Summarizing Software Artifacts

by

Sarah Rastkar

B.Sc., Sharif University of Technology, 2001
M.Sc., Sharif University of Technology, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

May 2013

Abstract

To answer an information need while performing a software task, a software developer sometimes has to interact with a lot of software artifacts. This interaction may involve reading through large amounts of information and many details of artifacts to find relevant information.

In this dissertation, we propose the use of automatically generated summaries of software artifacts to help a software developer more efficiently interact with software artifacts while trying to answer an information need. We investigated summarization of bug reports as an example of natural language software artifacts, summarization of crosscutting code concerns as an example of structured software artifacts and multi-document summarization of project documents related to a code change as an example of multi-document summarization of software artifacts.

We developed summarization techniques for all the above cases. For bug reports, we used an extractive approach based on an existing supervised summarization system for conversational data. For crosscutting code concerns, we developed an abstractive summarization approach. For multi-document summarization of project documents, we developed an extractive supervised summarization approach.

To establish the effectiveness of generated summaries in assisting software developers, the summaries were extrinsically evaluated by conducting user studies. Summaries of bug reports were evaluated in the context of bug report duplicate detection tasks. Summaries of crosscutting code concerns were evaluated in the context of software code change tasks. Multi-document summaries of project documents were evaluated by investigating whether project experts find summaries to contain information describing the reason behind corresponding code changes.

The results show that reasonably accurate natural language summaries can be automatically produced for different types of software artifacts and that the generated summaries are effective in helping developers address their information needs.

Preface

Parts of the research presented in this dissertation has been previously published in the following articles:

1. Sarah Rastkar, Gail C. Murphy, Gabriel Murray, "Summarizing software artifacts: a case study of bug reports", In *ICSE'10*: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, Pages 505-514, 2010, ACM.

2. Sarah Rastkar, "Summarizing software concerns", In *ICSE'10*: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, Pages 527-528, 2010, IEEE.

3. Sarah Rastkar, Gail C. Murphy, Alexander W.J. Bradley, "Generating natural language summaries for crosscutting source code concerns", In *ICSM'11*: Proceedings of the 27th IEEE International Conference on Software Maintenance, Pages 103-112, 2011, IEEE.

4. Sarah Rastkar, Gail C. Murphy, "Why did this code change?", In *ICSE'13*: Proceedings of the 35th ACM/IEEE International Conference on Software Engineering, New Ideas and Emerging Results (NIER) Track, 2013, to appear.

Part of this work involved other collaborators. Gabriel Murray contributed in designing the annotation process for the bug report corpus, training classifiers on email and meeting data and providing the code used to compute sentence features for conversation-based data. Alexander Bradley developed the plug-in for integrating crosscutting concern summaries for Java programs into the Eclipse IDE.

The UBC Behavioural Research Ethics Board approved the research in the certificates H10-01044 ("Summarization Study") and H11-00246 ("Software feature summarization") and in amendments and renewals to these certificates (H10-01044-A001, H10-01044-A002, H10-01044-A003, H10-01044-A004, H10-01044-A005, H11-00246-A001, H11-00246-A002).

Table of Contents

List of Tables

List of Figures

Acknowledgment

There are no proper words to convey my deep gratitude and respect for my supervisor, Gail Murphy. I feel extremely fortunate to have spent the past few years working under Gail's supervision. Since our very first meeting, she has constantly amazed and inspired me by her unique mixture of intellect, dedication and empathy. Leading by example, she has taught me to strive for excellence in research; to pay attention to all the details without losing sight of the big picture. This work would have been impossible without you, Gail. Thank you!

I would like to thank the members of my supervisory committee, Giuseppe Carenini and Raymond Ng for their support, encouragement and insightful feedback over the last few years and for sharing data and code developed by their research group. Thanks also go to the members of my examining committee, Philippe Kruchten, Lori Pollock and Eric Wohlstadter for their invaluable time and effort put into reading my thesis.

Special thanks to my parents for their unconditional love and unwavering support. They always encouraged me in everything I have done, even when it involved moving thousands of miles away from them. Thanks to my brother, Mohammad, for being a source of encouragement, for always having time to listen and for often reminding me that it was going to be over soon.

I am indebted to many friends and colleagues for their help and support during my PhD experience. They participated in my pilot studies, provided feedback on my paper drafts and practice talks, assisted with programming tasks or helped me take my mind off work when it was needed most. Many thanks to Albert, Alex, Apple, Baharak, Deepak, Emerson, Gabe, Gias, João, Julius, Justin, Lee, Meghan, Mona, Mona, Nick, Neil, Nima, Peng, Rahul, Reza, Roberto, Robin, Roozbeh, Ron, Sam, Sara, Sara, Seonah, Solmaz, Thomas and Vahid. Thanks also to all who participated in my numerous user studies or helped in annotating the corpora.

Thanks to all the staff at the Computer Science Department, in particular Michele Ng and Hermie Lam for their support and their infectious positive energy.

I would like to acknowledge the National Science and Engineering Re-

search Council of Canada (NSERC) for funding my research.

To Mom & Dad.

Chapter 1

Introduction

For a large-scale software project, significant amounts of information about the software and the development process are continuously created and stored in the project's repositories. This fast changing information comes in the form of different software artifacts including source code, bug reports, documentation, mailing list discussions and wiki entries. As an example of the substantial amount of information stored in a software project's repositories, there are more than 397,000 bug reports in the bug repository of Eclipse, an integrated development environment,[1] with more than 24,000 bug reports added to the repository in 2012. Also, more than 260,000 lines of code are added to the Eclipse's code repository per each major development cycle (12 months) [65].

A software developer working on a software task typically needs to consult information in the project repositories to address her various information needs. For example, for a newly submitted bug report, the developer may want to know if the problem described in the bug has been already reported [4]. When working on a code change task, the developer may want to know what parts of the code are relevant to the task [7], if similar changes have been made to the code in the past [16], or why the code was previously changed in a certain way [48]. To address these information needs, the developer has to navigate, query and search the software project's repositories and interact with many software artifacts. Various navigation, search and recommendation tools based on different mining techniques have been developed to help narrow the search space for a developer looking for information. For example, various bug report duplicate detection approaches have been proposed to recommend a list of potential duplicate bug reports to help a developer decide if a new bug report is a duplicate of an existing

[1] `www.eclipse.org`, verified 12/12/12

one (e.g., [91, 96, 106]). Several tools have been developed to recommend similar changes to the code (e.g., [5, 102]) and various techniques have been proposed to facilitate the search and navigation of source code (e.g., [43, 86]). Even when using these techniques, only a subset of the artifacts returned through them addresses the information need of the developer. To figure out which of the returned artifacts contain relevant information, a developer typically has to interact with each artifact. Sometimes a developer can determine relevance based on a quick read of the title of a bug report or a quick scan of a source file. Other times, lengthy discussions in bug reports may need to be read to see if the report includes pertinent information, substantial amounts of code may need to be investigated to figure out how different code elements are related, or documentation may need to be perused to understand how and why the code has evolved in a certain way. All of this work requires dealing with large amounts of information and many details of the artifacts. As an example, a developer using the bug report duplicate recommender built by Sun and colleagues [96] to get a list of potential duplicates for bug #564243 from the Mozilla system[2], has to go through a total of 5125 words (237 sentences, approx. 50 paragraphs) to investigate the top six bug reports (#540841, #562782, #549931, #542261, #550573, #541650) on the list.

My thesis is that the automatic provision of concise natural language summaries of software artifacts makes it easier for developers to more efficiently determine which artifacts address their information needs. A summary represents the most pertinent facts of a single software artifact or a collection of related software artifacts in a shorter form eliding less pertinent details. Once a developer determines if a software artifact contains relevant information based on this alternate representation, she can investigate the artifact in further details. Or, the developer can more efficiently determine an artifact does not require more investigation.

The content of most software artifacts is a combination of two different types of information: structured information and natural language information. Based on their content, software artifacts form a spectrum, where software artifacts with mostly structured information (e.g., code) are at one

[2] `www.mozilla.org`, verified 12/12/12

end and software artifacts with mostly natural language information (e.g, bug reports, documentation) are at the other end. To investigate whether our hypothesis holds across this range, we address both ends of this spectrum in this dissertation. Because software artifacts from each end of this spectrum contain information with different characteristics, different summarization approaches are needed for each of them. In a broad sense, a summarization approach can be either *extractive* or *abstractive* [40]. An *extractive* summary is formed by extracting full sentences from the original document(s) while an *abstractive* summary is formed by first creating a semantic representation of the content of the input document(s) and then generating sentences to convey the information in the intermediate semantic representation. We propose the use of extractive approaches for summarizing software artifacts with mainly natural language content as the current state of the art in summarizing natural language text relies on sentence extraction [72]. For software artifacts with mostly structured information, we propose the use of abstractive summarization as the precise structure of the content of such artifacts makes it feasible to build an abstract semantic representation. In both cases, even when the content of the input software artifact is mostly structured, we produce a *natural language* summary as opposed to other alternate forms such as keyword summaries or diagrams. We decided to focus on the natural language format as it provides several benefits. First, we believe natural language text is easy to read and understand and is flexible for including domain knowledge. Second, developers do not need training in a new language or formalism to use the produced summaries. Third, we can easily control the size of a natural language summary to be of a particular number of words. Natural language text has been shown to be more effective than diagrams in other applications such as decision making in medical scenarios [53, 101] and understanding product manuals [52].

1.1 Summarizing Natural Language Software Artifacts

Many software artifacts contain mostly natural language information, including requirement documents, bug reports, project's mailing list discussions and wiki entries. We investigate bug reports to explore the problem of summarizing software artifacts with mostly natural language content. Software developers access bug reports in a project bug repository to help with a number of different tasks, including understanding multiple aspects of particular defects [11, 15] and understanding how previous changes have been made [5, 102]. We hypothesize that summaries of bug reports can provide developers with enough essence of a report to more efficiently determine which bug reports in a repository contain relevant information.

1.1.1 Approach

Bug reports often consist mainly of a conversation among multiple people. Figure 1.1 shows an example of the conversational structure of a bug report. Since bug reports are similar to other conversational data, for example email threads and meetings, existing approaches for summarizing such data may be utilized. We thus applied an existing extractive conversation-based summarizer, developed by Murray and Carenini [67], to generate summaries of bug reports. This supervised approach relies on training a classifier on a human-annotated corpus. We investigated two classifiers trained on general conversation data, one trained on email data and the other trained on a combination of email and meeting data. We chose these general classifiers to investigate whether they can generate accurate enough summaries for bug reports. To understand whether summaries can be substantially improved by using a domain-specific training set, we also investigated a classifier trained on a bug report corpus we created. The corpus consists of 36 bug reports from four open source projects. We had human annotators create summaries for each of these reports. Using any of the three classifiers, a summary can be produced for a bug report. Figure 1.2 shows a 100-word summary of the bug report shown in Figure 1.1, produced with the classifier trained on our bug report corpus. The sentences in this summary have been

Bug 540914 - IMAP: New mail often not displayed in folder pane (read & unread folders). Mail not seen until going offline

Product: Thunderbird
Component: Folder and Message Lists
Version: 3.0
Platform: x86 Windows XP

Reported: 2010-01-20 12:53 PST by Chuck
Modified: 2011-07-31 09:21 PDT (History)
CC List: 7 users (show)

Chuck 2010-01-20 12:53:17 PST **Description**

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.7) Gecko/20091221 Firefox/3.5.7
Build Identifier: 3.0

Setup: IMAP with 150 subscribed folders, frequent checking for new mail, all filtering done at IMAP server. TB is left running overnight.

Problem: New mail not displayed in some (no pattern detected yet) folders. Those folders may have all mail read, in which case the folder name does NOT change to bold black, and folder names with previously unread mail do NOT change to blue.

Other folders (no pattern detected yet) seem to work as expected.

Not a problem with TB2.

Workaround: I have to manually step through each folder name in the folder pane with the down arrow to "update" the folder's display. When I do this, the message count and bold or blue color change occurs and I know I have new mail.

Reproducible: Sometimes

Steps to Reproduce:
1.See details field above

Actual Results:
1.See details field above

Expected Results:
New mail should be indicated by either a bold folder name (no previously unread mail) in the folder pane or a blue folder name (previously unread mail) and an updated message count.

One shouldn't have to manually refresh the folder by highlighting the folder name to be informed of the new mail.

Ludovic 2010-01-21 05:39:21 PST **Comment 1**

So the mails are in the folder but the folders aren't updated ?

Chuck 2010-01-21 10:34:39 PST **Comment 2**

How could I differentiate?

When I highlight a folder that appears to have all mail read and it turns bold black and adds a message count e.g. (1), I also see network traffic indicating it's checked the IMAP server.

Figure 1.1: An example of the conversational structure of a bug report; the beginning part of bug #540914 from the Mozilla bug repository. The full bug report consists of 15 comments from 7 people.

SUMMARY: IMAP: New mail often not displayed in folder pane (read & unread folders). Mail not seen until going offline

Those folders may have all mail read, in which case the folder name does NOT change to bold black, and folder names with previously unread mail do NOT change to blue. New mail should be indicated by either a bold folder name (no previously unread mail) in the folder pane or a blue folder name (previously unread mail) and an updated message count.

Below is the log for a ten-minute new mail check that does NOT find any mail in the INBOX, however I DO have new mail on the IMAP server in other subscribed folders - TB never pulls it down unless I manually refresh a folder or restart TB.

Figure 1.2: The 100-word summary for bug #540914 from the Mozilla bug repository generated by the classifier trained on bug report data.

extracted from the first comment (the description) and the sixth comment of the bug report.

1.1.2 Evaluation

We measured the effectiveness of the three classifiers, finding that the first two classifiers, trained on more generic conversational data, can generate reasonably good bug report summaries. We also found that the bug report classifier, having a precision of more than 66%, moderaltely out-performs the other two classifiers in generating summaries of bug reports. To evaluate whether a precision of 66% produces summaries useful for developers, we conducted two independent human evaluations. In the first evaluation, we had human judges evaluate the goodness of a subset of the summaries produced by the bug report classifier. On a scale of 1 (low) to 5 (high), the arithmetic mean quality ranking of the generated summaries by the human judges was 3.69 ($\pm$1.17), which suggests that the generated summaries may be helpful to developers. To determine if the summaries are indeed helpful, in the second human evaluation, we conducted a study in which we had 12 participants work on eight bug report duplicate detection tasks. We found that participants who used summaries could complete duplicate detection tasks in less time without compromising the level of accuracy, confirming that bug report summaries help software developers in performing software tasks.

1.1.3 Related Work

Our proposed approach (described in more details in Chapter 3) is the first attempt at summarizing natural language software artifacts. Since the publication of our initial results [84], other efforts have investigated using unsupervised approaches to summarize bug reports [56, 59]. While they claim improvement in the analytical results over our approach, none of them has subjected their work to a human task-based evaluation.

1.2 Summarizing Structured Software Artifacts

We focused on the summarization of code as an example of a software artifact which contains mostly structured information, but also some natural language information in the form of comments and identifier names. Programmers must typically perform substantial investigations of code to find parts that are related to the task-at-hand. It has been observed that programmers performing change tasks have particular difficulty handling source code that crosscuts several modules in the code [7]. Such crosscutting code which is scattered across different modules of the system, is often referred to as a *concern* and is usually implementing a specific feature in the system, e.g., logging or synchronization. Figure 1.3 shows how each module of the system might be intersected with several crosscutting concerns. We chose to focus on summarization support for crosscutting code concerns. We hypothesize that a natural language summary of a concern allows a programmer to make quick and accurate decisions about the relevance of the concern to a change task being performed. Our proposed approach automatically produces a summary about a crosscutting concern given a list of methods generated by an existing concern identification technique. The aim of the generated summary is to allow a programmer to make quick and accurate decisions about the relevance of a concern to a change task being performed.

1.2.1 Approach

Our abstractive approach to summarize a software concern consists of three steps. First, we extract structural and natural language information from the source code to build a unified semantic representation of the concern

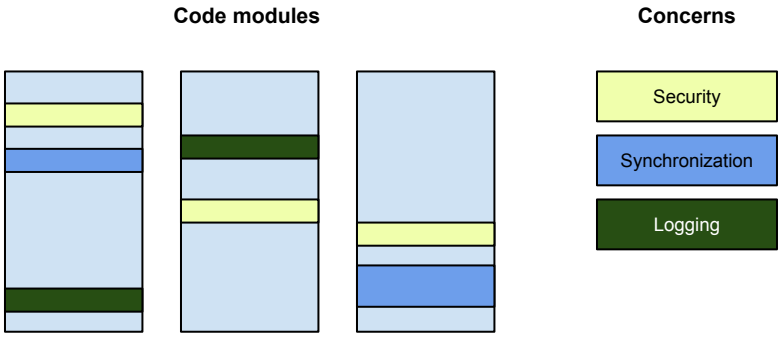


Figure 1.3: Crosscutting code concerns

integrating both types of information. Second, we apply a set of heuristics to the extracted information to find patterns and identify salient code elements in the concern code. Finally, using both the extracted information and the content produced from the heuristics, we generate the sentences that form the summary. Figure 1.4 shows an example summary created by our approach for a concern implementing *undo* feature in JHotDraw, an open source graphical framework.[3]

1.2.2 Evaluation

To determine if generated summaries can help programmers investigate code relevant to a given change task, we conducted a laboratory study in which eight programmers performed two software change tasks on two different software systems. For the second task performed, the programmers had access to concern summaries. We found that when concern summaries were available, the programmers found it easier to perform the specified task and were more confident about their success in completing the task. We also found evidence that the programmers were able to effectively use the concern summaries to locate code pertinent to the change task.

[3] `www.jhotdraw.org`, verified 12/12/12

Summary of 'Undo' Feature Implementation

1: The 'Undo' feature is implemented by at least 22 methods [show/hide].

2: This feature provides 'undoing' functionality for 'Select All Command', 'Connected Text Tool', etc. [show/hide].

3: The implementation of the 'Undo' feature highly depends on the following code element(s):
4: • org.jhotdraw.util.UndoableAdapter.undo().
5: • org.jhotdraw.util.Undoable.undo().

6: All of the methods involved in implementing 'Undo':
7: • are named 'undo'.
8: • override method org.jhotdraw.util.UndoableAdapter.undo().
9: • override method org.jhotdraw.util.Undoable.undo().
10: • are a member of a class named 'UndoActivity'.

[*3 other patterns involving all methods*]

[*6 patterns involving all but one of methods*]

11: Around half of the methods involved in implementing 'Undo' call one or more of the following methods:
12: • org.jhotdraw.framework.DrawingView.clearSelection().
13: • org.jhotdraw.util.UndoableAdapter.getAffectedFigures().

[*2 other patterns involving about half of methods*]

Figure 1.4: A part of summary of 'Undo' concern in JHotDraw.

1.2.3 Related Work

While our focus is on summarizing concern code which often crosscuts modules defined in the code, other efforts have investigated summarization for localized code, e.g. summarizing a class, a method, or a package. Examples include Haiduc and colleagues [41] who generate term-based summaries for methods and classes consisting of a set of the most relevant terms to describe the class or method. Sridhara and colleagues [95] proposed a technique to generate descriptive natural language summary comments for an arbitrary Java method by exploiting structural and natural language clues in the method. As opposed to our approach, none of this work has evaluated the generated summaries in the context of a software task.

