ISSN: 2321-7782 (Online)
Volume 4, Issue 1, January 2016
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

J-Summarizer

Suhas C. Mahakalkar[1]
Dept of Computer Engineering,
Savitribai Phule University, Pune, India

Prof. A. D. Gujar[2]
Dept of Computer Engineering,
Savitribai Phule University, Pune, India

Abstract: Consistent, correct and complete documentation of a software system is an important vehicle for the maintainer to gain its understanding, to ease its learning and/or relearning processes, and to make the system more maintainable. It is extremely important for achieving insight and visibility into the processes, important for their meaningful process measurement and thereby pivotal for achieving high process maturity.

In this case, a crucial question arises: "How much and what context of documentation is enough?" In this article, we present the results of a survey of software maintainers and developers to try to establish what documentation artifacts are the most useful to them.

Key words: Source code summarization, automatic documentation, program comprehension, Software Engineering.

I. INTRODUCTION

Programmers rely on good software documentation. [1], [2]. Software documentation exists to help software maintainers understand a system and its processes [3]. However, in practice, source code documentation is expensive to produce, resulting in often incomplete and low quality documentation [4]. Additionally, as software undergoes maintenance and updates, this high cost results in software documentation becoming outdated as software is update [5], [6]. Poor software documentation can lead to a software system rapidly degrading in quality and usability [3].

Writing documentation is difficult because it requires programmers to impart knowledge to other programmers. This difficulty creates a separation between the authors and readers of documentation. Authors are the initial developers of source code. Readers are any person attempting to understand source code for usage or maintenance after the fact. Authors typically write summaries of source code in order to communicate to readers. Authors often use document generation tools in order to communicate the readers in a standardized fashion. Tools such as Doxygen1 and JavaDoc2 have helped source code documentation by standardizing format and presentation of source code documentation. However, these tools still rely on manual author input. This means authors still face the cost of documenting source code if they wish to communicate to the readers.

Automatic source code summarization has begun to emerge as a means of helping authors expedite the documentation process [7], [8], [9], [10], [11], [12], [13]. These tools automatically provide a wide variety summarization techniques and outputs. For example, Haiduc et al. [7] and Rodeghero et al. [11] provide a list of keywords that describe the method. These keywords are acquired using a vectorspace model to determine importance. By contrast, work by Sridhara et al. [12] and my own work[8] generate natural language sentences that summarize methods. These tools use natural language generation templates to create human readable sentences summarizing methods.

In order for automatic summarization to mature, the field needs to better understand documentation quality. Developers of these tools must understand what makes a good summary. They must also understand what types of information are important. Additionally, developers need to understand if some information is irrelevant or misleading, to avoid including such information

in their tools. In this paper, I propose new ideas and directions of research and present early findings into understanding how to improve automatic source code summarization.

II. PROBLEM

The key problem facing automatic source code summarization is that there is no agreed upon or well-defined standard of "good" documentation. While research exploring measurement of documentation quality exists [9], [11], much of this research relies on assumptions that, in some cases, have not been verified in the literature empirically. It is unclear what a good source code summary entails. Further, it may be that a high quality manually written source code summary will be different than a high quality automatically generated summary. Developers of source code summarization tools need to understand what a "good" summary is before they can develop tools that create them. In this paper, I propose three specific questions related to understanding what "good" documentation looks like, both in manually written and automatically generated summarizations and automatic documentation.

By answering these questions, designers of documentation tools can better understand what the goals of automatic software documentation should be. As automatic summarization tools become more sophisticated, it will be important to direct research towards areas more likely to produce better summaries. While automatic source code summarization is still in its infancy, knowing what goals to strive for will direct early research more productively. Further, understanding what "good" documentation is with respect to automatic source code summarization will illuminate what manually written "good" documentation should include.

III. PROPOSED WORK

In this section, I state three research questions. I believe the literature and practice of source code summarization will be improved by answering these questions. Additionally, I describe work, both completed and ongoing, that seeks to answer these questions,

A. *Research Questions*

My research seeks to answer the following three research questions:

RQ1 How similar should the text in summaries be to text and keywords in the source code?

RQ2 To what degree should contextual information about code be included in summaries of that code?

RQ3 To what extent can the file structure of source code affect the quality of documentation?

The key theme that connects all of these research questions is the need to understand what makes a "good" source code comment. Automatic source code summarization tools must be built with the understanding what information can be beneficial. *RQ1* answers an important question about source code summary and comment quality. While existing literature assumes that having documentation be textually or semantically similar to source code improves comment and summary quality, this assumption had not been experimentally tested. My work [8] tests the veracity of this assumption. *RQ2* presents the idea of source code context. Context Previous studies show that programmers benefit from understanding source code context [9]. I propose studying whether using techniques to organize source code documentation topically results in a structure similar to the source code itself. If the structures are similar, than source code hierarchy can be leveraged to produce more informed summaries

B. *RQ1: Summary Similarity*

It has been assumed that source code summaries should be similar to the code being summarized [6], [7]. The literature describes different types of similarity used. Two of these types of similarity are textual similarity and semantic similarity.

