Why Did This Code Change?

Sarah Rastkar and Gail C. Murphy
Department of Computer Science
University of British Columbia, Canada
{rastkar, murphy}@cs.ubc.ca

***Abstract*—When a developer works on code that is shared with other developers, she needs to know why the code has been changed in particular ways to avoid reintroducing bugs. A developer looking at a code change may have access to a short commit message or a link to a bug report which may provide detailed information about how the code changed but which often lacks information about what motivated the change. This motivational information can sometimes be found by piecing together information from a set of relevant project documents, but few developers have the time to find and read the right documentation. We propose the use of multi-document summarization techniques to generate a concise natural language description of why code changed so that a developer can choose the right course of action.**

I. Introduction

A developer working as part of a software development team often needs to understand the reason behind a code change. This information is important when multiple developers work on the same code as the changes they individually make to the code can conflict. Such collisions might be avoided if a developer working on the code knows why particular code was added, deleted or modified. Often, the developer can access a commit message or a bug report with some information on the code change. However, this information often does not provide the context about why the code changed, such as which business objective or feature was being implemented. This higher-level information is often available in a set of project documents, perhaps requirements and design documents or perhaps epics and user stories. But finding and understanding this information that is spread across multiple documents is time consuming and cognitively demanding and thus seldom happens in the context of a code change.

In this paper, we propose the use of multi-document summarization techniques to generate a concise description of the motivation behind a code change based on the information in a set of documents relevant to the change. We propose an extractive summarization approach in which we identify the most relevant sentences to a change in a set of documents provided as including motivational information about the change. This extractive approach uses machine-learning to identify appropriate sentences in the given set of documents. We identified a set of sentence-level features to locate the most relevant sentences in a change-related set of documents to be included in a summary.

We have experimented with applying this approach to a corpus of human-generated summaries. Our results show that our set of features is effective in identifying sentences that are considered important by human summarizers. This shows the possibility of providing much broader motivational information behind a change than has been possible with previous approaches allowing a developer to ask and have answered, "why did this code change?".

We begin with an example and then describe our initial approach. We then provide an overview of our experiment applying the approach to the corpus we developed. We then review the related work and conclude the paper with discussion.

II. Motivating Example

Consider a developer who is working on the open source project CONNECT.[1] This project uses an agile development process in a way that makes information relevant to code changes available through explicit links to multiple levels of tasks, stories and epics that were used to organize the work that was performed. Imagine that the developer needs to make a change to the `provideAndRegisterDocumentSetb` method of the `AdapterDocSubmissionImpl` class as part of the task which she is working on. Since the developer last looked at the code, a call to `deleteFilesFromDisk` has been added to this method. The developer wants to know why there is this new dependency before determining how to proceed with her change. She starts by checking the commit message associated with the last modification to the method of interest, which states: "*GATEWAY-905: Added code to remove temporary files during streaming*". This gives a clue about the added code but does not say why streaming support has been added. The developer decides to investigate further by following the provided link to `GATEWAY-905` in the project's issue repository.[2] `GATEWAY-905`, labeled as a *task* in the repository and entitled *"Create a component to remove temporary streamed files from the file system"*, still does not provide any high-level reason of why streaming was needed. `GATEWAY-905` is linked to `GATEWAY-901` which is labeled as a *user story* and is entitled *"Create a pilot implementation for streaming large files for the Document Submission service"*. Finally, `GATEWAY-901` is linked to `GATEWAY-473`, labeled as a *Feature Request* and entitled *"As a CONNECT Adopter, I want to process and send large payload sizes of up to 1 GB to meet the requirements of all my use cases (User Story: EST010)*". To find this motivational

[1]http://connectopensource.org, verified 12/12/12, CONNECT is an open source project promoting IT interoperability in the U.S. healthcare system.

[2]https://issues.connectopensource.org, verified 12/12/12

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
New Ideas and Emerging Results

