Task-Driven Software Summarization

Dave Binkley[1], Dawn Lawrie[1], Emily Hill[2], Janet Burge[3], Ian Harris[4],
Regina Hebig[5], Oliver Keszocze[6], Karl Reed[7], John Slankas[8]
[1]Loyola University Maryland, Baltimore, MD, USA
[2]Montclair State University, Montclair, NJ, USA
[3]Miami University, OH, USA
[4]University of California, Irvine, CA, USA
[5]Hasso Plattner Institute at the University of Potsdam, Germany
[6]University of Bremen, Germany
[7] La Trobe University, Australia
[8]North Carolina State University, NC, USA
binkley@cs.loyola.edu, lawrie@cs.loyola.edu, hillem@mail.montclair.edu, burgeje@miamioh.edu harris@ics.uci.edu,
regina.hebig@hpi.uni-potsdam.de, keszocze@informatik.uni-bremen.de k.reed@latrobe.edu.au, jbslanka@ncsu.edu

***Abstract*—There is a growing interest in software summarization and tools for automatically producing summaries. Discussions of relevant papers at recent conferences led to the observation that software summarization needs to consider migrating away from "is this a *good* summary?" and towards "is this a *useful* summary?" As a result, it has been suggested that to judge usefulness, one needs to view the summary through the lens of a particular task. A preliminary investigation of this suggestion was undertaken at the 2013 ICSE workshop NaturaLiSE. Initial results and lessons learned from this investigation support the notion that task plays a significant role and thus should be considered by researchers building and accessing automatic software summarization tools.**

***Index Terms*—Source code summary, summarization evaluation, task-oriented**

I. Introduction

Comprehending source code is hard. A well written summary can aid a software maintainer in this task. The challenge here is that "well written" can depend on the current task. For example, *Renee the Reuser* benefits from context information, best practices for the effective use of the code, and the identification of replicated code. In contrast, *Teddy the Tester* benefits more from summaries focused on the functionality. Current summaries found in source code (i.e., comments) are often general, out of date, and even incorrect. They may, therefore, be inadequate for a given task. This is largely due to the fact that comments are written by engineers intimately familiar with the code, who can end up writing comments of use only to someone who already understands the code.

Recently, software engineering researchers have begun tackling the problem of automatic summarization of source code [1], [2], [3], [4], [5], [6]. The goal of automatic text summarization can be stated as "to generate a brief yet accurate representation (or summary) of a source document. An ideal summary is significantly shorter than the source document but retains its important information" [5]. What is missing from this notion of summarization is the potential influence of the consumer of the summary. The underlying question raised in this paper is

> *Does effective summarization require a specific target task or can source code be summarized in the abstract?*

While this could be the subject of a formal hypothesis test, the results reported herein are of a small experiment conducted recently as an indication of possible outcomes.

One of the major obstacles for source code summarization is that unlike natural language text, where all school children practice writing summaries, developers do not have as much practice summarizing computer programs. Therefore, they do not have as strong an innate sense for what should be part of a software summary.

Because of the challenges assessing the quality of automatic summarization's output, current empirical study tends to evaluate based on *goodness*, where a collection of programmers are asked if the summary satisfies some goodness measure, for example, if the summary is readable. While this assessment represents an important first cut, a more realistic assessment considers a summary's (down stream) *usefulness* on a software maintenance task. Does the summary help an engineer accomplish a task, perform it better or faster? To encourage movement beyond *good*, this paper presents a pilot study aimed at investigating how a target task impacts the content of a summary. The following section introduces the state of the art in text and source code summarization. Then the next section describes the target tasks and presents a pilot study for creating summaries for the target tasks. The paper concludes with lessons learned and a summary.

II. State of the Art

Beginning with Luhn, the automatic summarization of natural language text has been of interest for over fifty years [7]. As the field matured, template summaries and extractive summaries were proposed [8]. In template summaries, the system fills in blanks using text from a document. These types of summaries are limited by the scope of the templates. Extractive summaries extract sentences from documents and paste them together to form a summary. An extractive summary is more general purpose and can be used to summarize multiple

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.65

documents. In addition, it nicely avoids the difficult task of generating language; however, even the best summarizers (*i.e.*, humans) cannot produce good extractive summaries because they are hampered by the need to use sentences found in independent articles [9]. Another challenge is that a summary should include important facts, requiring highly subjective and content dependent judgments.