1.3 Multi-document Summarization of Software Artifacts

A software artifact is often inter-related to other software artifacts, possibly from different types (e.g., a piece of code related to a bug report) or at different levels of abstraction (e.g., a design diagram related to a requirement document).

Sometimes, a developer has to navigate a network of inter-related software artifacts to address an information need. For example, requirement documents, design diagrams and source code files might need to be investigated to understand how a certain feature has been implemented in the code. As another example, a software developer who is looking for the high-level information on the motivation behind a code change typically has to investigate a hierarchy of project documents starting from a bug report up to inter-related design and requirement documents. Finding and understanding motivational information that is spread across multiple natural language documents is time consuming and cognitively demanding and thus seldom happens in the context of a code change.

We hypothesize that a concise summary of documents related to the reason behind a code change can make it easier for a developer to understand why code was changed in a certain way. We see this problem as a special case of multi-document summarization of natural language software artifacts. Thus we propose the use of multi-document extractive summarization techniques, previously applied to generic natural language documents (e.g., [79]).

1.3.1 Approach

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 (e.g., [49, 105]). Our approach is different in that it investigates summarizing a set of documents each at a different level of abstraction. We modeled the set of documents as a hierarchical chain to account for the fact that they are at different

```
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)
--
Create a pilot implementation for streaming large files for the
Document Submission service
--
When an end to end run of document submission is run, no
temporary files should exist in the file system after the
transaction has been completed.
```

Figure 1.5: A summary describing the reason behind a code change produced for a collection of project documents related to the code change.

levels of abstraction. We took a supervised approach to extract the most important sentences. We created a corpus by asking human annotators to create summaries for 8 different chains. We then trained a classifier on the corpus based on eight sentence-level features we identified. Figure 1.5 shows a summary created by this classifier for a collection of project documents related to a code change. In Section 5.2, we discuss how such a summary can help a developer understand the reason behind a code change.

1.3.2 Evaluation

To evaluate whether generated summaries help developers in understanding the motivation behind a code change, we conducted an exploratory user study. We generated summaries for a few change-related chains and asked the developer who had made the change to evaluate if the summary described the motivation behind the code change. The developers participating in the study found the summaries to contain relevant information about the reason behind the code changes and suggested improving the summaries by including more technical details.

1.3.3 Related Work

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., [34]), impact analysis (e.g., [76]) and visualization (e.g., [103]) 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.

1.4 Contributions

The work presented in this dissertation makes the following contributions to the field of software engineering:

- It demonstrates end-to-end that reasonably accurate natural language summaries can be automatically produced for different types of software artifacts: bug reports (as an example of mostly natural language software artifacts), software concerns (as an example of mostly structured software artifacts) and a chain of documents related to a code change (as an example of inter-related natural language software artifacts) and that the generated summaries are useful for developers.
- It presents approaches developed to automatically summarize different types of software artifacts:
  - An extractive summarization approach to summarize bug reports based on an existing supervised summarization system for conversational data.
  - A novel abstractive summarization approach to summarize crosscutting code concerns.
  - A supervised extractive summarization approach to summarize a chain of project documents related to a code change based on various features burrowed from multi-document summarization.
- It demonstrates that summaries generated by the proposed summarization approaches help developers in performing software tasks:
  - It demonstrates that the generated bug report summaries help developers in the context of a particular software task, bug report duplicate detection.
  - It demonstrates that the generated concern summaries help developers in performing code change tasks.

– It provides initial evidence that developers find summaries generated for a chain of project documents related to a code change to be indicative of the motivation behind the code change.

1.5 Organization

In Chapter 2 we give an overview of background information on summarization systems and related work on summarization of software artifacts.

In the remainder of the dissertation, we discuss the approaches taken to summarize each different type of software artifact and how the goodness of summaries were evaluated in each case. In Chapter 3 we present the details of the technique used in summarizing bug reports, including the conversation-based supervised summarization framework used to train three different classifier using three different datasets (Section 3.3) along with the process of creating the bug report corpus (Section 3.2). We describe how three different classifiers are evaluated by comparing their generated summaries against human-generated summaries (Section 3.4). We describe the process of using human judges to evaluate the generated summaries by asking them to rank the goodness of summaries against the original bug reports (Section 3.5.1). We provide details of our task-based human evaluation in which we investigated whether provision of bug report summaries helps participants in performing bug report duplicate detection tasks (Section 3.5.2).

In Chapter 4 we discuss the summarization of crosscutting code concerns. We present the details of the abstractive summarization approach (Section 4.3) including steps to extract information and build a semantic representation of the concern, find patterns and salient code elements and generate summary sentences. We describe the task-based user study conducted to evaluate whether concern summaries help developers in performing software change tasks (Section 4.4).

In Chapter 5 we discuss the multi-document summarization of a set of documents related to a code change. We discuss sentence-level features used to identify summary sentences (Section 5.3) and the corpus annotated as the training set (Section 5.4). We also provide details of the human evaluation conducted to investigate if developers find generated summaries of potential

use (Section 5.5).

In Chapter 6 we provide a discussion of various decisions made in the development and evaluation of various summarization techniques and various future research directions. In Chapter 7, we summarize the contributions of the work described in this dissertation.

Chapter 2

Background and Related Work

With electronically available information growing at an exponential pace, there is substantial interest and a substantial body of work on automatic summarization systems. Such systems are designed to alleviate the problem of information overload by providing a concise natural language summary containing the most important information in one or more documents. Over the recent years, numerous summarization systems have been developed for various documents including news (e.g., [63, 80]), medical information (e.g., [27, 28]), email threads (e.g., [38, 81]), meeting transcripts (e.g., [68, 70]), scientific articles (e.g., [64, 77]), etc. In Section 2.1, we present a general overview of summarization approaches.

Similarly, in the development of a software system, large amounts of new information in the form of different software artifacts are produced on a continuous basis. Source code, bug reports, documentation, mailing list discussions, wiki entries, etc., are created by developers of the software system on a daily basis. As a result, over the past five years there has been an interest in automatic generation of summaries for various software artifacts in the software engineering community. We give an overview of these techniques in Section 2.2.

Summarization of software artifacts is an example of application of text analysis techniques to assist software developers in performing software tasks. Other efforts have considered the use of text analysis to help developers in tasks like traceability, concept location, code search and navigation, duplicate bug report detection, etc. We discuss some of the applications of text analysis techniques in software engineering research in Section 2.3.

2.1 Automatic Summarization

In this section, we present a general overview of background information on automatic summarization systems.

2.1.1 Extractive vs. Abstractive Summarization

Producing a concise and fluent abstract emulating a human-written summary requires semantic interpretation of input documents and the ability to modify and merge information, which is beyond the state of the art for automatic summarization of natural language documents [72]. Most existing summarization approaches rely on extractive techniques where the main focus is on identifying important sentences that should be included in the summary.

In extractive summarization, the task of selecting important sentences can be represented as a binary classification problem, partitioning all sentences in the input into summary and non-summary sentences. One way to distinguish between various extractive summarization approaches is based on whether they use unsupervised or supervised classification techniques. The advantage of using an unsupervised technique is that there is no need for a human annotated corpus. Examples of unsupervised summarization systems include topic representation approaches that rely on representing input documents by the topics discussed in the text (e.g., [55, 94]). Another example is graph-based methods (e.g. [30, 66]) in which input text is represented as a graph of inter-related sentences and the importance of each sentence is derived from its centrality in the graph. The main drawback of unsupervised methods is their inability to use any number of features. To tackle this problem, Kupiec and colleagues [50] proposed the use of supervised machine learning for summarization to provide the freedom to use and combine any desired number of features. A supervised approach relies on the availability of a document/summary corpus. The statistical analysis of the corpus determines how features should be weighted relative to each other. Length of a sentence, the position of a sentence in text and the number of topic words in a sentence are all examples of features.

Extractive summarization has several drawbacks both in terms of con-

tent and linguistic quality. Because full sentences are taken out of input documents to form a summary, unnecessary details may be added along with salient information. Also since sentences are taken out of context, extractive summaries usually lack coherence and referential clarity. To address these problems, there has recently been a focus on moving towards the alternative approach of abstractive summarization. Sentence fusion (e.g., [8]) and compression (e.g., [100]) have been introduced to improve the content of extractive summaries by focusing on rewriting techniques. However, a *fully abstractive* approach involves an intermediate step of building a semantic representation based on input information, then selecting salient content and finally generation of sentences. An abstractive summarization system can be categorized as a *text-to-text* or *data-to-text* generation approach. Data-to-text generation approaches have been used to summarize weather forecast [85], engineering [109] and medical data [75].

In this dissertation, we used supervised extractive summarization approaches to summarize natural language software artifacts (Chapter 3 and Chapter 5). We used an abstractive data-to-text approach to summarize code concerns (as an example of structured software artifacts) because the highly structured nature of software code enabled us to build an intermediate semantic representation (Chapter 4).

2.1.2 Single Document vs. Multi-document Summarization

While single document summarization systems deal with producing a summary of *one* document, multi-document summarization involves producing a summary of a collection of (related) documents. Multi-document summarization is typically motivated by the case of summarizing a collection of news articles covering the same event of interest [39]. Consequently, most multi-document summarization system rely on content similarity among documents. Different approaches have considered extractive multi-document summarization. Examples include centroid-based (e.g., MEAD [79]), cluster-based (e.g., [105]) and graph-based (e.g., LexPageRank [29]) summarization techniques. A centroid is a set of words that are statistically important to a collection of documents. In MEAD, a centroid is defined as a pseudo-document which consists of words with TF-IDF scores

above a pre-defined threshold [79]. The idea is that sentences that are more similar to the centroid are more important as they are more representative of the topic of the input documents. In cluster-based techniques, first clusters of similar sentences are formed and then clusters, treated as topics, are ranked and as the final step one representative sentence is selected form each main cluster. In graph-based techniques, input documents are represented as a graph of sentences that are related to each other. Each sentence is scored based on its centrality in the graph (e.g. sentences that are similar to many other sentences [29]).

Our approach to summarize a collection of natural language software artifacts (Chapter 5) is different from most other studied cases of multi-document summarization as we investigated cases where there is hierarchical relationships between the artifacts in the input collection and less degree of redundancy in terms of content.

2.1.3 Generic vs. Domain-Specific Summarization

Generic summarization does not make any assumption about the content, structure or any other characteristic of input documents. On the other hand, domain-specific summarization deals with documents with particular characteristics (e.g., input documents with particular format or content) and utilizes these characteristics to more accurately identify important information. Examples include summarization of conversational data (e.g., [19, 70]), scientific articles (e.g., [77]) and medical documents (e.g., [3]).

Research in the summarization of conversational data [17] ranges from summarization of email threads (e.g., [38, 81, 104]) to summarization of meetings (e.g., [70]) and phone conversations (e.g., [111]). Using the conversational features to summarize such data was first proposed by Rambow and colleagues [81]. They showed that using their supervised machine learning approach, best results were achieved when conversational features related to an email thread (e.g., the number of recipients) are added to features used by a generic text summarizer. Carenini and colleagues [38] proposed the use of a novel graph-based representations of an email thread capturing how each individual email mentions other emails. *Clue words* are defined as reoccurring words in adjacent emails in the graph and sentences contain-

ing more clue words are scored higher. We used an idea similar to clue words in multi-document summarization of natural language software artifacts (Chapter 5) by using the overlap between different artifacts to identify important sentences.

In a later work, Murray and Carenini [67] developed a summarizer for conversations in various modalities that uses features inherent to all multi-party conversations. They applied this system to meetings and emails and found that the generic conversation summarizer was competitive with state-of-the-art summarizers designed specifically for meetings or emails. We used this general conversation summarizer to generate summaries of bug report (as our studied case of natural language software artifacts) in Chapter 3. All the approaches proposed in this dissertation to summarize software artifacts fall under the category of domain-specific summarization as they make use of certain structure or particular content of input data.

2.1.4 Evaluation Methods

Methods for evaluating the goodness of automatically generated summaries can be categorized either as *intrinsic* or *extrinsic*. In *intrinsic* evaluation techniques, a summary is evaluated in its own right, typically by being compared to a reference set of model summaries. A model summary is often a human-generated summary or a baseline summary (e.g., the first few sentences of a news article). In *extrinsic* evaluation techniques, it is investigated whether summaries help an end user (or a tool) perform a task better.

Intrinsic evaluation involves computing analytical measures like precision, recall, pyramid precision [73] and ROUGE [54]. Extrinsic evaluation often involves a task-based human evaluation to establish that summarization systems are indeed helpful in a the context of a real-life task. Examples of earlier efforts include TIPSTER Text Summarization Evaluation (SUMMAC) [58] in which summaries were shown to be helpful in the context of the task of judging if a particular document is relevant to a topic of interest. McKeown and colleagues [62] conducted a user study in which participants were given a task of writing reports on specific topics. They found that when summaries were provided people tended to write better reports and

reported higher satisfaction.

In this dissertation, we used both intrinsic and extrinsic evaluation techniques to assess the summaries generated for software artifacts. We used analytical measures of precision, recall and pyramid precision to intrinsically evaluate bug report summaries against the gold standard summaries in our bug report corpus (Section 3.4). We conducted three user studies to extrinsically evaluate software artifact summaries. Bug reports summaries were evaluated in the context of bug report duplicate detection tasks (Section 3.5.2). Crosscutting code concern summaries were evaluated in the context of code modification tasks (Section 4.4). Multi-document summaries of project documents were evaluated by asking developers participating in the study whether each summary describes the reason behind the corresponding code change (Section 5.5).

2.2 Summarization for Software Artifacts

Existing approaches for summarizing software artifacts have mostly focused on bug reports or source code. Since the publication of our work on summarizing bug reports [84] (Chapter 3), two separate approaches, both based on unsupervised summarization techniques, have been proposed. Lotufo and colleagues [56] investigated an unsupervised approach based on the PageRank algorithm for extractive summarization of bug reports. With their approach, sentences in a bug report are ranked based on such features as whether the sentence discusses a frequently discussed topic or whether the sentence is similar to the bug reports title and description. Their paper reports on an analytic evaluation that shows a 12% improvement over our supervised summarization approach. In a separate work, Mani and colleagues [59] applied four unsupervised summarization approaches to bug reports. They showed that three of these approaches, *MMR*, *DivRank* and *Grasshopper* algorithms, worked at par with our proposed supervised approach. None of these approaches have been subjected to a human task-based evaluation.

Other work on summarizing software artifacts has focused on producing summaries of source code. Haiduc and colleagues [41] used techniques

based on Vector Space Model (VSM) and Latent Semantic Indexing (LSI) to generate term-based summaries for methods and classes. Such a summary contains a set of the most relevant terms to describe a class or method. Sridhara and colleagues [95] proposed a technique to generate descriptive natural language summary comments for an arbitrary Java method by exploiting structural and natural language clues in the method. As the first step, they choose the important or central code statements to be included in the summary comment. Then, for a selected code statement, they use text generation to determine how to express the content in natural language phrases and how to smooth between the phrases and mitigate redundancy. Our approach for summarizing code (Chapter 4) differs in targeting automated summarization for non-contiguous blocks of code, namely crosscutting code for a concern. We also applied a task-based evaluation rather than relying on human judges of summaries out of the context of use of a summary.

2.3 Text Analysis in Software Engineering

The natural language content of software artifacts provides semantic information necessary to develop and maintain a software system. Consequently, various efforts have considered the use of text analysis to make it easier for developers to benefit from the natural language information contained in various software artifacts. Text analysis borrows techniques from different areas including information retrieval (IR), natural language processing (NLP), machine learning and the Semantic Web. It also can be integrated with various static and dynamic source code analysis techniques. Text analysis techniques have been used to assist developers in performing various software tasks including concern location [93], source code search and navigation [43], traceability [60], duplicate bug report detection [91] and specification extraction [110]. In this section we mainly focus on approaches aimed at extracting semantic information from natural language content of software artifacts (code in particular) since the same techniques can be used in abstractive summarization of software artifacts.

Various techniques have been used to extract semantic information from

natural language content of the code. For example, Tan and colleagues [98] used a combination of part of speech (POS) tagging, semantic role labeling and clustering to extract information from *rule-containing* comments. An example of such a comment is one that requires a lock to be acquired before calling a function. Zhong and colleagues [110] inferred specifications from API documentations by using Named Entity Recognition and chunk tagging to extract (*resource, action*) pairs. The extracted information was then used to locate bugs in the code. Fry and colleagues [37] used POS tagging, chunking and pattern matching to analyze comments and identifiers of a method and extract *Verb-DO (Direct Object)* pairs. This extracted information then was used to identify action-oriented concerns [93] and to generate summary comments for methods [95]. Hill and colleagues [44] extended the Verb-DO extraction technique to extract all phrasal concepts (e.g., noun phrases or propositional phrases) to capture word context of natural language queries and make code searches more efficient. Similar text analysis techniques can be used to extract semantic information from code to be included in a concern summary. For example, we used Verb-DO extraction to identify the feature implemented by the concern (Section 4.3).

Chapter 3

Summarization of Bug Reports

We studied bug report summarization to explore the problem of summarizing software artifacts with mostly natural language content. A software project's bug repository provides a rich source of information for a software developer working on the project. For instance, the developer may consult the repository to understand reported defects in more details, to understand how changes were made on the project in the past, or to communicate with other developers or stakeholders involved in the project [9].

In this chapter, we investigate whether concise summaries of bug reports, automatically produced from a complete bug report, would allow a developer to more efficiently investigate information in a bug repository as part of a task. Our approach is based on an existing extraction-based supervised summarization system for conversational data [67]. Using this summarization system, a classifier trained on a human-annotated corpus can be used to generate summaries of bug reports. We start by motivating the need for summarization of bug reports in Section 3.1. In Section 3.2, we discuss the human-annotated bug report corpus we created. In Section 3.3, we present details of our summarization approach. We discuss the evaluation results in Section 3.4 (analytic evaluation) and Section 3.5 (human evaluations).

3.1 Motivation

When accessing the project's bug repository, a developer often ends up looking through a number of bug reports, either as the result of a search or a recommendation engine (e.g., [96, 102]). Typically, only a few of the bug reports a developer must peruse are relevant to the task at hand. Some-

Table 3.1: Statistics on summary-worthy bug reports for several open source projects, computed for bug reports created over the 24-month period of 2011-2012.