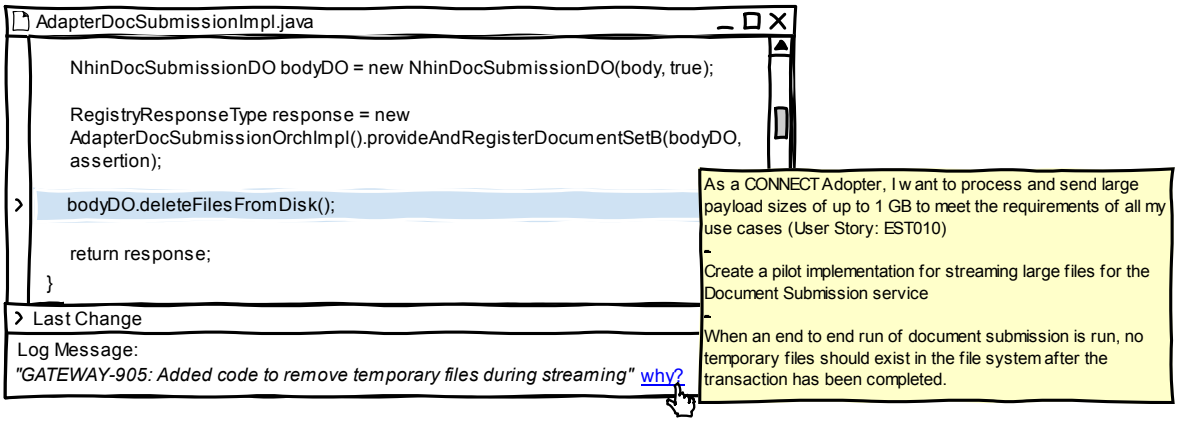

Fig. 1. A summary describing the motivation behind a code change of interest appearing as a pop-up in an IDE.

information, the developer has to trace the change to a chain of three inter-linked documents in the issue repository, each at a different level of abstraction, and has to read through the description and comments of those documents. If a summary could be automatically created with this information, it could be made accessible to the developer at the site of the code change. For example, imagine that a developer highlights a code change of interest in an IDE and the pop-up shown in Figure 1 appears making the information immediately accessible to the developer. This summary can help a developer immediately understand the motivation behind the change with very little effort on her part.

III. Approach

We see the problem of generating a summary of a set of documents related to a code change as a specific case of the more general problem of multi-document summarization, which often is defined as the automatic generation/extraction of information from multiple texts written about the same topic. Multi-document summarization techniques often deal with summarizing a set of similar documents that are likely to repeat much the same information while differing in certain parts; for example, news articles published by different news agencies covering the same event of interest. Our approach is different in that it investigates summarizing a set of documents each at a different level of abstraction.

To account for the fact that each document is at a different level of abstraction, we model the set as a hierarchical chain. For the example discussed in Section II, `GATEWAY-473` (a *feature request*) and the commit message are the top-most and the bottom-most documents in the chain of relevant documents respectively.

Once a chain of relevant documents is formed, we then find the most informative sentences to extract to form a summary. We identified a set of eight sentence-level features to locate the most relevant sentences. Based on the values of these features, sentences can be ranked and then a summary can be produced by extracting the highest-ranked sentences.

The eight features we investigated are:

1) $fOverlap$. This feature measures the content overlap between a sentence and the adjacent documents in a chain. There is often some content overlap in terms of common words between adjacent documents in a chain as one has motivated the creation of the other one (e.g., an epic motivating a user story). We hypothesize that sentences containing these words are more likely to explain the motivation behind the corresponding code change and should be scored higher. To locate these sentences we look for overlaps in terms of words between adjacent documents in a chain. This is similar to the idea of clue word score (CWS) previously shown to be an effective feature in summarizing conversational data (e.g., [1]). For word w in document D in a chain of documents, $overlapScore$ is computed as follows:
$$overlapScore(w, D) = IDF(w) \times [TF(w, Parent(D)) + TF(w, Child(D))]$$
$TF(w, D)$ and $IDF(w)$ are defined as:
$$TF(w, D) = \textit{number of times } w \textit{ appears in } D$$
$$IDF(w) = log(\frac{\textit{number of documents}}{\textit{number of documents containing } w})$$
For sentence s, $fOverlap$ is computed as:
$$fOverlap = \sum_{w \in s} overlapScore(w, D)$$
2) $fTF\text{-}IDF$. For a sentence s in document D, this score is computed as the sum of the $TF\text{-}IDF$ scores of all words in the sentence:
$$fTF\text{-}IDF = \sum_{w \in s} TF(w, D) \times IDF(w)$$
It is hypothesized that a higher value of $fTF\text{-}IDF$ for a sentence indicates that the sentence is more representative of the content of the document and hence is more relevant to be included in the summary.
3) $fTitleSim$. This feature measures the similarity between each sentence and the title of the enclosing document. If document titles are chosen carefully, the title of each document in a chain is a good representative of the issue discussed in the document. Hence it is

hypothesized that the similarity between a sentence in a document and the title of the document is a good indicative of the relevance of the sentence. Similarity to the title has been previously shown to be helpful in summarizing email threads [2] and bug reports [3]. $fTitleSim$ is computed as the cosine between the TF-IDF vectors of the sentence and the document's title.