Recent summarization tasks have attempted to address these problems by disqualifying extractive summaries and giving more direction to both systems and humans when creating the summaries. One representative example of this is the Guided Summarization task at the Text Analysis Conference (TAC) where systems compete to produce the best summaries of ten newswire documents [10]. The summaries consist of 100 words and must include some predefined aspects based on the topics of the ten documents. For example, documents in the health and safety domain must address five aspects such as "who is affected by the health/safety issue." Human accessors then evaluate the automatically-produced summaries for readability, content, and overall responsiveness.

Automatic summaries are commonly evaluated relative to human produced summaries using the ROUGE (Recall Oriented Understudy for Gisting Evaluations) score, which evaluates an automated summary relative to a set of human summaries [11]. In particular the human summaries are reduced to n-grams, which are runs of n words from a document. For example, ROUGE-2 is based on the set of all the bi-grams in all the human summaries. The score for a summary is the number of n-grams from the human summaries found in the automated summary divided by the total number of unique n-grams in the human summaries.

Moving from automatic summarization of general texts to the summarization of software, the state of the art is surveyed by considering six recent projects. Early work on software summarization includes Murphy's dissertation on summarizing structural information in source code [1]. She studied how semi-automatic (iterative) summaries can enable an engineer to assess, plan, and execute changes to a software system. While this study was done in the context of software evolution, it did not consider producing software evolution specific (i.e., task specific) summaries.

More recently Moreno and Aponte compared tool-generated summaries with those created by Java developers [12]. They discovered that developers create summaries of similar length for all types of entities (e.g., methods and classes) while the length of automatically generated term-based summaries correlated with the length of the artifact summarized. They conclude that to get the gist of source code artifacts automatically, the length of a term-based summary should range from ten to twenty words nearer ten for methods and twenty for packages. One approach to encourage a target length is the use of template-based summaries such as those described in Sridhara's dissertation [2], where templates are used in the automatic generation of comments to summarize Java methods, collections of statements, and formal parameters.

A similar approach was recently presented by Moreno et al. who aim to generate human readable summaries for Java classes using class and method stereotypes [3]. While the summaries focus on content and responsibilities of a class rather than its relationships with other classes, they fall short of being task specific. When asked, programmers judged the generated summaries readable and understandable. This evaluation leaves open the *usefulness* of the summaries for a particular task, which was pointed out in the discussion following the presentation of this work at the conference.

Finally, at ICPC 2013 Gail Murphy's group presented two summarization approaches that hint at the need for task specific summarization. In the first, Rastkar and Murphy propose the use of multi-document summarization techniques to generate a natural language description of why code changed [6]. Their approach is extractive as it extracts full sentences to form a summary from the documents related to the change. Initial results show that overall developers found the summaries to contain information related to the reason behind the code change. However, they also note the evaluation needs to go further and investigate whether developers find the generated summaries *useful*. Doing so is likely to motivate the incorporation of task specific summarization techniques.

In the second approach, Kamimura and Murphy observe that automatic test generators (e.g., CodePro) produce code that can be difficult to comprehend [13]. They propose a technique for generating human-oriented summaries of test cases aimed at improving a humans' ability to quickly comprehend unit tests. They conclude by noting that "much more work is needed to make truly usable human oriented summaries." Here again this future work is likely to consider task summarization specifically *useful* to the task.

III. Pilot Study

From the previous section, it is evident that recent work on automatic summarization hints at taking task into account. As a precursor to the construction of such tools, the case study presented in this section considers *task specific manually produced summaries*. These are useful in assessing the role task plays. On the one hand, if task does not play a role in manually produced summaries then it is unlikely to play a role in automatic summaries. However, if it does play a role then there is a need for manually produced task-specific summaries against which to measure automatic tools.

For the case study, the participants of the 2013 NaturaLiSE workshop formed pairs to produce summaries of two classes selected from the program jEdit 4.2 by the first two authors. Over the course of an hour and a half, each pair examined one or both of the classes and produced two summaries for each class, one aimed at a reuser and the other at a tester. This section first describes the two tasks in greater detail, then the two selected classes, and finally the analysis of the summaries produced.

A. Tasks

To investigate the impact that task has on summarization, two tasks were considered. The tasks were personified by

Table I
A COMPARISON OF DisplayManager SUMMARIES TO StatusBar SUMMARIES. (BOLD ENTRIES SHOW TASK DOMINANCE.)

