary evaluation of AutoComment on 23 projects to generate 102 comments automatically. The user study results show 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 for generating 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.

II. Examples and Challenges

We present two examples illustrating how AutoComment generates comments automatically. We describe the challenges, summarize our solutions, and highlight the unique benefits of AutoComment.

A. Example One

Figure 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| }
```

Fig. 1. Code from Java project Jajuk

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

Find on which row and column the mouse is.

Our user study shows that users consider this comment accurate and useful in helping them understand the code. The previous technique by Sridhara et al. [4] would not generate a comment for this example because the three method names in Line 2–4 share no common verb. Figure 2 shows the StackOverflow post that AutoComment leverages to generate the comment. It shows the title of the post, the code segment, and one paragraph before the code segment in the answer.

`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:*

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

Fig. 2. StackOverflow Post #10854831

Challenges in Comment Selection: Figure 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 segment, and AutoComment needs to select this relevant sentence for use as a comment.

AutoComment uses two techniques to address this challenge. First, many sentences ask and answer how to troubleshoot their code, and they often do not describe the code segment. 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 sentences that imply troubleshooting based on keyword filtering (Section III-B). Second, AutoComment leverages the *text similarity* between each sentence and the code segment to identify the most relevant sentences (Section III-E). In Figure 2, the shared words between the text and code are in bold (**row** and **column**).

Challenges in Description Refinement: The sentences from question titles and answers are often in a question form (e.g., "How 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 techniques to refine sentences. For example, AutoComment removes "From the event, you can" and "using" from the last sentence in Figure 2. One of the used techniques extracts a subtree that contains a verb phrase and a noun phrase from the parse tree of a sentence (Section III-B).

B. Example Two

Figure 3 shows a code segment from the Android project Barcode Scanner.