Textual similarity refers to how many words or word roots two sets of text have in common. Semantic similarity considers how similar the meanings of two sets of words are.

I conducted a study to among the IT Professionals, to determine how textually and semantically similar author-written summaries and reader written summaries of source code are to each other, as well as the source code being summarized. These two algorithms use the hierarchical knowledge base WordNet [24], which organizes words into nested sets of synonyms.

C. *RQ2: Context*

In this section, I will discuss the effects of source code context on summarization quality. I created an automatic source code summarization tool, called SumSlice. This tool generated natural language summaries to summarize Java methods, and was augmented by including method context. I compared SumSlice to an existing approach that also created natural language summaries for Java methods created by Sridhara et al. [12].

Sumslice, in detail, works by summarizing Java methods by considering a method's signature, its context, and its usage. Using SWUM, developed by Hill et al. [5], [6], SumSlice extracts keywords from the signature of a method m to form a subject and verb phrase that describe what task m performs. SumSlice then goes to the call graph of the Java project to find all methods that call m and are called by m. These methods form the context. The most important methods are determined by sorting the calling and called methods by their PageRank score with respect to the entire project. The two most important called and the two most important calling methods are included in the method summary. These methods also have their signatures used by SWUM to create a short phrase describing their purpose. Additionally, SumSlice finds a usage example of m from most important (according to PageRank) calling method to include in the summary.

D. *RQ3: Source Code Structure*

I propose an experiment to determine if source code structure is similar to documentation structure, and if the similarity of these structures affects documentation quality. Source code structure describes how sections of source code interact, including class depencies, call graphs, etc. We can also describe source code structure with respect to file structure. For example, in Java, source code is broken into methods. Each class typically contains several methods. Packages can contain classes as well as other nested packages. All the packages combined form the source code project. Two methods in the same class are usually more closely related than two methods in different packages.

I used our approach to generated a hierarchical model of source code keywords. I generated this hierarchy using a classification algorithm created by Weninger et al. [8]. This algorithm modified the existing call graph into a hierarchy, where more general methods focused higher in the tree, and more specific methods that were at the end of call trees formed the leaves.

IV. Related Work

The findings that result from this proposal would most directly affect automatic source code summarization. This paper cites several source code summarization tools. Haiduc et al. [7] used a vector space model to summarize methods. In this approach, methods were treated as documents, with keywords in the method being the terms. The importance of each keyword in each method was identified using term frequency/inverse document frequency (tf/idf). Eddy et al. [3] conducted a later study independently verifying that keywords selected by Haiduc et al.'s approach accurately summarized Java methods. Rodeghero et al. [11] modified Haiduc's algorithm by weighting tf/idf scores for keywords based on their immediate surroundings. Rodeghero et al based these weights on an eye-tracking study which found that method signature and method calls were more important to programmers, while control flow statements were less important.

Some source code summarization tools automatically generate natural language summaries. Sridhara et al. [12] developed an approach that selects important lines of code from a Java method and translates them into natural language summaries. The lines of code are translated by interpreting keywords using a Software Word-Usage Model (SWUM), developed by Hill et al.

[5], [6], which can infer parts of speech for keywords in identifier names. Moreno et al. [10] used knowledge of method stereotypes to create natural language summaries for Java classes.

V. CONCLUSION

In this paper, I have defined and discussed three important research questions related to source code summarization. Those three questions examine what type of content or information should be considered in order to generate "good" AUTOMATIC SUMMARIES.

References

1. A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in Proceedings of DocEng '02.
2. A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in Proceedings of VLHCC '04. Washington, DC, USA:
3. M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," Empirical Softw. Engg., vol. 10, no. 1, pp. 31–55, Jan. 2005.
4. S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in Proceedings of SIGDOC '05. New York, NY, USA: ACM, 2005, pp. 68–75.
5. T. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: the state of the practice," Software, IEEE, vol. 20, no. 6, pp. 35–39, Nov 2003.
6. W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," Journal of Systems and Software, vol. 85, no. 10, pp. 2293 – 2304, 2012, automated Software Evolution.
7. S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in Proceedings of WCRE 2010, Oct 2010, pp. 35–44.
8. P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in Proceedings of ICPC 2014. New York, NY, USA: ACM, 2014, pp. 279–290.
9. P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in Proceedings of ICPC 2014. New York, NY, USA: ACM, 2014, pp. 291–294.
10. 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 ICPC 2013. May 2013, pp. 23–32.
11. P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in Proceedings of ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401.
12. G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in Proceedings of ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.
13. G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in Proceedings of ICSE 2011. May 2011, pp. 101–110.

AUTHOR(S) PROFILE

Suhas C. Mahakalkar, received the B.E degree in Computer Engineering and DAC from CDAC Mumai in 2006 and 2007, respectively. Seven years of IT experience as a Software Developer. He is currently pursuing M.E Degree in Computer Engineering from Savitribai Phule University, Pune, India.

Prof. A. D. Gujar, received M.E Degree in Information Technology from Bharti Vidyapith University Pune, and is currently working as Assistant Professor at TSSMS BSCOER Savitribai Phule University, Pune, India.