Project	#all bug reports	#bug reports longer than 300 words
Eclipse Platform	7,641	2,382 (32%)
Firefox	10,328	3,310 (32%)
Thunderbird	6,225	2,421 (39%)

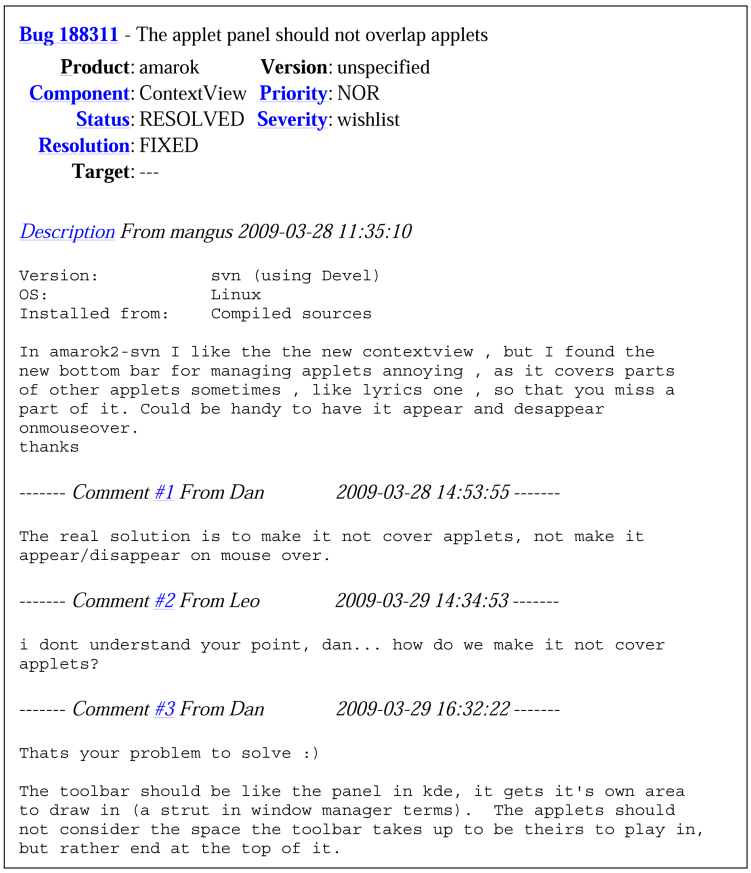
times a developer can determine relevance based on a quick read of the title of the bug report, other times a developer must read the report, which can be lengthy, involving discussions amongst multiple team members and other stakeholders. For example, a developer using the bug report duplicate recommender built by Sun and colleagues [96] to get a list of potential duplicates for bug #564243 from the Mozilla system[4], is presented with a total of 5125 words (237 sentences) in the top six bug reports on the recommendation list.

Perhaps optimally, when a bug report is closed, its authors would write a concise summary that represents information in the report to help other developers who later access the report. Given the evolving nature of bug repositories and the limited time available to developers, this optimal path is unlikely to occur. As a result, we investigate the automatic production of summaries to enable generation of up-to-date summaries on-demand and at a low cost.

Bug reports vary in length. Some are short, consisting of only a few words. Others are lengthy and include conversations between many developers and users. Figure 3.1 displays part of a bug report from the KDE bug repository[5]; the entire report consists of 21 comments from 6 people.

Developers may benefit from summaries of lengthy bug reports but are unlikely to benefit from summaries of short bug reports. If we target summaries of 100 words, a common size requirement for short paragraph-length summaries [74], and we assume a compression rate of 33% or less is likely beneficial, then a bug report must be at least 300 words in length to achieve the 33% compression rate. Table 3.1 shows the number and percentage of

[4] `www.mozilla.org`, verified 04/04/12
[5] `bugs.kde.org`, verified 04/04/12



Bug 188311 - The applet panel should not overlap applets

Product: amarok **Version**: unspecified
Component: ContextView **Priority**: NOR
Status: RESOLVED **Severity**: wishlist
Resolution: FIXED
Target: ---

Description From mangus 2009-03-28 11:35:10

```
Version:           svn (using Devel)
OS:                Linux
Installed from:    Compiled sources

In amarok2-svn I like the the new contextview , but I found the
new bottom bar for managing applets annoying , as it covers parts
of other applets sometimes , like lyrics one , so that you miss a
part of it. Could be handy to have it appear and desappear
onmouseover.
thanks
```

------- *Comment #1 From Dan* *2009-03-28 14:53:55* -------

```
The real solution is to make it not cover applets, not make it
appear/disappear on mouse over.
```

------- *Comment #2 From Leo* *2009-03-29 14:34:53* -------

```
i dont understand your point, dan... how do we make it not cover
applets?
```

------- *Comment #3 From Dan* *2009-03-29 16:32:22* -------

```
Thats your problem to solve :)

The toolbar should be like the panel in kde, it gets it's own area
to draw in (a strut in window manager terms).  The applets should
not consider the space the toolbar takes up to be theirs to play in,
but rather end at the top of it.
```

Figure 3.1: The beginning part of bug #188311 from the KDE bug repository.

summary worthy bug reports in three popular and large-scale open source software projects, computed over the two-year period of 2011-2012. In all these projects, almost one third of bug reports created over the 24-month period are longer than 300 words, suggesting that sufficient lengthy bug reports exist to make the automatic production of summaries worthwhile. The length of a bug report is the total number of words in its description and comments.

Many existing text summarizing approaches exist that could be used to generate summaries of bug reports. Given the strong similarity between bug reports and other conversational data (e.g., email and meeting discus-

sions), we chose to investigate whether an existing supervised approach for generating extractive summaries for conversation-based data [67] can produce accurate summaries for bug reports. The conversational nature of bug reports (Figure) plays an important role in facilitating communication involved in resolving bugs, both in open source projects [92] and collocated teams [9].

3.2 Bug Report Corpus

To be able to train, and judge the effectiveness, of an extractive summarizer on bug reports, we need a corpus of bug reports with good summaries. Optimally, we would have available such a corpus in which the summaries were created by those involved with the bug report, as the knowledge of these individuals in the system and the bug should be the best available. Unfortunately, such a corpus is not available as developers do not spend time writing summaries once a bug is complete, despite the fact that the bug report may be read and referred to in the future.

To provide a suitable corpus, we recruited ten graduate students from the Department of Computer Science at the University of British Columbia to annotate a collection of bug reports. On average, the annotators had seven years of programming experience. Half of the annotators had experience programming in industry and four had some experience working with bug reports.

3.2.1 Annotation Process

We had each individual annotate a subset of bugs from four different open-source software projects: Eclipse Platform, Gnome,[6], Mozilla and KDE. We chose a diverse set of systems because our goal is to develop a summarization approach that can produce accurate results for a wide range of bug repositories, not just bug reports specific to a single project. There are a total of 2361 sentences in these 36 bug reports. This corpus size is comparable to the size of the corpi in other domains used in training similar classifiers.

[6] `www.gnome.org`, verified 04/04/12

For example, the Enron email corpus, used to train a classifier to summarize email threads, contains 39 email threads and 1400 sentences [67].

The 36 bug reports (nine from each project) have been chosen randomly from the pool of all bug reports containing between 5 and 25 comments with two or more people contributing to the conversation. The bug reports have mostly conversational content. We avoided selecting bug reports consisting mostly of long stack traces and large chunks of code as we are targeting bug reports with mainly natural language content. The reports chosen varied in length: 13 (36%) reports had between 300 and 600 words; 14 (39%) reports had 600 to 900 words; 6 (17%) reports had between 900 and 1200 words; the remaining three (8%) bugs were between 1200 and 1500 words in length. The bug reports also had different numbers of comments: 14 reports (39%) had between five and nine comments; 11 (31%) reports had 10 to 14 comments; 5 (14%) reports had between 15 to 19 comments; the remaining six (16%) reports had 20 to 25 comments each. The reports also varied when it came to the number of people who contributed to the conversation in the bug report: 16 (44%) bug reports had between 2 and 4 contributors; 16 (44%) other had 5 to 7 contributors; the remaining four (12%) had 8 to 12 contributors. Nine of the 36 bug reports (25%) were enhancements to the target system; the other 27 (75%) were defects.

Each annotator was assigned a set of bug reports from those chosen from the four systems. For each bug report, we asked the annotator to write an abstractive summary of the report using their own sentences that was a maximum of 250 words. We limited the length of the abstractive summary to motivate the annotator to abstract the given report. The annotator was then asked to specify how each sentence in the abstractive summary maps (links) to one or more sentences from the original bug report by listing the numbers of mapped sentences from the original report. The motivation behind asking annotators to first write an abstractive summary (similar to the technique used to annotate the AMI meeting corpus [20]) was to make sure they had a good understanding of the bug report before mapping sentences. Asking them to directly pick sentences may have had the risk of annotators just selecting sentences that looked important without first reading through the bug report and trying to understand it. Although the annotators did not

Table 3.2: Abstractive summaries generated by annotators.

	mean	stdv
#sentences in the summary	5.36	2.43
#words in the summary	99.2	39.93
#linked sentences from the bug report	16.14	9.73

have experience with these specific systems, we believe their experience in programming allowed them to extract the gist of the discussions; no annotator reported being unable to understand the content of the bug reports. The annotators were compensated for their work. The annotation instructions used by the annotators has been provided in Appendix A.

To aid the annotators with this process, the annotators used a version of BC3 web-based annotation software[7] that made it easier for them to manipulate the sentences of the bug report. Figure 3.2 shows an example of part of an annotated bug report; the summary at the top is an abstractive summary written by an annotator with the mapping to the sentences from the original bug report marked.

The annotated bug report corpus is publicly available.[8]

3.2.2 Annotated Bugs

On average, the bug reports being summarized comprised 65 sentences. On average, the abstractive summaries created by the annotators comprised just over five sentences with each sentence in the abstractive summaries linked (on average) to three sentences in the original bug report. Table 3.2 provides some overall statistics on the summaries produced by the annotators.

A common problem of annotation is that annotators often do not agree on the same summary. This reflects the fact that the summarization is a subjective process and there is no single best summary for a document—a bug report in this paper. To mitigate this problem, we assigned three annotators to each bug report. We use the kappa test to measure the level

[7] `www.cs.ubc.ca/nest/lci/bc3/framework.html`, verified 04/04/12

[8] See `www.cs.ubc.ca/labs/spl/projects/summarization.html`. The corpus contains additional annotations, including an extractive summary for each bug report and labeling of the sentences.

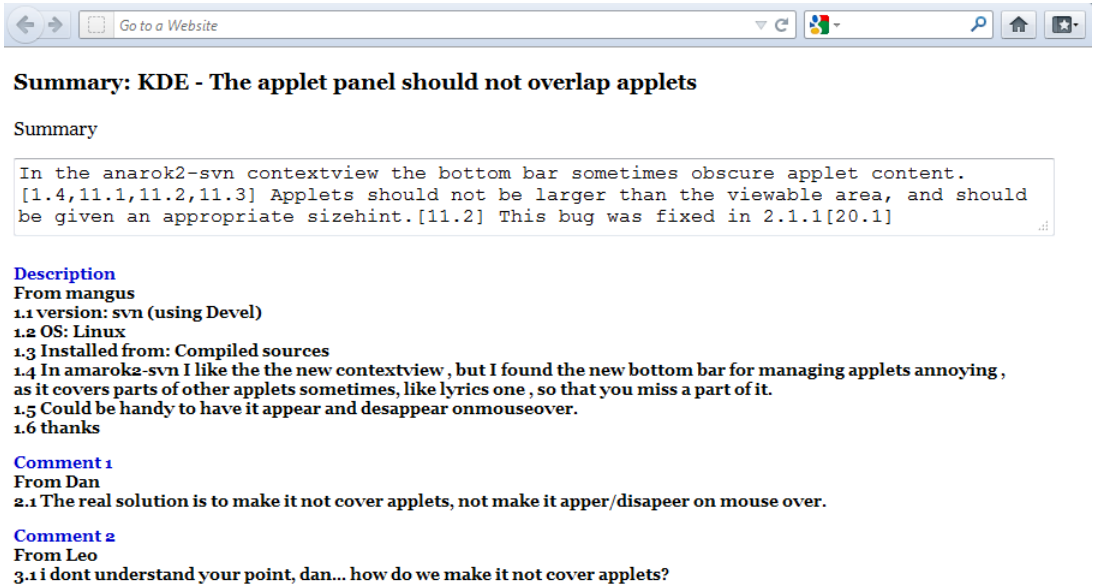


Figure 3.2: A screenshot of the annotation software. The bug report has been broken down into labeled sentences. The annotator enters the abstractive summary in the text box. The numbers in the brackets are sentence labels and serve as links between the abstractive summary and the bug report. For example, the first sentence of the abstractive summary has links to sentences 1.4, 11.1, 11.2, 11.3 from the bug report.

of agreement amongst the annotators with regards to bug report sentences that they linked in their abstractive summaries [33]. The result of the kappa test (k value) is 0.41 for our bug report annotations, showing a moderate level of agreement.

Table 3.3: The questions asked from an annotator after annotating a bug report.

Question	Average
What was the level of difficulty of summarizing the bug report?	2.67 (±0.86)
What was the amount of irrelevant and off-topic discussion in the bug report?	2.11 (±0.66)
What was the level of project-specific terminology used in the bug report?	2.68 (±0.83)

We asked each annotator, at the end of annotating each bug report, to answer a number of questions (first column in Table 3.3) about properties of the report. They answered each question using a scale of 1 to 5 (with 1 low, 3 medium and 5 high). The second column of Table 3.3 shows the average score computed across all the annotated bug reports. These scores (with standard deviation taken into account) indicate that the annotators did not find the bug reports to be difficult to summarize, to consist of a lot of off-topic discussion or to include much project-specific terminology. Using an ordinal scale, we have not accounted for the subjectivity of human opinions. For example, one annotator's perception of off-topic discussion in bug reports might be different from the perception of another annotator. A more qualitative study is needed to understand how different people approach an annotation (summarization) task and what might make the task difficult for them.

3.3 Approach

The bug report corpus provides a basis on which to experiment with producing bug report summaries automatically. We produce summaries using binary classifiers that consider 24 sentence features (Section 3.3.1). It is

based on values of these features, computed for each sentence, that it is determined whether the sentence should be included in the summary. To assign a weight to each feature, a classifier first has to be trained on human generated summaries.

We set out to investigate two questions:

1. Can we produce good summaries with existing conversation-based classifiers?

2. Can we do better with a classifier specifically trained on bug reports?

The existing conversation-based classifiers we chose to investigate are trained on conversational data other than bug reports. The first classifier, which we refer to as EC(Email Classifier), was trained on email threads [67]. We chose this classifier as bug report conversations share similarity with email threads, such as being multi-party and having thread items added at differing intervals of time. This classifier was trained on a subset of the publicly available Enron email corpus [47], which consists of 39 annotated email threads (1400 sentences in total).

The second classifier, which we refer to as EMC(Email & Meeting Classifier), was trained on a combination of email threads and meetings [67]. We chose this classifier because some of the characteristics of bug reports might be more similar to meetings, such as having concluding comments at the end of the conversation. The meetings part of the training set for EMC is a subset of the publicly available AMI meeting corpus [20], which includes 196 meetings.

The EC and EMC classifiers are appealing to use because of their generality. If these classifiers work well for bug reports, it offers hope that other general classifiers might be applicable to software project artifacts without training on each specific kind of software artifacts (which can vary between projects) or on project-specific artifacts, lowering the cost of producing summaries.

However, unless these classifiers produce perfect summaries, the question of how good of a summary can be produced for bug reports remains open unless we consider a classifier trained on bug reports. Thus, we also chose to train a third classifier, BRC(Bug Report Classifier), using the bug report

SUMMARY: The applet panel should not overlap applets

In amarok2-svn I like the the new contextview , but I found the new bottom bar for managing applets annoying , as it covers parts of other applets sometimes , like lyrics one , so that you miss a part of it.
Could be handy to have it appear and disappear onmouseover.

The applet should end where the toolbar begins.
Applets should not be larger than the viewable area, if there's an applet above it, then the lower applet should get a smaller sizehint, and resize if necessary when it's the active applet (and therefore the only one on the screen)
Basically, no applet should continue on off the screen, it should end at the panel.

The bug that is being shown here is the fact that you cannot yet resize your applets, and as such we also don't set default sizes sanely.
You are reporting a bug on a non-completed feature ;)

will be fixed in 2.1.1, done locally.

Figure 3.3: The gold standard summary for bug #188311 from the KDE bug repository. The summary was formed by extracting sentences that were linked by two or three human annotators.

corpus we created. To form the training set for *BRC*, we combined the three human annotations for each bug report by scoring each sentence of a report based on the number of times it has been linked by annotators. For each sentence, the score is between zero, when it has not been linked by any annotator, and three, when all three annotators have a link to the sentence in their abstractive summary. A sentence is considered to be part of the extractive summary if it has a score of two or more. For each bug report, the set of sentences with a score of two or more (a positive sentence) is called the *gold standard summary*. For the bug report corpus, gold standard summaries include 465 sentences, which is 19.7% of all the sentences in the corpus, and 28.3% of all words in the corpus. Figure 3.3 shows the gold standard summary for bug 188311 from the KDE bug repository, a portion of the original bug appears earlier in Figure 3.1.

As we have only the bug report corpus available for both training and testing the bug report classifier, we use a cross-validation technique when evaluating this classifier. Specifically, we use a leave-one-out procedure so that the classifier used to create a summary for a particular bug report is trained on the remainder of the bug report corpus.

All three classifiers investigated are logistic regression classifiers. Instead of generating an output of zero or one, these classifiers generate the prob-

ability of each sentence being part of an extractive summary. To form the summary, we sort the sentences into a list based on their probability values in descending order. Starting from the beginning of this list, we select sentences until we reach 25% of the bug report word count.[9] The selected sentences form the generated extractive summary. We chose to target summaries of 25% of the bug report word count because this value is close to the word count percentage of gold standard summaries (28.3%). All three classifiers were implemented using the Liblinear toolkit [31].[10]

3.3.1 Conversation Features

The classifier framework used to implement EM, EMC and BRC learn based on the same set of 24 different features. The values of these features for each sentence are used to compute the probability of the sentence being part of the summary.

The 24 features can be categorized into four major groups.

- *Structural* features are related to the conversational structure of the bug reports. Examples include the position of the sentence in the comment and the position of the sentence in the bug report.
- *Participant* features are directly related to the conversation participants. For example if the sentence is made by the same person who filed the bug report.
- *Length* features include the length of the sentence normalized by the length of the longest sentence in the comment and also normalized by the length of the longest sentence in the bug report.
- *Lexical* features are related to the occurrence of unique words in the sentence.

Table 3.4 provides a short description of the features considered. Some descriptions in the table refer to *Sprob*. Informally, *Sprob* provides the probability of a word being uttered by a particular participant based on the

[9] A sentence is selected as the last sentence of the summary if the 25% length threshold is reached in the middle or at the end of it.

[10] `www.csie.ntu.edu.tw/~cjlin/liblinear/`, verified 04/04/12

intuition that certain words will tend to be associated with one conversation participant due to interests and expertise. Other descriptions refer to *Tprob*, which is the probability of a turn given a word, reflecting the intuition that certain words will tend to cluster in a small number of turns because of shifting topics in a conversation. Full details on the features are provided in [67].

To see which features are informative for generating summaries of bug reports, we perform a feature selection analysis. For this analysis, we compute the F statistics score (introduced by Chen and Lin [21]) for each of the 24 features using the data in the bug report corpus. This score is commonly used to compute the discriminability of features in supervised machine learning. Features with higher F statistics scores are the most informative in discriminating between important sentences, which should be included in the summary, and other sentences, which need not be included in the summary.

Figure 3.4 shows the values of F statistics computed for all the features defined in Table 3.4. The results show that the length features (SLEN & SLEN2) are among the most helpful features. Several lexical features are also helpful: CWS[11], CENT1, CENT2[12], SMS[13] & SMT[14]. Some features have very low F statistics because either each sentence by a participant gets the same feature value (e.g., BEGAUTH) or each sentence in a turn gets the same feature value (e.g., TPOSE1). Although a particular feature may have a low F statistics score because it does not discriminate informative versus non-informative sentences on its own, it may well be useful in conjunction with other features [67].