```
1| private static Bitmap toBitmap(LuminanceSource source, int
   |     [] pixels) {
2|   int width = source.getWidth();
3|   int height = source.getHeight();
4|   Bitmap bitmap = Bitmap.createBitmap(width, height,
   |       Bitmap.Config.ARGB_8888);
5|   bitmap.setPixels(pixels, 0, width, 0, 0, width, height);
6|   return bitmap;
7| }
```

Fig. 3. Code from Android project Barcode Scanner

AutoComment generates the following comment for the lines highlighted in grey by leveraging the StackOverflow post #4665122 (not shown due to space constraints):

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

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 group the three statements into a semantic unit for comment generation because the StackOverflow code segments had already been grouped in the post. Such grouping does not rely on the quality of the method names or the structure of the methods, which is different from previous work [4].

III. AutoComment Design

Figure 4 shows the overview of AutoComment. AutoComment takes as input (1) a StackOverflow data dump containing information of all posts, and (2) source code of the target software. The output is a list of code segments and the corresponding generated comments.

AutoComment consists of two major components. The first component generates databases of code-description mappings (Section III-A) and leverages natural language processing techniques to refine the descriptions (Section III-B). The second component generates comments for the target software. It applies code clone detection technique to identify matched code between the databases and the target software (Section III-C),

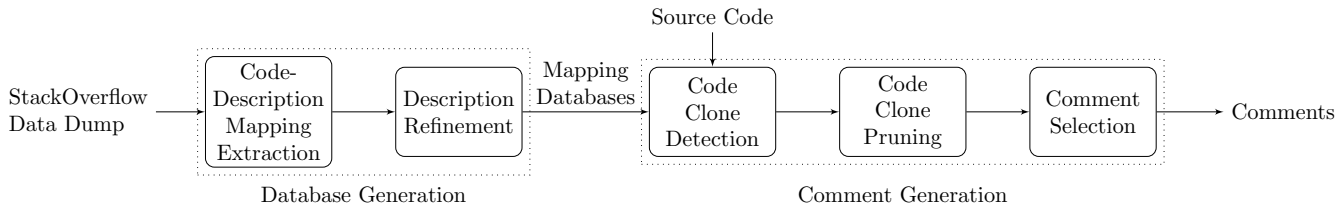

Fig. 4. Overview of AutoComment

TABLE I
LIST OF TERMS FOR SENTENCE FILTERING

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

prunes out bad matches (Section III-D), and selects the best comment for the matched code (Section III-E).

A. Code-Description Mapping Extraction

To build databases of code-description mappings, we choose a programming Q&A site called StackOverflow [5] as the data source. We build the Java database using Java questions (tagged with `java`) and the Android database using Android questions (tagged with `android`).

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 StackOverflow's voting system. Specifically, AutoComment only keeps questions with a positive 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. 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.

B. Description Refinement

To improve the quality of description sentences extracted from StackOverflow, AutoComment leverages natural language processing techniques to perform refinements.

Description Filtering: Sentences that ask and answer how to troubleshoot code often do not describe the code segment, e.g., "Android: problem retrieving bitmap from database". AutoComment filters out such sentences based on the manually collected terms in Table I.

Main Sub-Tree Extraction: Sentences that are in a question form (e.g., "How to...") or contain personal pronouns (e.g., "you") are not suitable as comments. Therefore, we adapt NLP techniques to convert sentences from a question form to a statement form by extracting the main sub-tree of a sentence in the following three steps.

Step one generates a parse tree from the input sentence using Stanford CoreNLP [8] (v1.3.4). AutoComment first uses the part-of-speech (POS) tagger to label each word of a sentence, then uses the parser to generate the parse tree. CoreNLP does fall short on interpreting certain technical terms because it was trained on well written text such as the Wall Street Journal. However, it is robust at parsing sentences and works well for our experiments.

Step two extracts the main sub-tree(s) 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 1 and Equation 2) in Stanford's Tregex [9] format to extract the main sub-tree(s) of a parse tree.

$$VP << (NP < /NN.?/) < /VB.?/ \quad (1)$$

$$NP\,!< PRP\,[<< VP\,|\,\$\,VP] \quad (2)$$

The two patterns ensure the VP is not a personal pronoun (PRP) because such words contribute no value in a code comment. Penn Treebank tag guideline [10] defines PRP to include personal pronouns proper ("I", "you", "he", etc.), reflexive pronouns ending in *-self* or *-selves* and nominal possessive pronouns ("mine", "yours", "his", etc.). The first pattern extracts a VP followed by an NP, and the second pattern extracts an NP followed by a VP.

Step three merges the extracted sub-trees (if there are more than one from step two) into a single sub-tree, and then converts the merged sub-tree into a sentence by retrieving all the leave nodes from the tree structure. To merge the extracted sub-trees, AutoComment invokes 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*.

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 but skip comments with java

The seven coordinating conjunctions are "for", "and", "nor", "but", "or", "yet", and "so" [11]. The CC words "but" and "yet" imply a contrasting meaning. Therefore, AutoComment removes the clause after the CC word "but" and "yet".

Number Removal: AutoComment removes numerical numbers from a sentence to make it more general by detecting cardinal number (CD) POS tags, which represent numeric words such as "three" and "3".

C. Code Clone Detection

AutoComment extends a token-based clone detection tool SIM [12] to detect similar code segments between the code-description databases and the target software. Parse-tree-based code clone detection tools are not directly applicable because code segments from StackOverflow are often not compilable.

SIM tokenizes the 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 detect code clones that contain line additions by allowing the additional lines from the target software to be skipped (lines from the StackOverflow code segment are not skipped).

We use the following thresholds for each match: the minimum number of tokens that have to match consecutively is 20, and the maximum number of lines that can be skipped is 4. In the future, we would like to study the impact of these thresholds on comment generation.

D. Code Clone Pruning

The output of the code clone detection tool consists of pairs of code segments that have a similar syntactical structure. However, it is necessary to ensure the code segments have a high-level of semantic matching.

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 software project, AutoComment prunes out such pairs of code segments. In the future, we would like to scale the value according to project size.

Line Percentage Matching: For each StackOverflow code segment, the higher proportion of lines that are matched, 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.

Particularly, 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 (i.e., `add`, `remove`, `put`, `post`, `get`, `set`, `read`, `write`, `delete`, `close`, `exit` and `hashCode`) 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.

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

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 }
```

Fig. 5. Example of a piece of template code. StackOverflow post #5035894.

Removal of Repetitive Method Calls: If a matched code segment in the target software only contains repetitive method calls on the same method (three or more times), it is performing a similar operation repetitively with different parameters. Since the value of the parameters impacts the functionality of the code segment in the target software, and the code clone detection tool does not perform exact matching on the value of the parameters (Section III-C), such matches are removed.