	StatusBar-Renee	StatusBar-Teddy
DisplayM-Renee-1	**0.148**	0.068
DisplayM-Teddy-1	0.076	0.072
DisplayM-Renee-2	**0.124**	0.090
DisplayM-Teddy-2	0.072	**0.084**
DisplayM-Renee-3	**0.060**	0.055
DisplayM-Teddy-3	0.065	**0.132**

```
public void appendPosition(String tag,
                int start, int end) {
  LinkedList<TMarkedStoreItem> ll =
    tagMap.get(tag);
  if (ll == null) {
    ll = new LinkedList<TMarkedStoreItem>();
    tagMap.put(tag, ll);
  }

  TMarkedStoreItem item =
    new TMarkedStoreItem();
  ll.add(item);
  item.end = end;
  item.start = start;
}
```

Figure 1. The method appendPosition for the program JabRef

Renee the Reuser and *Teddy the Tester*. To illustrate how summaries might differ given the specific tasks, Figure 1 shows a method found in JabRef 2.6. A summary directed towards Renee the Reuser might read:

Dear Renee,
Please be aware that this method creates lists of position pairs based on the <tag>.

In contrast, a summary directed towards Teddy the Tester might read:

Dear Teddy,
Please consider calling the same <tag> twice, once to test found and once to test not found.

While this example is simplistic, it is nevertheless evident that task driven summaries should emphasize different aspects of the code.

B. Code to Summarize

The classes for which the summaries were produced came from jEdit 4.2, a text editor for source code. This program was selected because it was presumed that workshop participants would be familiar with the domain vocabulary.

The two classes selected were DisplayManager and StatusBar. From each class multiple methods were also selected. The instructions asked participants to provide a summary for the chosen methods and then the class as a whole. For the DisplayManager class, the methods expandFold and narrow were chosen. Quantitatively DisplayManager includes 868 LoC and the two methods 111 LoC and 30 LoC respectively. For StatusBar, the methods propertiesChanged, statusUpdate, and updateBufferStatus were selected. Here StatusBar includes 482 LoC and the methods 69 LoC, 134 LoC, 9 LoC, respectively.

To give the reader a sense of the task of summarizing DisplayManager, understanding what a fold is within the editor along with the data structures that keep track of the folds are important background for writing a summary. The implementation uses parallel arrays of integers rather than, for example, creating a single array of fold descriptors. Considering StatusBar, the class visually displays information, which means that seeing the interface is likely to aid a person writing a summary.

C. Analysis

Pairs of participants examined the source code and then wrote two summaries (one for each task). One pair failed to identify the two separate summaries for the two tasks. Of the remaining three, only one pair considered both classes due to limited time. Thus, in the end, eight summaries were produced: two for StatusBar and six for DisplayManager.

To analyze the summaries, statistics were collected and summaries were compared pairwise using cosine similarity [14], which was computed by creating a vector space model in which each summary is treated as a bag of words comprised of word weights. The standard term frequency-inverse document frequency (tf-idf) was used to weight the words in each document's vector from which the cosine similarity was computed.

The eight summaries contained from between 18 and 103 non-stops words. The mean number of non-stop words was 42.5. A standard English language stop list was used for identifying the stop words, as summaries did not contain source code. The summaries were also stemmed using kstem [15] before computing cosine similarity, which is a standard practice in natural language to conflate words with different suffixes. When examining the unique stemmed vocabulary, summaries contained between 11 and 43 unique words with an average of 25.1 words. The total number of unique words used in all the summaries was 120, so a word occurred on average three times across the summaries.

Several observations concerning the summaries can be made by examining the similarity between pairs of summaries. The first observation is that there is evidence that there are words that are specific to testing and others that are specific to reuse. This can be seen in Table I, where the two StatusBar summaries are compared to the DisplayManager summaries. In the table, the summaries are labeled by task, followed by class name, and finally, for DisplayManager a number indicating the pair that produced the summary. The table reports the cosine similarity. In 83% of the cases (5 of 6), the summary for the particular task is more similar to the summary of the same task than the other task. These cases are shown in bold in the table. These patterns manifest in the summaries in the vocabulary used. For example, testing summaries included phrases such as "tester," "true and false," "range," and "completely test" while reuse summaries include "overall" and "the parameters."

The second observation comes from comparing the data in Table I with that shown in Table II, which shows the pairwise similarities between all pairs of DisplayManager summaries. First notice that the diagonal has all ones in it. This is expected

Table II
COSINE SIMILARITY COMPARISON OF DisplayManager SUMMARIES TO DisplayManager SUMMARIES

	DisplayM-Renee-1	DisplayM-Teddy-1	DisplayM-Renee-2	DisplayM-Teddy-2	DisplayM-Renee-3	DisplayM-Teddy-3
DisplayM-Renee-1	1.000	**0.434**	0.205	0.311	0.212	0.127
DisplayM-Teddy-1	**0.434**	1.000	0.322	0.293	0.202	0.150
DisplayM-Renee-2	0.205	0.322	1.000	**0.363**	0.179	0.140
DisplayM-Teddy-2	0.311	0.293	**0.363**	1.000	0.179	0.215
DisplayM-Renee-3	0.212	0.202	0.179	0.179	1.000	**0.579**
DisplayM-Teddy-3	0.127	0.150	0.140	0.215	**0.579**	1.000

when one compares something to itself because the vectors are identical. When considering the two tables together, the values in Table II are nearly all larger than the largest value in Table I, showing a dominance of terms from the domain of the code as opposed to the domain of the task (testing versus reuse). This is further supported by the fact that summaries for Renee are not necessarily more similar to each other than they are with summaries for Teddy in Table II, where all summaries are of the same source code.