The distribution of F statistics scores for the bug report corpus is different from those of the meeting and email corpi [67]. For example MXS and MXT have a relatively high value of F statistics for the email data while both have a relatively low value of F statistics for the bug report

[11] CWS measures the cohesion of the conversation by comparing the sentence to other turns of the conversation.

[12] CENT1 & CENT2 measure whether the sentence is similar to the conversation overall.

[13] SMS measures whether the sentence is associated with some conversation participants more than the others.

[14] SMT measures whether the sentence is associated with a small number of turns more than the others.

Table 3.4: Features key.

Feature ID	Description
MXS	max *Sprob* score
MNS	mean *Sprob* score
SMS	sum of *Sprob* scores
MXT	max *Tprob* score
MNT	mean *Tprob* score
SMT	sum of *Tprob* scores
TLOC	position in turn
CLOC	position in conversation
SLEN	word count, globally normalized
SLEN2	word count, locally normalized
TPOS1	time from beginning of conversation to turn
TPOS2	time from turn to end of conversation
DOM	participant dominance in words
COS1	cosine of conversation splits, w/ *Sprob*
COS2	cosine of conversation splits, w/ *Tprob*
PENT	entropy of conversation up to sentence
SENT	entropy of conversation after sentence
THISENT	entropy of current sentence
PPAU	time between current and prior turn
SPAU	time between current and next turn
BEGAUTH	is first participant (0/1)
CWS	rough ClueWordScore
CENT1	cosine of sentence & conversation, w/ *Sprob*
CENT2	cosine of sentence & conversation, w/ *Tprob*

data. Similarly SLEN2 has a relatively high F statistics score for the bug report data while it has a low value of F statistics for the meeting data. These differences further motivates training a new classifier using the bug report corpus as it may produce better results for bug reports compared to classifiers trained on meeting and email data.

3.4 Analytic Evaluation

To compare the EC, EMC and BRC classifiers, we use several measures that compare summaries generated by the classifiers to the gold standard summaries formed from the human annotation of the bug report corpus (Section 3.3). These measures assess the quality of each classifier and enable the comparison of effectiveness of the different classifiers against each other. In the next section, we report on two human evaluations conducted to investigate the usefulness of summaries generated by a classifier from a human perspective.

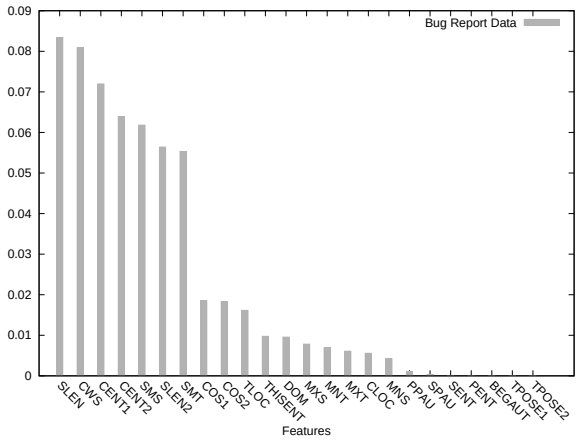


Figure 3.4: Features F statistics scores for the bug report corpus.

3.4.1 Comparing Base Effectiveness

The first comparison we consider is whether the EC, EMC and BRC classifiers are producing summaries that are better than a random classifier in which a coin toss is used to decide which sentences to include in a summary. The classifiers are compared to a random classifier to ensure they provide value in producing summaries. We perform this comparison by plotting the receiver operator characteristic (ROC) curve and then computing the area under the curve (AUROC) [32].

For this comparison we investigate different probability thresholds to generate extractive summaries. As described in Section 3.3, the output of the classifier for each sentence is a value between zero and one showing the probability of the sentence being part of the extractive summary. To plot a point of ROC curve, we first choose a probability threshold. Then we form the extractive summaries by selecting all the sentences with probability values greater than the probability threshold.

For summaries generated in this manner, we compute the false positive rate (FPR) and true positive rate (TPR), which are then plotted as a point in a graph. For each summary, TPR measures how many of the sentences present in gold standard summary (GSS) are actually chosen by the classifier.

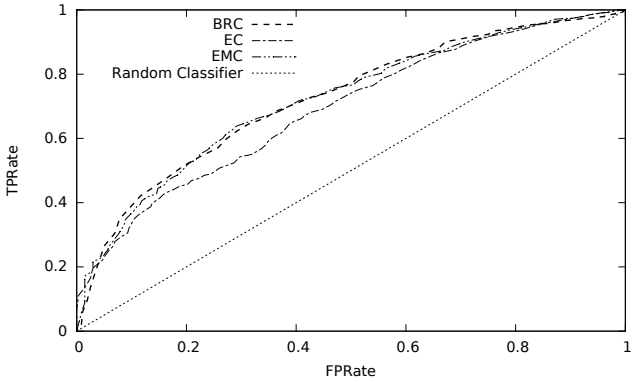


Figure 3.5: ROC plots for BRC, EC and EMC classifiers.

$$TPR = \frac{\#sentences\ selected\ from\ the\ GSS}{\#sentences\ in\ GSS}$$

FPR computes the opposite.

$$FPR = \frac{\#sentences\ selected\ that\ are\ not\ in\ the\ GSS}{\#sentences\ in\ the\ bug\ report\ that\ are\ not\ in\ the\ GSS}$$

The area under a ROC curve (AUROC) is used as a measure of the quality of a classifier. A random classifier has an AUROC value of 0.5, while a perfect classifier has an AUROC value of 1. Therefore, to be considered effective, a classifier's AUROC value should be greater than 0.5, preferably close to 1.

Figure 3.5 shows the ROC curves for all the three classifiers. The diagonal line is representative of a random classifier. The area under the curve (AUROC) for BRC, EC and EMC is equal to 0.723, 0.691 and 0.721 respectively, indicating that all these classifiers provide comparable levels of improvement in efficiency over a random classifier.

3.4.2 Comparing Classifiers

To investigate whether any of EC, EMC or BRC work better than the other two based on our desired 25% word count summaries, we compared

them using the standard evaluation measures of precision, recall, and f-score. We also used pyramid precision, which is a normalized evaluation measure taking into account the multiple annotations available for each bug report.

Precision, recall and f-score

F-score combines the values of two other evaluation measures: precision and recall. Precision measures how often a classifier chooses a sentence from the gold standard summaries (GSS) and is computed as follows.

$$precision = \frac{\#sentences\ selected\ from\ the\ GSS}{\#selected\ sentences}$$

Recall measures how many of the sentences present in a gold standard summary are actually chosen by the classifier. For a bug report summary, the recall is the same as the TPR used in plotting ROC curves (Section 3.4.1).

As there is always a trade-off between precision and recall, the F-score is used as an overall measure.

$$F\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Pyramid Precision

The pyramid evaluation scheme by Nenkova and Passonneau [73] was developed to provide a reliable assessment of content selection quality in summarization where there are multiple annotations available. We used the pyramid precision scheme of Carenini et. al [19] inspired by Nenkova's pyramid scheme.

For each generated summary of a given length, we count the total number of times the sentences in the summary were linked by annotators. Pyramid precision is computed by dividing this number by the maximum possible total for that summary length. For example, if an annotated bug report has 4 sentences with 3 links and 5 sentences with 2 links, the best possible summary of length six consists of four sentences with 3 links and two sentences with 2 links. The total number of links for such a summary is equal to $(4 \times 3) + (2 \times 2) = 16$. The pyramid precision of an automatically

Table 3.5: Evaluation measures.

Classifier	Pyramid Precision	Precision	Recall	F-Score
BRC	.66	.57	.35	.40
EC	.55	.43	.30	.32
EMC	.54	.47	.23	.29

generated summary of length 6 with a total of 8 links is therefore computed as:

$$\textit{Pyramid Precision} = \frac{8}{16} = 0.50$$

Results

Table 3.5 shows the values of precision, recall, F-score, and pyramid precision for each classifier averaged over all the bug reports.

To investigate whether there is any statistically significant difference between the performance of the three classifiers, we performed six paired t-tests.[15] Table 3.6 shows the p-value for each individual test. These results confirm that the bug report classifier (*BRC*) out-performs the other two classifiers (*EC* and *EMC*) with statistical significance (where significance occurs with $p < .025$).[16] There is no significant difference when comparing the performance of *EC* and *EMC*.

The results obtained for the *EC* and *EMC* classifiers are similar to those produced when the same classifiers are applied to meeting and email data [67].

The results demonstrate that based on standard measures, while classifiers trained on other conversation-based data (*EC* and *EMC*) can generate reasonably good bug report summaries, a classifier specifically trained on bug report data (*BRC*) can generate summaries that are better with statistical significance.

[15] The data conforms to the t-test normality assumption.

[16] Since every two classifiers are compared based on two measures (F-score & pyramid precision), we used the Bonferroni correction [1] to adjust the confidence interval in order to account for the problem of multiple comparisons.

Table 3.6: Paired t-tests results.

Tested Measure	P-Value
Pyramid Precision	
Comparing BRC and EC	.00815
Comparing BRC and EMC	.00611
Comparing EC and EMC	.89114
F-Score	
Comparing BRC and EC	.01842
Comparing BRC and EMC	.00087
Comparing EC and EMC	.28201

3.5 Human Evaluation

The generated summaries are intended for use by software developers. Does a classifier with pyramid precision of 0.66 produce summaries that are useful for developers? To investigate whether the summaries are of sufficient quality for human use, we conducted two separate human evaluations. In the first, we evaluated the summaries generated by the BRC classifier with a group of eight human judges. In the second evaluation, we performed a task-based user study in which twelve participants performed a set of eight bug report duplicate detection tasks. For some of the tasks, the participants worked with summaries generated by the BRC classifier instead of interacting with original bug reports. We chose to focus on the BRC classifier since it had performed the best based on the earlier measures.

3.5.1 Human Judges

Eight of our ten annotators agreed to evaluate a number of machine generated summaries. We asked the eight judges to evaluate a set of eight summaries generated by the BRC classifier. Each human judge was assigned three summaries in such a way that each summary was evaluated by three different judges. The human judges were instructed to read the original bug report and the summary before starting the evaluation process. The generated extractive summaries the judges were asked to evaluate were in the format shown in Figure 1.2 and were 25% of the original bug reports in length. We asked each judge to rank, using a five-point scale with five the

highest value, each bug report summary based on four statements (mean and standard deviations are provided in parentheses following the statement):

1. The important points of the bug report are represented in the summary. (3.54 ± 1.10)
2. The summary avoids redundancy. (4.00 ± 1.25)
3. The summary does not contain unnecessary information. (3.91 ± 1.10)
4. The summary is coherent. (3.29 ± 1.16)

An evaluation of meeting data summarized by multiple approaches uses similar statements to evaluate the goodness of the generated summaries [71].

We ensured in this judging process that the bug reports were assigned to judges who had not annotated the same reports during the annotation process. We also took care to choose summaries with different values of pyramid precision and F-score so as to not choose only the best examples of generated summaries for judging. The scores suggest that, on average, human judges found summaries to be of reasonable quality. The relatively large values for standard deviation show a wide range of opinions for each question indicating both the subjectivity of human evaluation and the presence of summaries with different qualities. To determine the actual helpfulness of bug reports summaries, we conducted a task-based evaluation.

3.5.2 Task-based Evaluation

According to the conventional measures of pyramid precision and F-score and the scores given by human judges, bug report summaries generated by the *BRC* classifier are of reasonable quality. Yet still the question of whether summaries can help developers in performing software tasks remains to be answered. To investigate this question, we conducted a task-based evaluation of the usefulness of bug report summaries. The particular task we chose to investigate is bug report duplicate detection: determining whether a newly filed bug report is a duplicate of an existing report in a bug repository.

Bug report duplicate detection is performed when a new bug report is filed against a bug repository and has to be triaged and assigned to a

developer. One step of triage is deciding whether the new bug report is a duplicate of one or more already in the repository. Early determination of duplicates can add information about the context of a problem and can ensure that the same problem does not end up being assigned to multiple developers to solve. Developers use different techniques to retrieve a list of potential duplicates from the bug repository including their memory of bugs they know about in the repository, keyword searches and machine learning and information retrieval approaches (e.g., [91, 96, 106]). In any approach other than memory-based approaches, a developer is presented a list of potential duplicate reports in the repository based on search or mining results. The developer must go over the list of retrieved potential duplicate bug reports to determine which one is a duplicate of the new report; this may require significant cognitive activity on the part of the developer as it might involve reading a lot of text both in description and comments of bug reports.

Our hypothesis is that concise summaries of original bug reports can help developers save time in performing duplicate detection tasks without compromising accuracy.

Our task-based evaluation of bug report summaries involved having 12 subjects complete eight duplicate detection tasks similar to real-world tasks under two conditions: *originals* and *summaries*. Each task involved a subject reading a new bug report and deciding for each bug report on a presented list of six potential duplicates whether it is a duplicate of the new bug report or not. All the bug reports used in the study were selected from the Mozilla bug repository. We scored the accuracy of a subject's determination of duplicates against information available in the bug repository.

Experimental Method

Each subject started the study session by working on two training tasks. Then the subject was presented with eight main tasks in random order. Half (four) of these tasks were performed under *originals* condition where the subject had access to potential duplicate bug reports in their original form. The other half were performed under *summaries* condition where the subject had access to 100-word summaries of potential duplicate bug reports, but

not to their originals. Summaries were generated by the *BRC* classifier. As opposed to the 25% work count summaries used in the evaluation performed in Section 3.4, we decided to use fixed-size summaries in the task-based user evaluation to make summaries consistent in terms of size. We produced summaries of 100 words, the size of a short paragraph, to make them easy to read and interact with.

Figure 3.6 shows a screenshot of the tool used by subjects to perform duplicate detection tasks. Based on the condition a task is performed under, clicking on the title of a potential duplicate in the top left window shows its corresponding bug report in original or summarized format in the right window. The new bug report can be viewed in the bottom left window.

For each task, the subject can mark a potential duplicate as *not duplicate*, *maybe duplicate*, or *duplicate*. To complete a task, a subject has to mark all potential duplicates and provide a short explanation for any *duplicate* or *maybe duplicate* marking. At most one bug report on the list of potential duplicates can be labeled as *duplicate*. We put this restriction because, based on information in the bug repository, for each task there is either zero or one actual duplicate on the list of potential duplicates. A subject can mark as many potential duplicate bug reports on the list as *maybe duplicate* or *not duplicate*.

Six out of eight new bug reports have an actual duplicate appearing on their corresponding list of potential duplicates. Subjects were not informed of this ratio and were only told that "There may or may not be a duplicate on the recommendation list".

Subjects were recommended to limit themselves to 10 minutes per task, but the time limit was not enforced. Each study session was concluded with a semi-structured interview.

All 12 users worked on the same set of eight duplicate detection tasks. Each task was performed under each condition (*summaries*, *originals*) by 6 different users. For each task, the users to whom the task was assigned under a particular condition (e.g., *summaries*) were randomly selected.

A number of questions in the form of a semi-structured interview were asked from each subject at the end of a study session (Table 3.7). During the interview, the subjects discussed the strategy they used in identifying

Table 3.7: Questions asked of each participant at the end of a study session.

Question
1. Did you find it easier or harder to detect duplicates given summaries for the potential duplicates? Why?
2. Did summaries change time spent on determining duplicates?
3. What kind of information in general do you think would be helpful in determining duplicates?
4. Did you find summaries to contain enough information to determine if they represented a duplicate?
5. Did you use attributes (e.g., version, platform, etc.)? How?

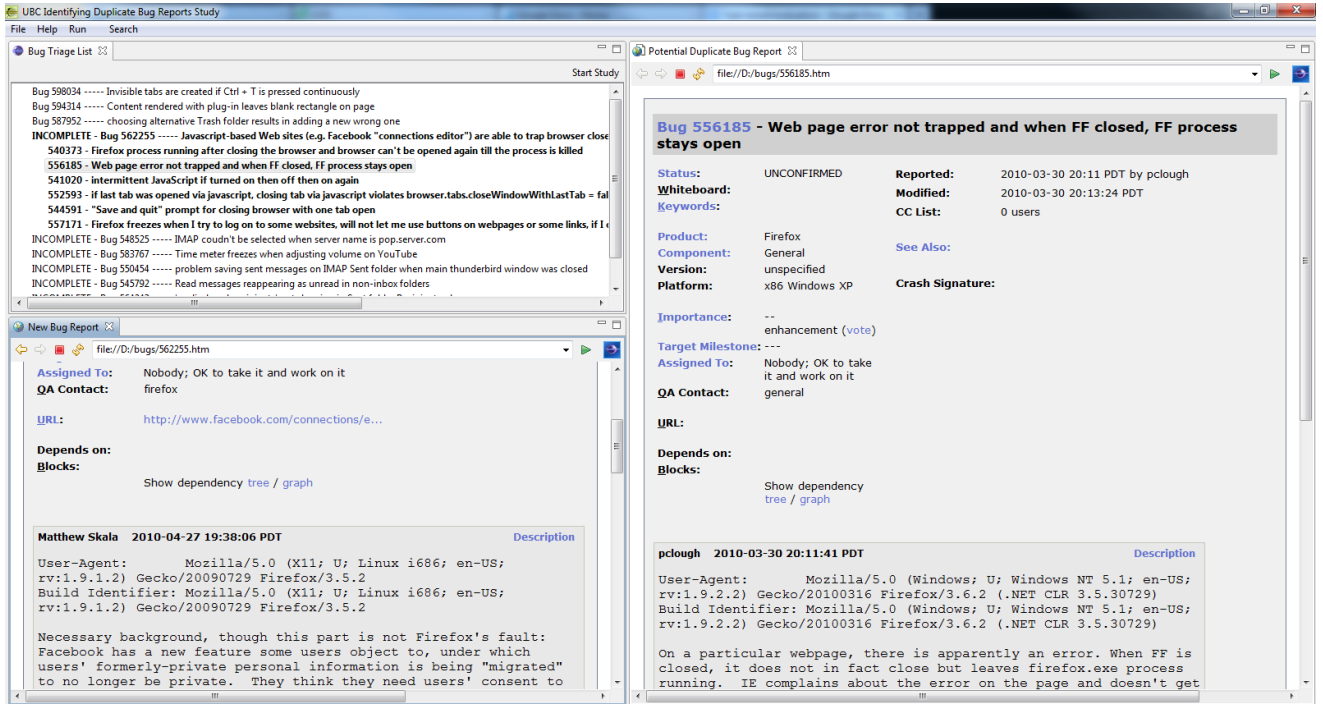


Figure 3.6: The tool used by participants in the user study to perform duplicate detection tasks. The top left 'Bug Triage List' window shows the list of tasks, each consisting of a new bug report and six potential duplicate bug reports. The new bug report and the selected potential duplicate can be viewed in the bottom left window and the right window respectively.

duplicate bug reports. Subjects were asked to compare working with and without summaries in terms of time, difficulty, and having access to sufficient information.