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 5 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 StackOverflow code segments that contain a pair of curly braces with no statements in it as template code and remove them.

Other Filters: AutoComment requires the matching of at least one line that contains a method call. In addition, it filters out matches that contain the term "Exception" because exception code is inherently different from the main flow code. Lastly, it prunes out long code matches (over fifteen lines of code) because StackOverflow is unlikely to contain detailed enough descriptions.

E. Comment Selection

For each remaining match, one or more description sentences can remain as a comment candidate. If the code from the target software matches with multiple StackOverflow code segments, AutoComment includes all available description sentences of each StackOverflow code segment as a candidate. It then selects the comment candidates that best describe the matched code segment in the target software.

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

expressions (combined with camel cases) and removes such sentences.

Text Similarity: To select the best description sentences from the remaining sentences, AutoComment measures the *text similarity* between each remaining description sentence and the code segment in the target software.

There are three steps to measure text similarity. First, it converts the code and description sentences to a set of tokens. A token is a consecutive sequence of at least three characters (alphabets and numbers) because short tokens such as "is" often have no semantic meaning. The dot operator bridges multiple tokens together. For example, it converts `obj.method()` to `obj.method` instead of two tokens, `obj` and `method`. This is because `obj.method()` is an atomic operation and should only contribute to the text similarity once. Second, it lemmatizes [13] tokens to their base forms, e.g., converting "takes" to "take". Then it removes duplicate tokens and stop words (i.e., "new", "the", "and", "but", "for", and "you"). Third, 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 that achieve the highest text similarity. If multiple sentences have the same highest text similarity, it combines all of them as the generated comment.

IV. Experimental Methods

We conducted a user study similar to that of Sridhara et al. [4] to answer 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?

RQ2 is a new research question that Sridhara et al. [4] did not evaluate. It is an important question because a comment can be accurate, adequate, and concise, but does not help developers understand the code, e.g., if the comment is a simple paraphrase of the code.

A. Evaluated Projects and Databases

We apply AutoComment to extract two databases of code-description mappings. The extracted Java and Android databases contain 87,785 and 44,982 mappings respectively. We apply the Java database on 16 Java projects and the Android database on 7 Android projects. Table II shows the number of lines of code that each project contains. AutoComment generated a total number of 102 comments for the 23 projects.

B. User Study

We conducted a user study to evaluate the comments generated by AutoComment. The evaluator group included 14 graduate students and 1 undergraduate student, all of whom have industrial experience in Java programming (ranging from 2–10 years). We provided each user with a questionnaire of 15 randomly selected comments (10 from Java projects and 5 from Android projects) to evaluate.

The user study evaluation has two steps. First, we show users a matched code segment with its surrounding code and ask users to write a comment to describe the matched code segment. Second, we provide users with the generated comment and ask them to rate the comment on accuracy, adequacy, conciseness, and usefulness using the five-point Likert scale.

TABLE II
Evaluated Open-Source Projects

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

	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

V. Preliminary Results

Table III shows the human judgement from the participants for the 102 comments. The responses show that the majority of the users agree that the generated comments are accurate, adequate, concise, and useful in helping them understand the code segments.

In terms of **accuracy**, the main cause of disagreement is the failure at identifying comments that contain an incorrect description of the code segment. The reason is that text similarity does not guarantee a sentence to be semantically correct. In terms of **adequacy**, the main cause of disagreement is the user expectation. When we present a code segment with its surrounding code to help users understand the code segment, they often think that the comment should also explain the surrounding context. For 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 inferred that the *string* is the *path of an action event* from the surrounding code and rated the generated comment inadequate. In terms of **conciseness**, the main cause of disagreement is that the generated sentences can be too long, wordy or contain overlapping content. The reason is that AutoComment extracted the sentences from a Q&A site and used only basic NLP techniques to refine and select them. In terms of **usefulness**, the main cause of disagreement is on code that is easy-to-understand (so that no comment is needed to help comprehension), or the comment is too trivial. We discuss possible solutions to address the above disagreements in Section VII.