Finally, Table II shows the expected result that the authors of the summary are more similar to themselves, moving between tasks than they are to summaries written for a particular task. This is born out in the similarity scores that appear in bold, which are the second largest in each row and likely indicate the importance of personal word choices. Examples causing authors to be more similar to themselves include the use of the term "method" versus "function".

IV. LESSONS LEARNED

After producing the two summaries, an open discussion was held by all participants to gather impressions and lessons learned. The dominant take home message was that summarizing non-trivial unfamiliar code is extremely challenging. This manifests itself in requests for the kind of information traditionally held in a data dictionary (e.g., acronym expansions, a data structure summaries, and an explanation of vague variable names).

Several participants suggested that dynamic information would have been helpful. For example, movies showing the Status Bar GUI in action or an example of a fold operation. This hints at potential future work on incorporating dynamic information into source code summarization.

Finally, a handful of the comments were meta comments regarding the summarization process itself. These included crowd sourcing the process to involve more geographically-separated researchers in developing a shared corpus and an appreciation for the two codes used not being trivial or just *academic exercises*. One pair writing the reuse summary first found the test summary easier to produce (and the resulting summary shorter). This is likely evidence of a learning effect. Finally, unlike the summarization of more standard natural language texts such as news articles, it was not clear to participants what level of detail was needed for the summarization. For example, one participant asked "does Renee need to understand how the code works to re-use it or is it sufficient just to know what it does?" Such a question further motivates the study of task-driven software summarization.

V. CONCLUSION

Task driven summarization is an important direction for software summarization as it provides focus for the summaries, which is evident in the fact that a person produces different summaries for different tasks. However, the task must be well defined. The development of aspects for the task, as in the TAC Guided Summarization Task, is most likely needed to produce greater consistency among human generated summaries. These could then be used as gold sets against which system performance could be judged using, for example, the ROUGE score.

ACKNOWLEDGMENT

Thanks to all workshop participants and in particular Alexandre Bassel and Michael Pradel for their help with the summaries. Support for this work was provided in part by NSF grant CCF 0916081.

REFERENCES

[1] G. Murphy, "Lightweight structural summarization as an aid to software evolution," PhD thesis, University of Washington, Washington, DC, USA, 1996.

[2] G. Sridhara, "Automatic generation of descriptive summary comments for methods in object-oriented programs," PhD thesis, University of Delaware, 2012.

[3] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *International Conference on Program Comprehension*, 2013.

[4] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the Working Conference on Reverse Engineering*, 2010.

[5] B. Eddy, J. Robinson, N. Kraft, and J. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *International Conference on Program Comprehension*, 2013.

[6] S. Rastkar and G. Murphy, "Why did this code change?" in *International Conference on Program Comprehension*, 2013.

[7] H. P. Luhn, "The automatic creation of literature abstracts," *IBM Journal of research and development*, vol. 2, no. 2, pp. 159–165, 1958.

[8] I. Mani, *Automatic summarization*. John Benjamins Publishing Company, 2001, vol. 3.

[9] P.-E. Genest, G. Lapalme, and M. Yousfi-Monod, "Hextac: the creation of a manual extractive run," in *Proceedings of the Second Text Analysis Conference, Gaithersburg, Maryland, USA. National Institute of Standards and Technology*, 2009.

[10] K. Owczarzak and H. Dang, "Overview of the tac 2010 summarization track," in *Proceedings of TAC 2010*, 2010.

[11] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, 2004, pp. 74–81.

[12] L. Moreno and J. Aponte, "On the analysis of human and automatic summaries of source code," *CLEI ELECTRONIC JOURNAL*, vol. 15, no. 2, 2012.

[13] M. Kamimura and G. Murphy, "Towards generating human-oriented summaries of unit test cases," in *International Conference on Program Comprehension*, 2013.

[14] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge Press, 2008.

[15] R. Krovetz, "Viewing morphology as an inference process," in *Proceedings of the 16th ACM SIGIR Conference*, R. K. et al., Ed., June 1993.