Computing Task Completion Accuracy

A reference solution for each bug duplicate detection task is available in the project's bug repository. If the actual duplicate of the new bug report is among the six potential duplicates, the solution would be the actual duplicate marked as a *duplicate* and the other five marked as *not duplicate*s. To score a subject's solution for such a task, we compare the marking of each potential duplicate to the marking of the same bug report in the reference solution. If the markings are the same, we give a score of 1. If the potential duplicate is marked as a *maybe duplicate* (indicating insufficient information to make a decision) we give a score of 0.5. Otherwise we give a score of 0. To aggregate these scores to a single score for the task, we give a weight of five to the score of the actual duplicate. We chose to use this weighting scheme because we wanted the score of the actual duplicate to equally contribute to the final score as the scores of the other five potential duplicates. In this case the maximum score (that of the reference solution) would be 10 ($(1 \times 5) + 1 + 1 + 1 + 1 + 1$). The score of a solution in which the actual duplicate and one other bug report on the list are marked as a *maybe duplicate* and everything else is marked as a *not duplicate* is $(0.5 \times 5) + 0.5 + 1 + 1 + 1 + 1 = 7$. Finally the accuracy of each task is computed by dividing its score by the maximum score or the score of the reference solution which is 10. The accuracy of a task with a score of 7 (like the example above) would then be 0.7.

If the actual duplicate is not on the list, the reference solution would be all potential duplicates marked as *not duplicate*s. In this case, because there is no actual duplicate on the list, scores of potential duplicates have all the same weight in computing the total score of the task with 6 being the score of the reference solution.

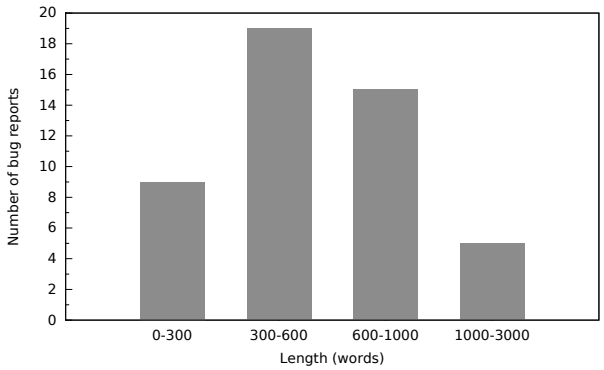


Figure 3.7: The distribution of potential duplicates based on their length.

Tasks

We used the recommender built by Sun and colleagues [96] to generate the list of potential duplicate bug reports. This recommender outperforms previous duplicate detection techniques (e.g., [45, 91, 97]) by comparing bug reports based on an extended version of BM25F, an effective textual similarity measure in information retrieval. The top six bug reports retrieved by this recommender for each new bug report form the list of potential duplicates for each task. The list is sorted with the most similar bug report at the top.

The new bug reports for the duplicate detection tasks (listed in Table 3.8) were randomly selected from the pool of all Thunderbird and Firefox bug reports filed in 2010 and labeled as DUPLICATE with the constraint that at least 4 out of 6 corresponding potential duplicate bug reports be longer than 300 words. We made this choice to ensure that the 100-word summaries are visibly shorter than most of the potential duplicate bug reports. The new bug reports have been drawn from different components of the projects. The diagram in Figure 3.7 shows the distribution of length (in words) of potential duplicates (total number of potential duplicates: $8 \times 6 = 48$). The average length of a potential duplicate in our study is 650 words leading to an average compression rate of 15% for a summary length of 100 words.

Table 3.9 shows the list of potential duplicates for each task where '+' indicates an actual duplicate. Tasks 2 and 5 do not have an actual dupli-

Table 3.8: New bug report used for each task in the user study.

Task	Bug Report	Title	Product	Component
1	545792	Read messages reappearing as unread in non-inbox folders	Thunderbird	Folder and Message Lists
2	546095	Behavior with IMAP attachments utterly broken	Thunderbird	Message Compose Window
3	548525	IMAP coudn't be selected when server name is pop.server.com	Thunderbird	Account Manager
4	550454	problem saving sent messages on IMAP Sent folder when main thunderbird window was closed	Thunderbird	Message Compose Window
5	562255	Javascript-based Web sites (e.g. Facebook "connections editor") are able to trap browser close	Firefox	Security
6	564243	'undisclosed-recipients' not showing in Sent folder Recipient column	Thunderbird	Folder and Message Lists
7	583767	Time meter freezes when adjusting volume on YouTube	Firefox	General
8	587952	choosing alternative Trash folder results in adding a new wrong one	Thunderbird	Account Manager

Table 3.9: List of potential duplicates per task, retrieved by extended BM25F with '+' indicating an actual duplicate. Numbers in parentheses show the length of each bug report.

Task	1	2	3	4	5	6	7	8
Potential Duplicates	539035+ (612)	538803 (309)	538121 (452)	538340+ (209)	540373 (790)	540841 (381)	580795 (581)	558659 (1539)
	538756 (275)	544748 (432)	547812+ (465)	543508 (603)	556185 (204)	562782 (801)	571000 (537)	547682 (335)
	540846 (264)	543399 (723)	547530 (663)	543746 (640)	541020 (692))	549931+ (942)	578804+ (319)	542760+ (1071)
	544655 (1908)	545650 (523)	538115 (415)	549274 (871)	552593 (263)	542261 (167)	542639 (40)	547455 (2615)
	540289 (933)	539494 (827)	541256 (307)	544837 (543)	544591 (338)	550573 (2588)	571422 (656)	564173 (68)
	540914 (569)	541014 (408)	538125 (576)	540158 (997)	557171 (516)	541650 (246)	577645 (456)	539233 (849)

cate among the list of potential duplicates. The length of each potential duplicate, in the number of words, is shown next to it. The data in the table exhibits a recall rate of 75% (6/8: 6 out of 8 tasks have the actual duplicate on the recommendation list) which exceeds the recall rate of all existing duplicate detection techniques that retrieve potential duplicates for a bug report based on its natural language content.

To generate a realistic task setting, for each bug report in Table 3.8, we used the date the bug report was created (filed) as a reference and reverted the bug report and all the six duplicate bug reports to their older versions on that date. All the comments that had been made after that date were removed. The attribute values (e.g. Status, Product, Component) were all reverted to the values on that date.

Participants

All 12 people recruited to participate in the study had at least 5 years (average: 9.9± 4) of experience in programming. This amount of programming experience helped them easily read and understand bug reports which often contain programming terms and references to code.

Participants had different amounts of programming experience in an industrial context. 5 participants had 0-2 years, while the other 7 had an average of 7.1 years of experience in programming in industry. The second group had an average of 5.8 years of experience working with issue tracking systems.

We decided to choose participants with different backgrounds because although people with more industrial experience may have better performance working on duplicate detection tasks, it is often novice people who are assigned as triagers for open bug repositories like Mozilla.

Results

Bug report summaries are intended to help a subject save time performing a bug report duplicate detection task by not having to interact with bug reports in their original format. At the same time it is expected that summaries contain enough information so that the accuracy of duplicate detection is not compromised. We investigated the following three questions

using the data collected during the user study:

1. Do summaries, compared to original bug reports, provide enough information to help users accurately identify an actual duplicate?
2. Do summaries help developers save time working on duplicate detection tasks?
3. Do developers prefer working with summaries over original bug reports in duplicate detection tasks?

Accuracy

Table 3.11 shows the accuracy of performing each of the 8 tasks by each of the 12 participants. The accuracy scores for tasks performed under the *summaries* condition have been marked with a '*'. The top diagram in Figure 3.8 plots the accuracy of performing each task under each condition. In this figure, each accuracy value is the average of six individual accuracy scores corresponding to six different users performing the task under the same condition. On average (computed over all 48 corresponding non-starred accuracy scores in Table 3.11), the accuracy of performing a task under the *originals* condition is 0.752 ($\pm$0.24) while the accuracy of performing a task under the *summaries* condition (computed over all 48 corresponding starred accuracy scores in Table 3.11) is 0.766 ($\pm$0.23). Applying the Mann-Whitney test[17] shows that there is no statistically significant difference between the two groups of accuracy scores ($p = 0.38$). Thus, using summarized bug reports does not impair the accuracy of bug duplicate detection tasks.

Time to Completion

In each study session, we measured the time to complete a task as the difference between the time the task is marked as completed and the time the previous task on the task list was marked as completed. Table 3.10 shows the time each participant took to complete each task, with a '*' indicating the

[17]We opted to use Mann-Whitney instead of t-test as the data does not pass normality tests.

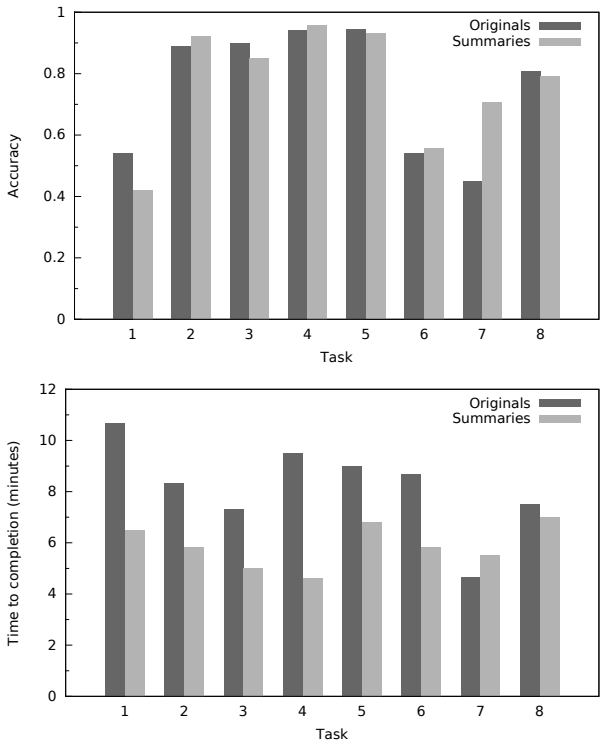


Figure 3.8: The average accuracy and time of performing each duplicate detection task under each condition (original bug reports, bug report summaries).

Table 3.10: Time (in minutes) to complete each task by each participant. '*' indicates *summaries* condition.

	Participant											
Task	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}
1	7*	8*	8*	13	11	4*	15	12	5*	6	7*	7
2	8*	12	7*	12	4*	3	9	4*	6*	3	11	6*
3	5*	8	9	6*	10	4*	4*	4*	7*	4	4	9
4	11	4*	21	9*	4*	2	7	6	10	5*	2*	4*
5	9*	14	7*	13	6	4*	7	11*	9	6*	5	4*
6	6	9*	5*	24	8*	6	6*	3	10	3*	4*	3
7	7	4	7	11*	3	6*	5*	6*	2*	5	3*	2
8	5	6*	10	12*	8*	7	5*	10	7	5*	6	6*

Table 3.11: Accuracy of performing each task by each participant. '*' indicates *summaries* condition.

	Participant											
Task	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}
1	0.45*	0.35*	0.45*	0.45	0.4	0.4*	0.95	0.5	0.4*	0.55	0.45*	0.4
2	0.92*	0.83	1*	1	1*	0.92	0.92	0.75*	0.83*	0.83	0.83	1*
3	0.75*	0.7	0.95	0.95*	1	0.45*	1*	0.95*	1*	0.75	1	1
4	0.95	1*	0.95	0.9*	1*	1	0.95	0.85	0.95	0.85*	1*	1*
5	1*	0.92	1*	1	1	1*	1	0.75*	0.75	0.83*	1	1*
6	0.45	0.5*	0.75*	0.4	0.45*	0.75	0.5*	0.45	0.7	0.45*	0.7*	0.5
7	0.5	0.5	0.5	0.5*	0.5	0.85*	0.75*	0.9*	0.75*	0.2	0.5*	0.5
8	0.95	1*	0.4	0.7*	0.75*	1	0.95*	0.75	1	0.35*	0.75	1*

task was performed under the *summaries* condition. The bottom diagram in Figure 3.8 plots the amount of time for each task under the two conditions averaged over the users.

By comparing time to completion across the two conditions we were interested to know if summaries, by being shorter in length, help users to perform duplicate detection tasks faster. Based on our data, on average it took 8.21 ($\pm$4.52) minutes to complete a task under the *originals* condition while it took 5.90 ($\pm$2.27) minutes to perform a task under the *summaries* condition. The difference is statistically significant based on the Mann-Whitney test ($p = 0.0071$).

The comparison of accuracy scores and time to completion across the two conditions show that using summaries in duplicate detection tasks can help developers save time without compromising the quality of performing the tasks.

Participant Satisfaction

9 out of 12 (75%) participants (all except p_3, p_6, p_9) preferred working with summaries mentioning that it 'was less intimidating/daunting', 'seemed more clear than full bug reports', 'less noisy because they didn't have the header info', 'made them relieved', 'easier to read', 'seemed to preserve what was needed' and 'had less cognitive overload'. The remaining 3 thought they needed to have originals to be sure of the decision they made, mentioning the ideal setting would be to have both summaries and originals available ('wanted to know what was left out from the summaries', 'had problem with loss of context', 'wanted to know who was talking').

10 out of 12 (83%) participants thought they were faster while working with summaries, the other two (p_6 and p_{12}) mentioned that they felt time depended more on task difficulty than the task condition (*summaries* vs. *originals*).

7 out of 12 (58%) participants (p_1, p_3, p_6, p_8, p_{10}, p_{11}, p_{12}) mentioned that the 'steps to reproduce', was probably the most important piece of information in comparing two bug reports and deciding if they were duplicates. One way to further improve the quality of produced summaries is by including in the summary an indication of the the availability of this

information in the original bug report.

3.5.3 Threats to Validity

One of the primary threats to the internal validity of the evaluations we have conducted is the annotation of the bug report corpus by non-experts in the projects. Optimally, we would have had summaries created for the reports by experts in the projects. Summaries created by experts might capture the meaning of the bug reports better than was possible by non-experts. On the other hand, summaries created by experts might rely on knowledge that was not in the bug reports, potentially creating a standard that would be difficult for a classifier to match. By assigning three annotators to each bug report and by using agreement between two to form the gold standard summaries, we have attempted to mitigate the risk of non-expert annotators.

The use of non-project experts in both human evaluations (human judges and task-based evaluation) is a threat to the external validity of our results. This threat has been mitigated by using systems (e.g., Firefox and Thunderbird) that participants were familiar with at least from a user perspective. In case of human judges, one threat is the possibility of them wanting to please the experimenters. In future studies, we will consider interspersing classifier-generated and human-generated summaries to reduce this risk. For the task-based user study, one main threat is the use of general purpose summaries for a particular task. Optimally different summaries should exist for different types of tasks involving bug reports. For instance, a bug report duplicate detection task more likely needs more information on the symptoms of the problem rather than how it should be fixed.

The bug reports in the corpus and the bug reports used in the task-based user study have been chosen to be representative of the intended target of the approach (i.e., lengthy bug reports). There is an external threat that the approach does not apply to bug repositories with mostly shorter reports.

The other threat is the use of different summary lengths for the evaluation of summaries. Summaries as long as 25% of original bug reports were used for the analytic evaluation and the evaluation by human judges because we wanted to match the length of gold standard summaries. We used 100-word summaries for the task-based evaluation to make them visi-

bly shorter compared to the original bug reports and easier to interact with for a participant that has to go over a lot of bug report summaries in the course of a study session.

3.6 Summary

In this chapter, we have investigated the automatic generation of one kind of software artifacts, bug reports, as a representative of software artifacts with mostly natural language content. Using an extraction-based supervised summarization system for conversational data, we found that existing classifiers, trained on email and meeting data, can produce reasonably accurate summaries for bug reports. We also found that a classifier trained on bug reports produces the best results. The human judges we asked to evaluate report summaries produced by the bug report classifier agree that the generated extractive summaries contain important points from the original report and are coherent. We showed that generated bug report summaries could help developers perform duplicate detection tasks in less time without degrading the accuracy, confirming that bug report summaries help software developers in performing software tasks.

Chapter 4

Summarization of Crosscutting Code Concerns

In this chapter we discuss the summarization of structured software artifacts. Compared to natural language software artifacts, the precise and non-ambiguous nature of structured software artifacts enables us to build a semantic representation of an artifact's content and then generate an abstractive summary based on this semantic representation. As an example of structured software artifacts, we chose to focus on summarizing source code that crosscuts different modules in the system as it is particularly difficult to handle this kind of code, referred to as a *crosscutting code concern*, while performing a software change task [6]. The summarization approach we developed produces a natural language summary that describes both what the concern is and how the concern is implemented [83]. We start by presenting background information in Section 4.1. We provide an example of a concern summary in use during a software change task in Section 4.2. We describe our approach in Section 4.3 and present the results of an experiment performed to investigate whether summaries help programmers in performing change tasks in Section 4.4.

4.1 Background

In this dissertation, we define a concern as a collection of source code elements often implementing a feature in the system such as logging, synchronization, authentication, etc. A concern is said to be *crosscutting* when its code is scattered across the code base and possibly tangled with the source code related to other concerns [46]. As an example, Figure 4.1 shows two methods, `comment_get_display_ordinal(...)` and `book_export_html(...)`, from

```
function comment_get_display_ordinal($cid, $node_type) {
  // Count how many comments (c1) are before $cid (c2).
  $query = db_select('comment', 'c1');
  $query->innerJoin('comment', 'c2', 'c2.nid = c1.nid');
  [...]
  if (!user_access('administer comments')) {
    $query->condition('c1.status', COMMENT_PUBLISHED);
  }
  [...]
  return $query->execute()->fetchField();
}



function book_export_html($nid) {
 if (user_access('access printer-friendly version')) {
    $export_data = array();
    $node = node_load($nid);
    if (isset($node->book)) {
      $tree = book_menu_subtree_data($node->book);
      $contents = book_export_traverse($tree, 'book_node_export');
      [...]
    }
    else {
      throw new NotFoundHttpException();
    }
 }
 else {
  throw new AccessDeniedHttpException();
 }
}
```

Figure 4.1: Two methods in Drupal; code elements highlighted in bold font are part of the *authorization* crosscutting concern. The concern is scattered across the codebase and tangled with code of other concerns in the system.

Drupal,[18] an open source content management system. The code elements highlighted in bold font, `user_access(...)` and `AccessDeniedHttpException(...)`, are part of the *authorization* crosscutting concern. The concern is scattered across several modules in the system. As an example, `user_access(...)` is called from 156 different locations in the Drupal codebase. The concern is also tangled with the code of other concerns, for example the (non-crosscutting) concerns implementing *Count the number of preceding comments* and *Generate HTML for export* in Figure 4.1.

[18] `drupal.org`, verified 12/12/12

```
system-authorized-run(), system-authorized-init(),
user-access(), node-access(), authorize-access-allowed(),
authorize-access-denied-page(), filter-permission()
```

Figure 4.2: A sample output of a concern identification technique for the *authorization* concern in Drupal.

It has been observed that a programmer performing a change task has particular difficulty handling a crosscutting code concern as it typically requires browsing through several modules and reading large subsets of code contributing to the concern to determine if the concern is relevant to the task at hand [6]. In cases where the concern is not pertinent to the task, the developer may lose context and become needlessly distracted [22]. Compared to localized code, crosscutting concerns are harder to understand because developers must reason about code distributed across the software system. Crosscutting concerns can also be harder to implement and change because several locations in the code must be updated as part of the same task [26].