VI. Related Work

Much work automatically generates comments from source code. Some generates comments automatically for certain code structures, such as failed test cases [14], exceptions [15], code changes [16] and method parameters [17]. Other approaches generate comments automatically for software concerns [18], MPI methods [19], Java classes [20], Java methods [3] and high-level actions within methods [4]. In addition, many studies mine descriptions or documentation for code artifacts from developers' communications, such as bug reports, forum posts and emails [21], [22], [23]. Previous work by Sridhara et al. automatically generates high-level actions within methods [4], but their technique focuses 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.

VII. Discussion and Future Work

We proposed a new approach to mine Q&A sites for automatic comment generation. The generated comments can contain information that is not explicitly in the code segment, which is a significant advantage over the previous techniques on automated comment generation. The code-description databases can be leveraged for other purposes such as automatically generating code from natural language descriptions.

Some generated comments are incorrect, contain overlapping information, or are too trivial at describing the code. We can apply NLP techniques such as semantic role labeling to analyze the semantics of the sentences. This will help AutoComment improve comment refinement and selection.

If a Q&A site does not discuss a code segment, AutoComment cannot generate a comment for it. AutoComment had generated a low number of comments for the evaluated projects mainly for the following reasons. First, the current implementation only accepts posts that have the highest vote and only considers the description sentences immediately before the code segment, which limit the size of the databases. 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 allow line skipping on the target software only. 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. Another improvement is to replace the code clone detection tool with one that can detect addition and reordering of lines to increase the number of code matches.

Acknowledgments

We thank Theo Pan and Javier Munster for their help with the experiments, and Yuancheng Tu for her valuable feedback regarding natural language processing. This work is supported by the National Science and Engineering Research Council of Canada and a Google gift grant.

Availability: The extracted code-description databases and generated comments are available at `http://asset.uwaterloo.ca/AutoComment/`.

References

[1] K. Aggarwal, Y. Singh, and J. Chhabra, "An Integrated Measure of Software Maintainability," in *Proc. Ann. Reliability and Maintainability Symp.*, pp. 235–241, 2002.
[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A Study of the Documentation Essential to Software Maintenance," SIGDOC, 2005.
[3] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," ASE, 2010.
[4] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically Detecting and Describing High Level Actions within Methods," in *ICSE*, 2011.
[5] "StackOverflow." Available at http://stackoverflow.com/, 2013.
[6] C. Parnin, C. Treude, L. Grammel, and M.-A. 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.
[7] S. Subramanian and R. Holmes, "Making Sense of Online Code Snippets," MSR, 2013.
[8] "Stanford CoreNLP." Available at http://nlp.stanford.edu/software/corenlp.shtml, 2013.
[9] "Tregex." Available at http://nlp.stanford.edu/software/tregex.shtml, 2013.
[10] B. Santorini, "Part-Of-Speech Tagging Guidelines for the Penn Treebank Project (3rd revision, 2nd printing)," tech. rep., Department of Linguistics, University of Pennsylvania, 1990.
[11] A. Curzan and M. Adams, *How English Works: A Linguistic Introduction*. Pearson Education/Longman, 2012.
[12] D. Grune, "The software and text similarity tester SIM," 2012.
[13] G. A. Miller, "WordNet: A Lexical Database for English," 1995.
[14] S. Zhang, C. Zhang, and M. Ernst, "Automated Documentation Inference to Explain Failed Tests," ASE, 2011.
[15] R. P. Buse and W. R. Weimer, "Automatic Documentation Inference for Exceptions," ISSTA, 2008.
[16] R. P. Buse and W. R. Weimer, "Automatically Documenting Program Changes," ASE, 2010.
[17] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating Parameter Comments and Integrating with Method Summaries," ICPC, 2011.
[18] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating Natural Language Summaries for Crosscutting Source Code Concerns," ICSM, 2011.
[19] S. G. Manjunath, "Towards Comment Generation for MPI Programs," Master Thesis, 2011.
[20] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," ICPC, 2013.
[21] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining Source Code Descriptions from Developer Communications," ICPC, 2012.
[22] B. Dagenais and M. Robillard, "Recovering Traceability Links between an API and Its Learning Resources," ICSE, 2012.
[23] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching Documents with Examples: A Corpus Mining Approach," *ACM Trans. Inf. Syst.*, vol. 31, pp. 1:1–1:27, Jan. 2013.