4) $fCentroidSim$. While $fTitleSim$ is a local feature as it computes the similarity of a sentence with the title of the containing document, $fCentroidSim$ is a global feature as it measures the similarity of the sentence with the centroid of the chain. A centroid is a vector that can be considered as representative of all documents in a chain. Centroid-based techniques have extensively been used in multi-document summarization (e.g, [4]). We take the approach of multi-document summarization system MEAD [4] to compute $fCentroidSim$. The chain centroid is computed as a vector of words' average TF-IDF scores in all documents in the chain. For each sentence, $fCentroidSim$ is computed as the sum of all centroid values of common words shared by the sentence and the centroid.
5) $fDocPos$. This feature captures the relative position of a sentence in the enclosing document. The motivation is that it might be the case that opening or concluding sentences are more important than the rest of sentences. $fDocPos$ is computed by dividing the number of preceding sentences in the document by the total number of sentences in the document.
6) $fChainPos$. This feature is similar to $fDocPos$, but it measures the relative position of a sentence in the chain.
7) $fDocLen$. Usually longer sentences are more informative. $fDocLen$ is the length of the sentence normalized by dividing it by the length of the longest sentence in the document.
8) $fChainLen$. Similar to $fDocLen$ with the length of the sentence divided by the length of the longest sentence in the chain.

IV. Does the Approach Work?

To investigate if the features discussed in Section III can be effective in identifying important sentences, we created a corpus of human generated summaries of chains of documents related to code changes. As an initial corpus, we identified eight chains in the CONNECT project, each linking together three documents from the project's issue repository and one commit log message. The average length of selected chains is 386±157 words. This is four times the size of the example chain summary shown in Figure 1.

We recruited seven human summarizers (all from the UBC Department of Computer Science) and asked them, for each chain of documents related to a code change, to create a summary explaining the motivation behind the change by highlighting sentences that should be included in such a summary. Similar to the approach taken by Carenini and colleagues [1], human summarizers were asked to distinguish between selected sentences by labeling them as *essential* for critical sentences that always have to be included in the summary or *optional* for sentences that provide additional useful information but can be omitted if the summary needs to be kept short. Human summarizers were advised to choose at most one third of sentences in a chain (whether labeled as *essential* or *optional*). Each chain was summarized by three different people. For each chain, we merged these three summaries to a gold standard summary by extracting sentences with the following set of labels: {*optional*, *essential*}, {*essential*, *essential*}, {*optional*, *optional*, *optional*}, {*optional*, *optional*, *essential*}, {*optional*, *essential*, *essential*} and {*essential*, *essential*, *essential*}.

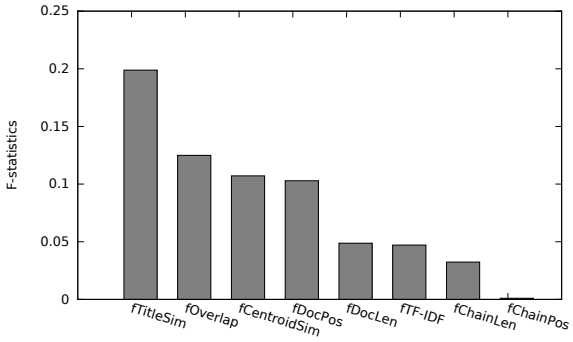

Fig. 2. F-statistics value for sentence features

We used the feature analysis technique introduced by Chen and Lin [5] to compute the F-statistics measure for all 8 sentence features based on the gold standard summaries in our corpus. Figure 2 shows F-statistics values for each feature. Based on these values $fTitleSim$ and $fOverlap$ are the two most helpful features in identifying the important sentences. As we hypothesized earlier, $fTitleSim$ is a useful feature because the titles of the documents were well-chosen and well-phrased. $fOverlap$ is a useful feature because the document authors have used repetition and similar phrases to discuss motivation. On the other hand, $fChainLen$ and $fChainPos$ are the two least helpful features. Removing the two least helpful features, we trained a classifier based on Support Vector Machines (SVMs) using the *libSVM*[3] toolkit. This classifier has a 10-fold cross validation accuracy of over 82% showing a high level of effectiveness in identifying important sentences.

V. Related Work

A. Analyzing Code Changes

Various approaches have addressed analyzing source code changes to gain insight about past and current states of a software project. Examples include identification (e.g., [6]), impact analysis (e.g., [7]) and visualization (e.g., [8]) of code changes. While these approaches mainly focus on the 'what' and 'how' of a code change, the approach presented in this paper tries to address the 'why' behind a code change.

[3]http://csie.ntu.edu.tw/~cjlin/libsvm, verified 12/12/12

B. Summarizing Software Artifacts

While our proposed approach focuses on generating summaries to address a particular question ('why did this code change?'), other approaches have considered generating general-purpose summaries of software artifacts. This includes efforts on producing summaries of source code, for example term-based summaries for methods and classes [9], natural language summary comments for arbitrary Java methods [10] and natural language summaries for crosscutting concerns [11]. Other work investigated extractive summarization of bug reports, including a supervised summarization approach relying on the conversational nature of bug reports [12] and an unsupervised approach based on the PageRank algorithm [3].