The crosscutting nature of a software concern makes it difficult for developers to cost-effectively document the concern in source code and to keep the documentation consistent. As a result, we investigate the automatic generation of natural language concern summaries to enable generation of on-demand concern documentation. Our summarization approach takes as input a list of methods implementing a crosscutting concern. The list of methods contributing to a concern can be generated by using a concern identification technique. Various concern identification approaches, have been proposed in the literature. For example, Marin and colleagues use structural features in the code to mine methods that are called from many different places, which can be seen as a symptom of crosscutting functionality [61]. Breu and colleagues analyze program execution traces for recurring execution patterns [14]. Shepherd and colleagues use natural language processing to mine methods that implement the same feature [93]. Adams and colleagues use the history of code changes to mine code elements that were changed together [2]. Figure 4.2 shows a sample output of a concern identification as a list of code elements (methods) belonging to the *authorization* concern discussed in Figure 4.1.

4.2 Using Concern Summaries

To illustrate how a concern summary, automatically generated by our proposed summarization approach (Section 4.3), can impact a software change task, we consider a programmer who has been asked to add *undo* support for the *change attributes* functionality in a drawing program built on the JHotDraw framework.[19] The *change attributes* functionality, implemented by a class named `ChangeAttributesCommand`, enables a user to change attributes, such as the fill colour or the font, of a figure.

Working in a version of the Eclipse programming environment[20] that includes a plug-in we built to support concern summaries, the programmer begins the task by searching for methods named "undo" in the code. From the search results, the programmer decides to investigate the first result, the `undo()` method in the `PolygonScaleHandle` class. The programmer notices that this method is highlighted in the environment (see the main editor window in Figure 4.3), indicating that the method is part of a crosscutting concern and a summary of the concern is available. The programmer clicks on the highlighted method, causing the associated UNDO concern summary to become visible in the rightmost view in Figure 4.3 (We have reproduced the summary in Figure 4.4 to ease readability; the full generated summary has been provided in Appendix B). Reading the summary, the programmer notes that all of the methods implementing the UNDO concern override the `UndoableAdapter.undo()` method (line 8 in Figure 4.4). Because all methods of the concern override this one method, the programmer hypothesizes that this method provides a mechanism for supporting *undo* functionality in JHotDraw. The programmer realizes that one of the methods implementing the concern may provide an example of how to use this mechanism; the programmer expands the list of methods contributing to the concern implementation (line 1 in Figure 4.4) and chooses one of them, the `undo()` method in the `InsertImageCommand` class, to investigate.

When reading the summary, the programmer had also noted that each concern method is declared by a class named *UndoActivity* (line 10 in Figure 4.4). As the programmer investigates the `InsertImageCommand`, the pro-

[19] JHotDraw601b, `jhotdraw.org`, verified 08/03/11
[20] `eclipse.org`, verified 20/07/11

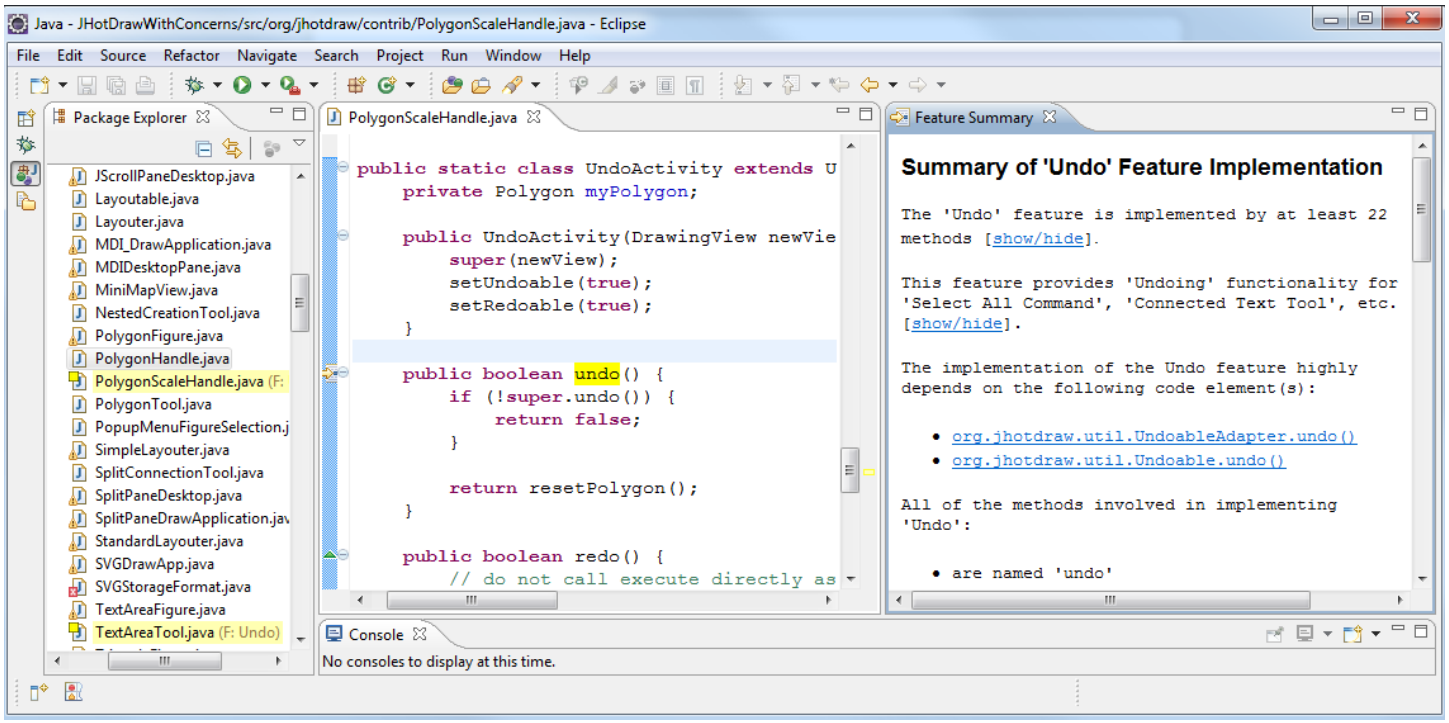


Figure 4.3: The concern summary Eclipse plugin in action.

grammer looks for an `UndoActivity` class, finding it as a static inner class of `InsertImageCommand` which implements the `UndoableAdapter` interface. The programmer has now learned, with the help of information in the summary, that implementing *undo* feature for the *change attributes* functionality will involve adding a new static inner class named `UndoActivity` implementing the `UndoableAdapter` interface to the class supporting the *change attributes* command. Three different parts of the summary have helped the programmer quickly determine how to proceed with the change task of interest.

In this scenario, the provided summary explicitly describes the functionality a programmer wants to extend. We believe summaries can also help a programmer understand code that is less directly related to a change task at hand and can help a programmer determine when code is not related to a change task. The laboratory study we describe later in this chapter (Section 4.4) includes a task where the concerns are less directly related to the task-at-hand.

4.3 Approach

Given a set of methods belonging to a concern of interest and the source code for the system with the concern, our approach generates a natural language summary of the software concern code. A generated summary includes information about structural patterns amongst the methods implementing the concern and natural language facts embedded in identifiers in the code.

Our approach consists of three steps. First, we extract structural and natural language facts from the code; these extracted facts are represented in an ontology instance. Next, we apply a set of heuristics to the ontology instance to find patterns and salient code elements (e.g., classes, methods or fields) related to the concern code. In the last step, we generate the sentences comprising the summary from the patterns, salient code elements and information from the ontology instance. We developed the approach iteratively using three example concerns from the JHotDraw, Drupal and Jex systems identified at the top of Table 4.1 and tested it on the other five

Summary of 'Undo' Feature Implementation

1: The 'Undo' feature is implemented by at least 22 methods [show/hide].

2: This feature provides 'undoing' functionality for 'Select All Command', 'Connected Text Tool', etc. [show/hide].

3: The implementation of the 'Undo' feature highly depends on the following code element(s):

4: • org.jhotdraw.util.UndoableAdapter.undo().

5: • org.jhotdraw.util.Undoable.undo().

6: All of the methods involved in implementing 'Undo':

7: • are named 'undo'.

8: • override method org.jhotdraw.util.UndoableAdapter.undo().

9: • override method org.jhotdraw.util.Undoable.undo().

10: • are a member of a class named 'UndoActivity'.

[*3 other patterns involving all methods*]

[*6 patterns involving all but one of methods*]

11: Around half of the methods involved in implementing 'Undo' call one or more of the following methods:

12: • org.jhotdraw.framework.DrawingView.clearSelection().

13: • org.jhotdraw.util.UndoableAdapter.getAffectedFigures().

[*2 other patterns involving about half of methods*]

Figure 4.4: A part of summary of 'Undo' concern in JHotDraw.

Table 4.1: Concerns used for developing and testing of the summarization approach

Concern	System	# Methods	Origin	Dev./Test?
Undo	JHotDraw	23	similar to [61]	Dev.
Save	Drupal	6	manual creation	Dev.
Anonymous	Jex	8	similar to [89]	Dev.
MoveFigure	JHotDraw	8	`basicMoveBy(...)` methods in `figures` package	Test
DrawPlot	JFreeChart	25	`draw(...)` methods in `plot` package	Test
Autosave	jEdit	9	similar to [90]	Test
Property	jEdit	10	property handling in `jEdit` class	Test
Marker	jEdit	40	similar to [87]	Test

concerns.[21]

4.3.1 Step 1: Extracting Information

We extract both structural and natural language information from the code. We use an ontology to represent extracted information so that it can be handled in a unified and formalized way.

Extracting Structural Information Information about how methods that belong to a concern interact with each other and with the rest of the code for the system can help a programmer deduce how the functionality of a concern works. To provide this structural information in a summary, we analyze the system's source code to extract structural facts about all of the code elements in the system.

Our current prototype focuses on generating summaries for Java code.[22] We use the JayFX[23] system to extract structural relationships between various Java source code elements. Table 4.2 lists the structural facts we extract. We ignore facts involving Java library classes and methods.

We use an ontology instance based on the SEON Java Ontology scheme[24] to store the extracted information [107] . This ontology scheme provides all the concepts needed to model the structural relationships in Table 4.2 and has been shown to be helpful for natural language querying of Java code [108]. Every extracted structural fact is represented in the ontology instance by one or more (*subject*, *predicate*, *object*) triples. For example, the fact that class c_1 extends class c_2 (Extends(c_1, c_2) in Table 4.2), is represented by triples (c_1, `hasSuperClass`, c_2) and (c_2, `hasSubClass`, c_1).

A convenient way to manipulate an ontology instance, which we use in step two of our approach, is through an RDF graph.[25] In an RDF

[21] The details for the systems follow: Drupal (`drupal.org`), Jex (`cs.mcgill.ca/~swevo/jex`), jEdit (`jedit.org`) and JFreeChart (`jfree.org/jfreechart`), all verified 10/03/11.

[22] The overall summarization approach can easily be generalized to programming languages other than Java.

[23] `cs.mcgill.ca/~swevo/jayfx`, verified 10/03/11

[24] `evolizer.org/wiki/bin/view/Evolizer/Features/JavaOntology`, verified 10/03/11.

[25] `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-data-model`, verified 12/12/12

Table 4.2: Structural relationships between code elements

Calls(m_1, m_2)	method m_1 calls method m_2
Reads(m, f)	method m reads field f
Writes(m, f)	method m writes field f
Declares(c, m)	class c declares method m
Declares(c, f)	class c declares field f
Implements(c, i)	class c implements interface i
Extends(c_1, c_2)	class c_1 extends class c_2
HasInnerClass(c_1, c_2)	class c_2 is an inner-class of class c_1
Overrides(m_1, m_2)	method m_1 overrides method m_2
HasParameter(m, p)	method m has parameter p
Returns(m, t)	method m has return type t

graph, each (*subject, predicate, object*) triple is represented by an edge from *subject* to *object* labeled with *predicate*. Figure 4.5 shows part of the RDF graph populated with facts extracted from JHotDraw. In this graph, for example, `Class1`, whose name is *UndoActivity* implements `Interface1`, whose name is *UndoableAdapter*. This fact is shown with an edge labeled `implementsInterface` from `Class1` to `Interface1`. We use Jena[26], a Java framework for building Semantic Web applications, to create and process ontology instances.

Extracting Natural Language Information In addition to information about how a concern is implemented, a programmer can benefit from information in a concern summary that describes what the concern is about. To provide such information, we extract natural language information from identifiers in the source code. For example, when many methods in a concern share a substring in their name, such as "save" or "undo", the shared substring may be a useful clue to the functionality of that concern. To determine this information, we use the approach of Fry and colleagues to extract the Verb-DO (Direct Object) pairs for each method listed in the concern [37]. A Verb-DO pair is intended to capture the action of a method and the context, or object, to which the action is applied. For example, extraction of a Verb-DO pair (*Save, Comment*) for a method implies the method

[26] `jena.sourceforge.net`, verified 10/03/11

may be performing a *Save* action on a *Comment* object. As we will describe in the next subsection, having Verb-DO pairs can help in finding patterns amongst methods of a concern that implement the same functionality.

We extend the SEON Java Ontology to include the natural language facts we extract by modeling these facts as additional properties of existing classes in the ontology, namely `hasVerb` and `hasDO` which are added to the `method` ontology class. Figure 4.5 shows how the natural language facts are represented in the ontology. For instance, `Method3`, whose name is *undo* has *Rotate Image Command* as its Direct Object (DO).

4.3.2 Step 2: Generating Abstract Content

A summary of a concern should include the most pertinent information about the concern, making the concern code easier to understand for a programmer who must interact with the code as part of performing a change task. After extracting structural and natural language facts, we process the facts to generate pertinent content to be included in the concern summary. The processing is performed based on two sets of heuristics: finding similarities between methods of the concern and finding source code elements in the system's code that are important to the implementation of the concern.

The first set of heuristics, which we refer to as path patterns, are based on the observation that similarity among concern methods can provide valuable insight into the *what* and *how* of the concern. For example, the fact that all of the methods in the concern have the same verb "save" in their name might indicate that the concern is about implementing the saving of some information in the system. The fact that all methods in a concern override the same method implies the method being overridden may be part of a mechanism for implementing the functionality of interest.

The second set of heuristics is based on the observation that the implementation of a concern is often highly dependent on a few code elements, which we refer to as salient code elements. Salient code elements may include concern methods but also may be classes, interfaces, fields or other methods of the system. These code elements can provide potential good starting points for a programmer who needs to further investigate the concern.

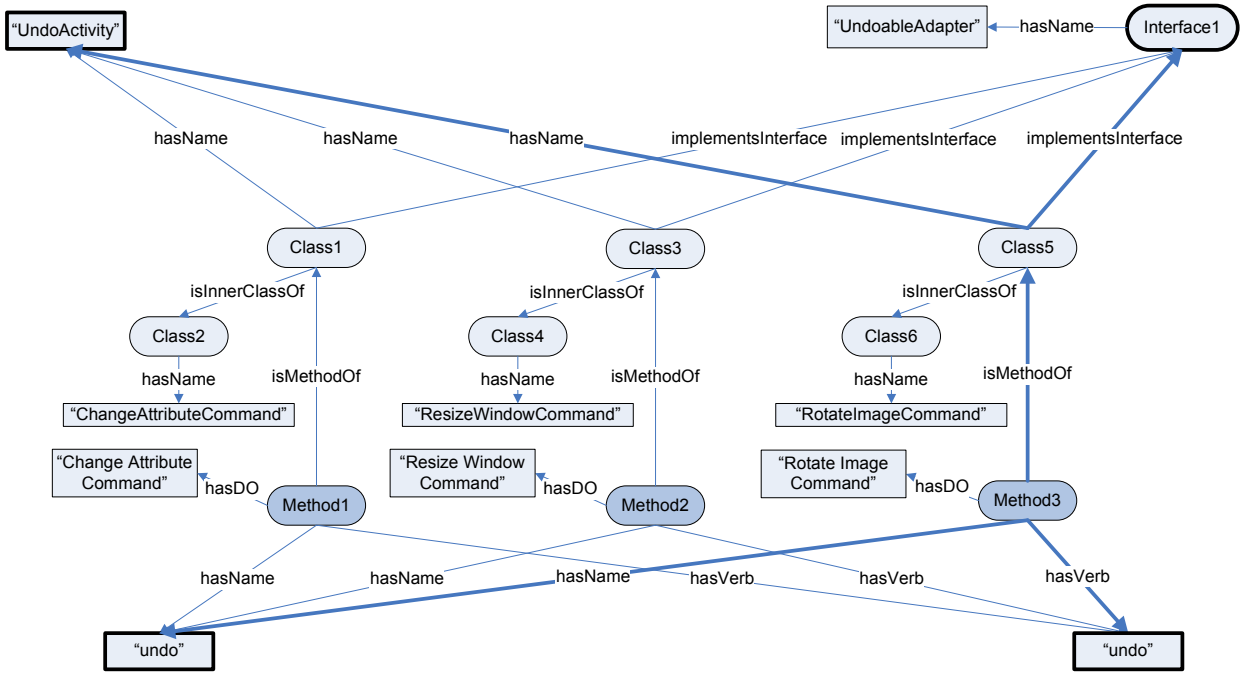


Figure 4.5: Part of the JHotDraw RDF graph (Method1, Method2 and Method3 belong to the Undo concern).

Path Patterns We find similarities shared amongst concern methods by finding similar paths involving the concern methods in the RDF graph constructed from the extracted structural and natural language facts. We refer to a set of similar paths as a path pattern. Our approach looks for path patterns centered around a single code element in the RDF graph; we refer to this single code element as the target of a pattern. To extract these patterns we consider all paths in the RDF graph, with a maximum length of d ($d = 3$ for our current prototype), that start at a concern method and that end at an arbitrary node in the RDF graph. For each of these paths, we check to see how many other concern methods share the same path expression, which is a sequence of node types and edge labels on the path. As an example, the RDF graph in Figure 4.5 includes three concern methods of the UNDO concern in JHotDraw, specifically `Method1`, `Method2` and `Method3` (all named *undo* evident from the outgoing `hasName` edges). In this graph, the path expression that corresponds to the path from `Method1` to `Interface1` is "`isMethodOf--Class:Class--implementsInterface`". The same path expression describes the paths between other concern methods (e.g., `Method2` and `Method3`) and `Interface1`. Hence, the concern methods are all involved in a pattern with "`isMethodOf--Class:Class--implementsInterface`" as the path expression and `Interface1` as the target node. Our algorithm finds three other such patterns in the RDF graph shown in Figure 4.5. The corresponding four target nodes are highlighted with bold borders in the figure.

This naive algorithm can extract a lot of path patterns. Since a concern summary is supposed to include the most pertinent information, we must rank the patterns. Our ranking approach considers three factors:

1. the percentage of concern methods involved in the pattern; a pattern in which most members of a concern are involved shows a higher degree of similarity between concern elements and can help to describe the concern in a general and abstract way,

2. the length of the path expression; we believe patterns with shorter path expressions are easier for a programmer to read and understand, and

3. the kind of structural pattern; we believe patterns about the hierar-

chical structure of the code are easier for a programmer to read and interpret quickly than patterns about calling structure.

We rank patterns first by the percentage of code elements involved in the pattern and then on the length of the path expression. Among patterns with the same percentage and the same path expression length, a calling pattern is ranked lower.

Salient Code Elements We use the term salience to describe the likely importance of a code element from the system in the implementation of a concern. A salient code element may or may not be a method used to define the concern. We identify potentially salient code elements by applying graph theoretic measures of node centrality to the RDF graph representing the ontology instance of the code. Centrality describes the relative importance of a node in a graph.

A number of different algorithms exist to identify central nodes in a graph (e.g., degree, closeness, or betweenness centrality) [35]. We experimented with different algorithms on the sample concerns used to help define our approach and found that the betweenness centrality measure [51] generates results that are closer to our intuition of a salient code element, namely that the code element is involved in many shortest paths between two concern methods. For example, the salient node in the UNDO concern is `UndoableAdapter.undo()` as it is the common connecting point between different concern elements.

4.3.3 Step 3: Producing Sentences for the Summary

The concern summaries we generate consist of four main parts (see Figure 4.4 for an example):

1. **Listing**: The first part is a sentence stating how many, and which methods, comprise the concern (Figure 4.4, line 1). This sentence is generated by using the following template:

   The [concern title] is implemented by at least [concern size] methods [expandable list of the concern elements]

2. **Description**: The second part of a concern summary is a short description of the concern (Figure 4.4, line 2). This sentence is only generated when the names of the concern methods share the same verb (e.g. *Undo*). The following is the template for generating the sentence:

   This feature provides [Verb]ing functionality for [DO_1], [DO_2], [expandable list of other DOs]

3. **Salient code elements**: The third part is generated using the following template and the two most salient code elements identified in the content generation phase (Figure 4.4, lines 3-5):

   The implementation of feature [concern title] is highly dependent on the following code elements:

   [SalientCodeElement1]

   [SalientCodeElement2]

4. **Patterns**: The generation of the fourth part involves textual translation of path patterns, starting with those with highest rank (Figure 4.4, lines 6-10 and 11-13). A path pattern is translated to a textual form by using both its target node and its pattern expression. For example, the path pattern in Figure 4.5 with the path expression of "`isMethodOf--Class:Class--hasName`" and a target node of "`UndoActivity`" is translated to

   is a method of a class named "UndoActivity"

   Before including this sentence in the summary, we need to specify the number of concern methods associated with it. In this example, all concern methods are involved in the pattern, resulting in the use of the following template when generating the final sentence:

   All of the methods involved in implementing [concern title]

Depending on the number of code elements involved in a path pattern "*All*" in the above sentence can be replaced by "*All but one*", or "*Around half*" (when the percentage is in the 45 to 55% interval), "*65%*", etc.

4.4 Task-based Evaluation

Concern summaries are intended to help programmers better perform software change tasks. To understand if concern summaries meet this goal, we performed a laboratory study to investigate the following three questions:

Q1 How does a programmer use concern summaries when they are available?

Q2 Can concern summaries help a programmer identify code pertinent to a change task?

Q3 Can concern summaries help a programmer perform a change task more easily?

4.4.1 Participants

Our study involved eight participants ($p_1 \ldots p_8$). Each participant had at least three years of programming experience; one year of which included Java programming in the Eclipse environment. Two of the participants were professional programmers; the other six were students (five graduate, one senior undergraduate) in the UBC Computer Science Department.

4.4.2 Method

Each participant was asked to perform two software change tasks: adding autosave capabilities to the jEdit[27] text editor, a task defined and used in an earlier study [88], and adding undo functionality for a command in the JHotDraw 2D graphical framework,[28] similar to the task outlined in Section 4.2. The jEdit task involves changes to different locations in different

[27] jEdit4.1-pre6, `www.jedit.org`, verified 08/03/11
[28] JHotDraw60b1, `www.jhotdraw.org`, verified 08/03/11

classes in the code; the JHotDraw task requires changes to only one class. Detailed descriptions of these tasks have been provided in Appendix B. Four of the participants (p_1, p_3, p_5 and p_7) worked on the jEdit task as their first task; the other four (p_2, p_4, p_6 and p_8) worked on the JHotDraw task first. A participant had access to concern summaries only for the second task on which he or she worked. We decided not to provide concern summaries for the first task as that might have caused the participant to try to extract concern summaries on her own during the second (no summaries provided) task. For the jEdit task, a participant had access to two concern summaries: one for an AUTOSAVE concern and the other for a PROPERTY concern (summaries in Appendix B); the first concern relates directly to the task whereas the second concern describes functionality that interacts with the autosave functionality. For the JHotDraw task, a participant had access to one concern summary, the UNDO concern summary shown partially in Figure 4.4 (full summary in Appendix B); this concern directly related to the given task.

Each participant was given a maximum of 30 minutes to complete each task and was instructed to create a plan for performing the task, identifying the code elements that either needed to be created or changed to complete the task. We chose this time limit as it was sufficient for the participants in the pilot study to create a high level plan of how each task should be performed. Each participant was instructed not to make any changes to the code and not to use the debugger, so as to keep the participant's attention focused on program navigation activities. Before performing the second task, a participant was asked to read a one-page tutorial on how to use the Eclipse plug-in to view a concern summary; this tutorial focused on the tool and not the intent or composition of a summary. Similar to [23], after each task, a participant was asked to complete the NASA TLX questionnaire [42] to assess the participant's views about the difficulty of the task and his or her performance on the task. Each session concluded with a guided interview. Each participant was compensated with a $25 gift card for their time.

We asked each participant to think-aloud as she or he performed the change task and collected screen data synched to audio as the participant worked. In addition, the experimenter took notes as a participant worked

Table 4.3: Number of interactions with summaries

Task	JHotDraw				jEdit			
Participant	p_1	p_3	p_5	p_7	p_2	p_4	p_6	p_8
# Interact.	9	8	1	7	(4,2)	(4,6)	(3,8)	(2,2)

and during the follow-up interviews and we collected the plans created by each participant for each task and the results of the two TLX surveys completed by each participant.

4.4.3 Results

We consider each of the three research questions.

Q1 – Use of Concern Summaries Table 4.3 reports the number of interactions each participant had with a concern summary as he or she performed the second task. This data is drawn from an analysis of the video and audio captured as participants worked. An interaction was considered to have occurred when a participant read the summary or used it to navigate the code. For the AUTOSAVE task, the first number in the parenthesis reports the number of interactions with the PROPERTY summary and the second number with the AUTOSAVE summary. This data shows that all participants interacted with a concern summary at least once during a task; some participants interacted with a summary more extensively than others.

We observed during the sessions that all participants used the summaries to navigate to parts of the code mentioned in the summary. We also observed that participants consulted summaries after long navigation paths through the code, seemingly using the summary as a focal point that could help the participants reorient themselves to the task.

For the jEdit task, all four participants considered the listings of the concern elements in both summaries to be the most useful part of the summary. The next most useful part they reported was the sentences about salient code elements. None of these four participants found the patterns section of the summary useful.

For JHotDraw task, three of the four (75%) participants (p_1, p_3 and p_7) reported all parts of the summary except the calling patterns equally

useful. These participants reported that the salient nodes described in the summary provided good clues to start investigating the code and the listing of concern elements helped to find an example of how the undo functionality had already been implemented in the code. These participants reported that the patterns which describe facts about the hierarchical structure in the code also helped to figure out how the change should be implemented. Only one participant (p_1) reported the patterns describing which code calls each other as useful. One possible reason is that patterns involving code calls were seen as less useful is that these patterns are only likely helpful in the later stages of performing the task and the time provided to most participants was insufficient to reach these stages.

Q2 – Finding Pertinent Code Table 4.4 reports the ratio of relevant code elements visited by a participant as part of a task to the total number of code elements visited by that participant. We refer to this ratio as the navigation efficiency; a high value suggests a participant spent more effort on task than off task. We consider a code element relevant to the task if: 1) the element must be changed to complete the task or 2) the element provides information needed to complete the task. A list of all the code elements that were counted as relevant for each task in our user study is available online.[29] The data in Table 4.4 shows that navigational efficiency rose when summaries were available. On average, when summaries were available, navigational efficiency was 87.3% (SD: 8.7) whereas it was 67.6% (SD: 22.44) when no summaries were available. It is interesting to note the much smaller standard deviation, even across tasks, when summaries are available. To investigate the effect of summaries on individual performance, we computed a paired t-test on the navigational efficiency within subjects; this t-test shows that the improvement in the navigational efficiency with summaries is statistically significant at the 95% confidence level ($p \approx 0.045$). This data suggests that the presence of summaries did help programmers find code relevant to the task.

[29] `cs.ubc.ca/labs/spl/projects/summarization.html`

Table 4.4: Navigation efficiency

Task Order	jEdit - JHotDraw			
Participant	p_1	p_3	p_5	p_7
Task 1	39% (9/23)	100% (11/11)	87% (14/16)	68% (15/22)
Task 2	92% (12/13)	93% (14/15)	95% (19/20)	86% (18/21)

Task Order	JHotDraw - jEdit			
Participant	p_2	p_4	p_6	p_8
Task 1	81% (29/36)	38% (13/34)	74% (20/27)	54% (22/41)
Task 2	74% (14/19))	89% (17/19)	93% (14/15)	76% (16/21)

Q3 – Ease of Task Completion The TLX questionnaire provided to each participant after each task asks the participant to rank on a scale of 1 (low) to 10 (high) the mental demand, temporal demand, frustration level, effort and performance (level of success) perceived by the participant for the task. Table 4.5 focuses on the data for the mental demand and performance perceptions ratings as the former directly relates to the participant's perceived ease of completing a task and the latter relates to the participant's perceived success on the task. This data shows that, regardless of task, six of the eight (75%) participants (p_1, p_3, p_7, p_4, p_6, and p_8) felt that the task performed with summaries was less demanding than without summaries. Only one participant (p_5) reported an increase in mental demand on the task with a summary; the other participant (p_2) felt both tasks were equally (and very) mentally demanding. This data also shows that all but one participant (p_2), or 87%, felt their level of success (i.e., performance) was better on the task with summaries than without.

4.4.4 Threats to Validity

Three major threats to the internal validity of the laboratory study are the likely lack of isomorphism between the two tasks used, the differences in relationships of the concerns to the tasks in the two systems, and likely large differences in abilities of participants. We believe these are acceptable threats for an initial exploratory study.

Three major threats to the external validity of the study results are the

Table 4.5: TLX scores: mental demand and performance

Mental Demand

Task Order	jEdit - JHotDraw				JHotDraw - jEdit			
Participants	p_1	p_3	p_5	p_7	p_2	p_4	p_6	p_8
Task 1	8	8	8	9	10	8	9	9
Task 2	3	6	9	6	10	4	5	8

Performance

Task Order	jEdit - JHotDraw				JHotDraw - jEdit			
Participants	p_1	p_3	p_5	p_7	p_2	p_4	p_6	p_8
Task 1	2	6	3	2	7	5	2	6
Task 2	9	8	5	8	7	8	7	7

use of only two tasks, the limited number of participants and the limited time given to each participant to work on a task. The first threat is mitigated, to some extent, by the use of non-isomorphic tasks as the chance that a task is representative of actual change tasks is higher. By using non-isomorphic tasks on different systems, we also mitigated risks of learning effects from one task to another. The second threat is mitigated by using participants with some level of experience with the programming language and the environment. We chose to accept the risk of the third threat to be able to perform a laboratory study within a reasonable time frame for participants.

4.5 Summary

In this chapter, we have investigated summarization of crosscutting code concerns to explore the problem of summarizing structured software artifacts. We introduced an automated summarization approach for crosscutting concern code that aims to help programmers make judicious choices about what code to investigate as part of a software change task. Our approach is a form of abstractive summarization that produces a description of crosscutting code through a series of natural language sentences. These sen-

tences describe facts extracted from structural and natural language information in the code and patterns detected from the extracted facts. Through a study with eight programmers, we demonstrated that these concern summaries, although less precise than the code itself, can help a programmer find pertinent code related to a change task on a non-trivial system. Through the study, we also determined that the presence of concern summaries made programmers perceive a task as less difficult and perceive their performance on the task as more successful. We view our approach as a proof-of-concept demonstration that natural language summaries of crosscutting code concerns can play a useful role in software development activities.

Chapter 5

Multi-document Summarization of Natural Language Software Artifacts

The information to answer a question a developer has is often located across many artifacts. To find and understand this information, the developer has to put together various pieces of information from different artifacts. This process, involving manually navigating back and forth between related artifacts, can be tedious and time consuming [36]. A multi-document summary of software artifacts containing relevant information can help a developer more easily address her information needs. As an example, a developer trying to understand the reason behind a change in code, often needs to consult a number of project documents related to the change. Generating a summary of these documents is a special case of multi-document summarization of natural language software artifacts. In this chapter, we investigate whether such a summary generated by our proposed multi-document summarization technique can help a developer more easily find the rationale behind a code change.

We begin with providing background information (Section 5.1) and a motivational example (Section 5.2). We then describe our approach (Section 5.3) and provide details on our human annotated corpus (Section 5.4). Finally we discuss an exploratory user study conducted to investigate if generated summaries provide motivational information behind a code change (Section 5.5).

5.1 Background

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. The rationale behind a code change often can be found by consulting a number of project documents. As a starting point, a 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 by manually navigating the set of related documents can be time consuming and cognitively demanding.

In this chapter, we investigate whether a multi-document summary of project documents related to a code change can help a developer understand the reason behind a code change. We propose a supervised extractive summarization approach in which we identify the most relevant sentences in a set of documents provided as including motivational information about the change [82]. While some efforts have considered helping developers explore the information in the project's repositories to make it easier for them to find the reason behind a code change (e.g., [13]), our approach is the first attempt at providing the answer to the 'why' of a code change. Other approaches that have analyzed source code changes, mainly focused on the 'what' and 'how' of a code change. Examples include identification (e.g., [34]), impact analysis (e.g., [76]) and visualization (e.g., [103]) of code changes.

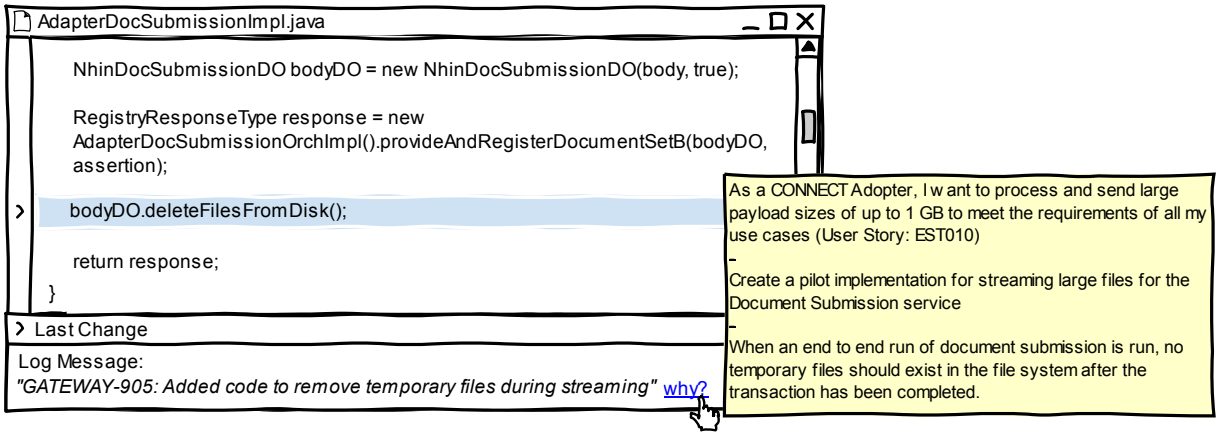


Figure 5.1: A summary describing the motivation behind a code change of interest appearing as a pop-up in a mock-up code development environment.

5.2 Motivating Example

Consider a developer who is working on the open source project CONNECT.[30] 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 as part of the task the developer is working on, she needs to make a change to the `provideAndRegisterDocumentSetb` method of the `AdpaterDocSubmissionImpl` class. 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.[31] `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 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

[30] `http://connectopensource.org`, verified 12/12/12, CONNECT is an open source software and community that promotes IT interoperability in the U.S. healthcare system.
[31] `https://issues.connectopensource.org`, verified 12/12/12

a code change of interest in an IDE and the pop-up shown in Figure 5.1 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 the part of the developer.

5.3 Approach

The problem of generating a summary of a set of documents related to a code change is a special case of the more general problem of multi-document summarization of natural language documents. 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 (e.g., [63, 80]). 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 5.2, `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 important sentences to extract to form a summary. We identified a set of eight sentence-level features to locate the most relevant sentences. We chose to use supervised learning summarization approach in which a classifier scores each sentence of a chain based on the values of the eight sentence features. The highest scored sentences are then extracted to form a summary.

The eight features we investigated are:

1. *fOverlap*. This feature measures the content overlap between a sentence (of a document) 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 or a task leading to a commit message). We hypothesize that sentences containing these

common 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 parent and child 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., [38, 67]). In document D in a chain of documents relevant to a code change, after word stemming and stop-words removal, $overlapScore$ for word w is computed as follows:

$$overlapScore(w, D) = IDF(w) \times \\ [TF(w, Parent(D)) + TF(w, Child(D)]$$

Where TF and IDF stands for Term Frequency and Inverse Document Frequency respectively. For sentence s in document D, $fOverlap$ is computed by summing up the $overlapScore$s of all the words in the sentence:

$$fOverlap = \sum_{w \in s} overlapScore(w, D)$$

2. $fTF\text{-}IDF$. For a sentence s in document D, this score is computed as the sum of the TF-IDF scores of all words in the sentence:

$$fTF\text{-}IDF = \sum_{w \in s} TF\text{-}IDF(w, D)$$

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 [104] and bug reports [56]. $fTitleSim$ is computed as the cosine between the $TF\text{-}IDF$ 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, [79]). We take the approach of multi-document summarization system MEAD [78] to compute $fCentroidSim$. The chain centroid is computed as a vector of words' average $TF\text{-}IDF$ 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.

5.4 Corpus

To investigate if the features discussed in Section 5.3 can be effective in identifying summary-worthy sentences and also to train a classifier, 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 summary shown in Figure 5.1.

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 [38], 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 users. 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*}.

We used the feature analysis technique introduced by Chen and Lin [21] to compute the F statistics measure for all 8 sentence features based on the gold standard summaries in our corpus. Figure 5.2 shows F statistics scores 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

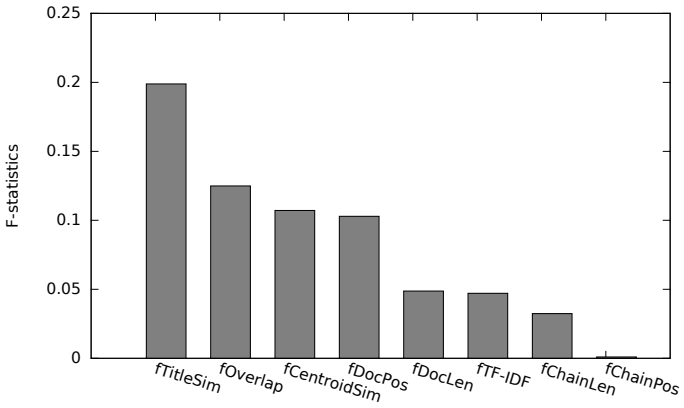


Figure 5.2: F statistics scores for sentence features

two least helpful features. One possible explanation is that each document in the chain contributed almost equally to the final summary. Removing the two least helpful features, we trained a classifier based on Support Vector Machines (SVMs) using the *libSVM*[32] toolkit. This classifier has a k-fold cross validation accuracy of over 82% showing a high level of effectiveness in identifying important sentences.