C. Multi-document Summarization

In contrast to other work on summarizing software artifacts which investigates summarization of a single document, the approach proposed in this paper focuses on multi-document summarization of software artifacts. Most multi-document summarization system rely on content similarity among documents. Examples include centroid-based (e.g., MEAD [4]), cluster-based (e.g., [13]) and graph-based (e.g., [14]) summarization approaches. Our approach differs in the sense that the summary is generated from a set of documents each at a different level of abstraction.

VI. Discussion

In this paper we proposed an approach based on multi-document summarization techniques to produce a summary providing the motivational information behind a code change. The proposed approach relies on input in the form of a set of documents related to the change. It is also assumed that the documents in that set are at different levels of abstractions and thus can be ordered in the form of a chain. In some projects, like CONNECT which was used in this paper, there are explicit links between documents that can be used to track and form a chain of relevant document to a code change. In cases where relevant information is spread across documents in different repositories with no explicit links between documents, techniques like text mining and information retrieval (e.g., [15]) could be used to retrieve a set of documents related to a code change.

In this paper, as a first step, we took an extractive approach where full sentences are extracted to form a summary. Another option would be to form a shorter and easier to go through summary by extracting keywords or phrases from a set of change-related documents. It is yet to be investigated whether such summaries can effectively be generated and whether they can address the information need of developers.

There are several unanswered questions regarding the approach presented in this paper. For example, it is unclear how well the classifier trained on our human-annotated corpus generalizes to data from other projects. Also, it needs to be investigated whether developers find the generated summaries helpful. As the first step towards answering these questions, we performed an exploratory study in which we used the classifier to generate summaries for a few chains from a different open source project, Eclipse Mylyn.[4] For each chain, we asked the Mylyn developer who had made the code change whether the chain summary contained information relevant to the reason behind the change. Our initial results show that overall the developers found the summaries to contain information related to the reason behind the code change. The developers suggested altering the approach to include more technical details related to the code changes they examined.

Acknowledgment

We would like to thank Julius Davies and the reviewers for comments on an earlier version of this paper. This work was funded by NSERC.

References

[1] G. Carenini, R. Ng, and X. Zhou, "Summarizing email conversations with clue words," in *WWW'07: Proc. of the 16th International World Wide Web conf.*, 2007, pp. 91–100.

[2] S. Wan and K. McKeown, "Generating overview summaries of ongoing email thread discussions," in *COLING'04: Proc. of the 20th International conf. on Computational Linguistics*, 2004, pp. 549–556.

[3] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *ICSM'12: Proc. of the 28th International conf. on Software Maintenance*, 2012.

[4] D. Radev, T. Allison, S. Blair-Goldensohn, J. Blitzer, A. Celebi, S. Dimitrov, E. Drabek, A. Hakim, W. Lam, D. Liu *et al.*, "MEAD-a platform for multidocument multilingual text summarization," in *LREC'04: Proc. of the International conf. on Language Resources and Evaluation*, 2004.

[5] Y.-W. Chen and C.-J. Lin, "Combining SVMs with various feature selection strategies," in *Feature extraction, foundations and applications*. Springer, 2006, pp. 315–324.

[6] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Trans. on*, vol. 33, no. 11, pp. 725 –743, nov. 2007.

[7] R. Purushothaman and D. Perry, "Toward understanding the rhetoric of small source code changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 511 – 526, june 2005.

[8] S. L. Voinea, A. Telea, and M. Chaudron, "CVSscan: Visualization of code evolution," in *Softviz'05: Proc. of the 2005 ACM Symposium on Software Visualization*, 2005, pp. 47–56.

[9] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *WCRE'10: Proc. of the 17th Working conf. on Reverse Engineering*, 2010, pp. 35–44.

[10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *ASE'10: Proc. of the 25th international conf. on Automated software engineering*, 2010, pp. 43–52.

[11] S. Rastkar, G. Murphy, and A. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *ICSM'11: Proc. of the 27th International conf. on Software Maintenance*, 2011, pp. 103–112.

[12] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *ICSE'10: Proc. of the 32nd International conf. on Software Engineering*, 2010, pp. 505–514.

[13] X. Wan and J. Yang, "Multi-document summarization using cluster-based link analysis," in *SIGIR'08: Proc. of the 31st annual international ACM SIGIR conf. on research and development in information retrieval*, 2008, pp. 299–306.

[14] G. Erkan and D. Radev, "Lexpagerank: Prestige in multi-document text summarization," in *EMNLP'04: Proc. of the 2004 conf. on Empirical Methods on Natural Language Processing*, 2004, pp. 365–371.

[15] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[4] www.eclipse.org/mylyn, verified 12/12/12