5.5 Exploratory User Study

To investigate whether developers find chain summaries to be indicative of the motivation behind a code change, we conducted an exploratory user study. We set out to answer the following two research questions:

- *Q1*- Does the chain of related project documents contain information on the reason behind a code change?
- *Q2*- Is the summarization approach (Section 5.3) effective in identifying and extracting information on the rationale behind a code change?

We performed the study with project experts who had worked on the code changes since their knowledge of the project and the code change was

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

Table 5.1: Data used in the exploratory user study

Chain	Bug reports in the chain	Length
#1	279709-361014-369290-370635	1743
#2	355974-376233-378002-385127	2334
#3	158921-349510	3161
#4	333930-370620	1090

needed to accurately investigate the above questions. Because of the participation of experts, the study was exploratory and focused on gaining more insight about how the summarization approach can be improved to more effectively identify the information about the reason of a code change.

The study was performed in the context of the open source project Eclipse Mylyn.[33] We chose this project because the chain of related documents for each code change can be retrieved from the project's bug repository by following available explicit links. The commit message related to the code change includes the identifier of the last bug report in the chain. This bug report is in turn connected to other bug reports in the chain via 'blocks/depends-on' links. Figure 5.3 shows an example of how a code change and its related chain are linked in Mylyn.

We invited a number of active developers of Mylyn to participate in our study. Two of them accepted to take part in a one-hour study session. For each participant in our study, we identified two chains related to two code changes made by the participant. Table 5.1 shows detailed information about the selected chains. *Participant#1* worked on *Chain#1* and *Chain#2*. *Participant#2* worked on *Chain#3* and *Chain#4*. Using the classifier trained on our human-annotated corpus, we produced 150-word summaries for the selected chains.

In a study session, for each chain, the participant was first given the links to the related code change (in the Eclipse's version control repository) and the bug reports in the chain (in the Eclipse's bug repository). To investigate *Q1*, we asked the participant the following questions:

- *D*oes the chain of bug reports contain information describing why the code was changed? If so, highlight those parts of text that in particular

[33] `www.eclipse.org/mylyn`, verified 12/12/12

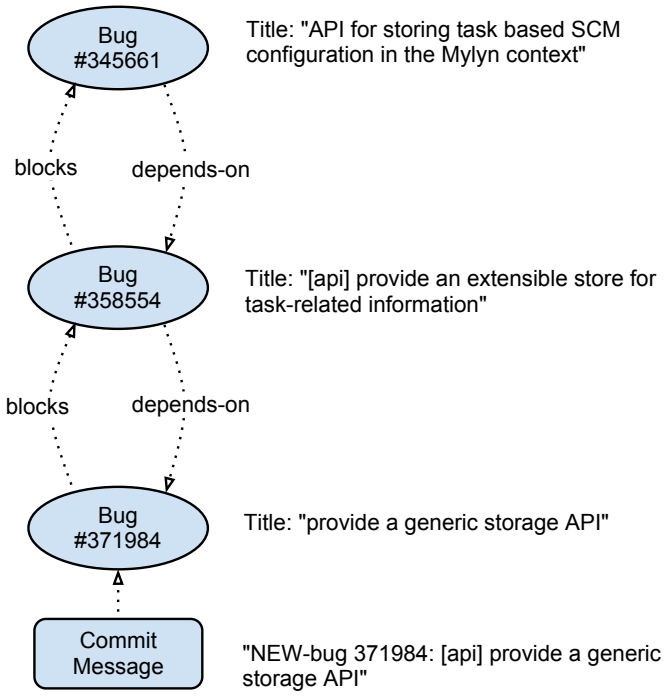


Figure 5.3: An example of the links between a code change (commit) and the related chain in Mylyn.

provide information on the reason behind the code change.

- *I*s the highlighted information complete in describing why the code has changed? Based on your knowledge of the code, were there any other reasons behind the code change that have not been documented in the chain?

Then, the participant was given the automatically generated summary for the chain. To investigate *Q2*, we asked the participant the following question:

- *D*oes the summary contain information describing why the code was changed (in particular in reference to the information you highlighted

in the bug reports)?

The session was concluded by asking the participant the following questions:

- In case of wanting to know the reason behind a code change, would you rather consult the chain or the summary?
- Where else you might look for relevant information?
- Any suggestion on how the summaries can be improved, either in the content or in the way it is presented?

5.5.1 Results

Relevance of Chains

Here are the participants' responses when asked whether the chains provided information about the related code changes (*Q1*):

- *Chain#1*: The participant (*Participant#1*) thought the information in the chain was enough to understand the reason behind the code change ("I would say the information is sufficient to understand why the code changed"). At the same time, some information was missing ("The bugs do not describe the design decisions in detail, e.g. why this was implemented as an SPI and what kind of extensibility is expected").
- *Chain#2*: *Participant#1* did not highlight any part of the first bug report in the chain (355974) as he found it irrelevant to the code change. He thought the rest of the chain contained all the information related to the reason of the code change ("I would say the information is complete, there weren't any other reasons to change the code as far as I know").
- *Chain#3*: *Participant#2* considered the first bug report in the chain (158921) to be irrelevant (nothing was highlighted). He thought the second bug report included the reason behind the code change minus some explanation ("I think it's complete, except that there is 1 bit of code that was added which include a comment explaining why it was added - this explanation is not in the bug reports").

- *Chain#4*: *Participant#2* thought the chain included all information needed to understand the reason behind the code change ("I would say it is complete").

Overall, the participants found the chains of bug reports formed by following the dependency links to contain information relevant to the code change. In case of *Chain#2* and *Chain#3*, the first bug report in the chain did not provide any relevant information. In case of *Chain#1* and *Chain#3*, the chain was missing some rather low-level detail related to the reason behind the code change.

Relevance of Summaries

Here are the participants responses when asked whether the chain summaries provided information relevant to the code changes (*Q2*):

- *Chain#1*: The participant (*Participant#1*) found the summary to contain relevant information, but is missing some technical details ("the summary describes the high level requirement for making the change, but misses some of the technical details in bug #4").

- *Chain#2*: *Participant#1* thought the summary is relevant, but misses part of the information on the reason behind the code change ("the summary is missing the particular problems (identified in the UI review) though that prompted the code change").

- *Chain#3*: *Participant#2* found the chain summary to be not so relevant to the code change ("well, it contains the title of the relevant bug, but the rest of it seems unrelated; it doesn't have any detailed information about this code change").

- *Chain#4*: *Participant#2* found the summary highly relevant ("I would say the summary is highly relevant; it does a pretty good job of cutting out unimportant text; the one thing I think is missing is the mention of filtering on connectorKind").

Overall, with the exception of the summary of *Chain#3*, participants found the summaries to contain information related to the code changes. In

the case of *Chain#3*, one of the two bug reports in the chain was considered as irrelevant by *Participant#2*, contributing to the fact that the summary contained mostly irrelevant information. In all cases, summaries were missing some technical details related to the code change, often to be found in the last bug report in the chain. So while participants appreciated the presence of high level information in the summaries, they wanted to see more low level technical information on the reason of the code change to be included in the summaries.

Post-session Questions

When asked whether they prefer consulting a chain or its summary when trying to understand the reason behind a code change, both participants showed an interest in working with summaries:

Participant#1: "if I had the summary linked in the development tool and instantly accessible I would probably consult it first (if it also linked to the original source of the information to drill into details)."

Participant#2: "I would love to have a summary like the second one (summary of *Chain#4*), as it can be quite time consuming to sift through many comments on multiple bug reports looking for the relevant ones. The mental effort required to do that makes it very difficult to hold on to the details about the code that you have in your head when you start."

When asked where else (other than the chain of bug reports) they might look for relevant information:

Participant#1: "some of that information could probably be found on other linked bug reports that are not in the direct dependency chain; if the information on the bugs is not sufficient I would look for linked artifacts such as wiki pages or mailing list discussions".

Participant#2: "patches and other files attached to bug reports, comments on Gerrit code reviews, bug reports related to other commits to related code and source code comments".

They both also mentioned that they might contact the developer who had made the code change.

When asked for suggestions on how to improve the summaries, they both expressed an interest in having summaries linked to the original bug reports.

Participant#2 also mentioned the possibility of highlighting the summary sentences in the original bug reports. *Participant#1* suggested providing several levels of detail (e.g. short summary and long summary).

5.5.2 Threats

The main threat to the construct validity of our study was that the summaries for the Mylyn chains were generated by a classifier trained on a corpus of human-annotated chains from another project (CONNECT). The chains in the corpus are smaller in length compared to chains used in the study (386±157 words compared to 2082±880 words). Moreover, CONNECT chains have no or very few comments compared to Mylyn chains. Also, while the corpus was annotated by non-experts, the study was conducted with project experts as participants. These differences might be one of the reasons of why, according to the study participants, some relevant technical details were not included in the summaries.

5.6 Summary

In this chapter, we addressed a special case of multi-document summarization for natural language software artifacts. A multi-document summarization technique was proposed to generate a summary of a chain of project documents related to a code change. In an exploratory user study, professional developers who authored the code changes, found the summaries to be indicative of the reason behind the code change, but missing some of relevant technical details. The participants also pointed out the possibility of including more information (e.g., code comments and bug reports not in the dependency chain) as well as other linked artifacts (e.g., project wiki pages and email discussions) as the input of the summarization approach.

Chapter 6

Discussion and Future Work

In this chapter, we discuss various decisions we made in developing summarization approaches for software artifacts and in evaluating generated summaries. We also discuss future directions for extending the work presented in this dissertation.

6.1 Producing Natural Language Summaries

In this dissertation, we chose natural language text as the format of automatically produced summaries as this format is easy to read and understand and is flexible for including semantic information and domain knowledge. Another alternative is to extract indicative keywords or phrases from an artifact to form a keyword summary (similar to [25, 41]). A keyword summary is typically less informative compared to a paragraph-length natural language summary. However, keyword summaries may be more effective in cases where a developer only needs to know the gist of an artifact to answer an information need. A keyword summary for a software artifact can be generated by using topic modeling techniques such as Latent Semantic Indexing (LSI) [24] or Latent Dirichlet Analysis (LDA) [12] to mine latent topics amongst the natural language content of the artifact.

Using graphs or diagrams is another alternative for conveying summary-worthy information. Graphical representations are typically more precise and less ambiguous compared to natural language text. As an example, for a crosscutting code concern, a UML class diagram or an RDF graph (similar to Figure 4.5) could help represent some relationships between parts of the code belonging to the concern, such as method overriding. But, using these diagrams, it is not straightforward to represent abstract content of the concern summary. For example, the fact that a large fraction of methods

belonging to a concern override the same method, can not be represented in a UML diagram as easily as it can be represented in natural language text.

Different summary formats can complement each other by being used at the same time to represent information in a software artifact summary. For example, for a bug report, a natural language summary can be accompanied by a flowchart visualizing the information in the *steps to reproduce* section of the bug report. Having access to summary information in different formats may improve the performance of a software developer. For example, while performing a bug report duplicate detection task, a developer may quickly decide that a recommended bug report is not a duplicate based on a keyword summary or she may decide to switch to a paragraph-length natural language summary if more investigation is needed.

Previous efforts have considered the use of a visualization mode as well as natural language text to represent summary information (e.g., [18]). For software artifacts, future research is needed to investigate what summary format would be most beneficial for a software developer trying to address a particular information need and whether different formats can be integrated to achieve higher efficiency.

6.2 What Should Be Included in a Summary?

An automatically generated summary needs to convey the key information in the input. Different factors contribute to determining important information to be included in a summary. In supervised approaches (e.g., our approaches in summarizing bug reports and chains of project documents), rules to identify important information are learnt based on the analysis of a set of human-generated summaries. In unsupervised approaches (e.g., our approach in summarizing software concerns), a set of heuristics are used to determine the key information of the input.

The other contributing factor in determining summary-worthy information is whether the goal is to generate a general-purpose or a task-specific summary. General-purpose summaries are produced with no assumption made about the purpose of the summary whereas task-specific summaries are intended to help users in performing a certain task or addressing a partic-

ular information need. For example, the summarization approach we used in Chapter 3 to summarize bug reports, produces general-purpose summaries that can be used in any software task involving interaction with bug reports. The bug report summarizer extracts generically informative sentences because it has been trained on a corpus annotated with generic summaries. We took a different approach in multi-document summarization of a chain of project documents (Chapter 4). In this case, generated summaries are intended to help a developer address a specific information need (the reason behind a code change). Therefore, to annotate the corpus, annotators were asked to pick sentences that best described the reason behind a code change. In the case of crosscutting code concerns, we generated general-purpose summaries based on a set of heuristics to describe *what* and *how* of a concern.

While one benefit of general-purpose summaries is the ability to use them in various tasks, it is likely that summaries generated to be used in certain task settings or to answer particular information needs are more effective. For example, it is likely that a bug report duplicate detection task benefits more from a summary which focuses more on the symptoms of a bug rather than how it should be fixed. As another example, a bug report may be referenced during a software change task to understand how a similar fix was performed in the past [102]. In this case, a bug report summary is probably more helpful if includes more information on the fix. If several task-specific summaries are available for an artifact, either there should be a mechanism to automatically detect the task or users should have the option of choosing the appropriate summary. Future research is needed to investigate whether task-specific summaries are more effective in helping software developers compared to general-purpose summaries.

6.3 Task-based Evaluation of Summaries

In this dissertation, we have used both intrinsic and extrinsic evaluation approaches. For example, bug report summaries were evaluated both intrinsically be comparing them against gold standard summaries (Section 3.4) and extrinsically by using them in a task-based human evaluation (Section 3.5.2).

The intrinsic evaluations focus on measuring the quality of information content of a generated summary typically by comparing it with a a set of reference (often human-written gold standard) summaries. Intrinsic measures are easily reproducible and can be used to compare the performance of a new technique against previous approaches.

However, the ultimate goal in development of any summarization approach is to help end users perform a task better. A task-based evaluation needs to be performed to establish that automatically generated summaries are effective in the context of a task. Similarly, as our investigation throughout this dissertation was concerned with whether summaries could help software developers in addressing their information needs, our focus was on extrinsic evaluation of artifact summaries. Bug report and crosscutting concern summaries were extrinsically evaluated in the context of software tasks. The extrinsic evaluation of summaries generated for chains of software documents was conducted by asking project experts whether summaries answer the question of interest (the reason behind a code change). Instead of software tasks, activities like comprehension, question answering or relevance judging can be used as the context of extrinsic evaluation of software artifact summaries. For example, summaries of software artifacts can be assessed by developing a set of questions that probe a programmer's understanding of a software artifact via a summary versus investigation of the artifact itself. This style of evaluation is very common in the natural language processing community where numerous task-based evaluations have been performed to establish the effectiveness of summarization systems in a variety of tasks. The tasks range from judging if a particular document is relevant to a topic of interest [58], writing reports on a specific topic [62], finding scientific articles related to a research topic [99] and decision audit (determining how and why a given decision was made) [69]. Compared to task-based user studies, such evaluations are typically less expensive to conduct and therefore might facilitate future research in incremental tuning and improvement of the summarization approaches.

It is an open question whether there is a correlation between intrinsic measures and the usefulness of a summary in a task-based setting. For example, the bug report summarizer used in Section 3.3 has a pyramid

precision of .65. Summaries generated by this bug report summarizer were shown to be helpful in the context of duplicate bug report detection task. Yet it is not clear whether an improvement in the intrinsic measure (e.g., an increase from .65 to .75) would make summaries noticeably more helpful for a developer working on a duplicate detection task. These questions can be investigated as part of future research.

6.4 Improving Summaries by Using Domain-Specific Summarization Approaches

The summarization approaches proposed in this dissertation were shown to generate summaries that help developers. Given this initial evidence, future efforts may focus more on domain-specific features of software artifacts to improve generated summaries.

For example, the accuracy of the bug report summarizer may be improved by augmenting the set of generic features with domain-specific features. For instance, comments made by people who are more active in the project might be more important, and thus should be more likely included in the summary. As another example, it was noted by many of the participants in our task-based evaluation that 'steps to reproduce' in a bug report description helped them in determining duplicates. This information has also been identified as important by developers in a separate study [10]. The usefulness of bug report summaries might be improved if an indication of the availability of 'steps to reproduce' information be included in the summary so that a developer can refer to the original bug report in cases she needs this information in addition to the summary.

As another example, as part of the annotation process, we also gathered information about the intent of sentences, such as whether a sentence indicated a 'problem', 'suggestion', 'fix', 'agreement', or 'disagreement'. Similar to the approach proposed by Murray and colleagues [68] in using speech acts, this information can be used to train classifiers to map sentences of a bug report to appropriate labels. Then an abstractive summary can be generated by identifying patterns that abstract over multiple sentences. We leave the investigation of generating such abstractive summaries to future

research.

In case of crosscutting code concerns, more sophisticated analyses can be performed on the code to extract more data-flow information between concern elements. This information might help alleviate navigations a programmer might otherwise have to perform to learn the same information.

6.5 Summarizing Other Software Artifacts

In this dissertation, we developed techniques to generate summaries for two different types of software artifacts; bug reports as an example of natural language software artifacts and crosscutting code concerns as an example of structured software artifacts. One possibility for future research is to investigate summarization for other types of software artifacts. Examples include the summarization of requirement documents, UML diagrams, project's wiki pages and test suites. The content of all these software artifacts is a combination of structured and natural language information with some (e.g., requirement documents) being more natural language and some (e.g., test suites) being more structured. Different summarization techniques (e.g., abstractive vs. extractive) and different representation formats (e.g., natural language vs. graphical representations) might be needed to be integrated to account for the mixed content of software artifacts. Research is needed to determine what information should be included in summaries and to investigate how developers may benefit from summaries in performing software tasks.

6.6 Multi-document Summarization of Software Artifacts

We investigated a special case of multi-document summarization of software artifacts in Chapter 5 involving a chain of project documents containing information about a code change. However, considering the inter-connected network of software artifacts in a software project, multi-document summarization is quick to emerge even when a summary is to be generated for a single software artifact. For example, in the case of bug report sum-

marization, a bug report may be related to other bug reports via explicit dependency links or it may refer to certain modules in the code. Depending on the information need the summary intends to answer, multi-document summarization might be needed to generate a summary for a collection of related software artifact. Future research may investigate different applications of multi-document summarization of software artifacts to address various information needs. For example, different software artifacts can be related to a software change task including bug reports, pieces of code, and pages in the project's wiki. A multi-document summary of these software artifacts can serve as a concise summary of the entire task, enabling a developer to understand different aspects of the task (e.g., *what*, *how* and *why* of the task) at an abstract level.

The main challenges in multi-document summarization of software artifacts is to retrieve and abstractly model the network of software artifacts relevant to the information need. For example, we modeled the input of our multi-document summarization approach in Chapter 5 as a chain of project documents. The chain was formed by following explicit dependency links between documents. However, in many software projects, all related software artifacts are not explicitly linked together. Future research may investigate techniques like text mining and information retrieval (e.g., [57]) to retrieve a set of the most relevant documents to answer an information need. Moreover, it is not always possible to put the relevant documents in a total order to form a chain. We leave the investigation of such cases, where the input set of documents can be modeled as a graph, to future research.

Chapter 7

Summary

To answer an information need while performing a software task, a software developer sometimes has to interact with a lot of software artifacts to find relevant information. Sometimes, it only takes the developer a quick glance to determine whether an artifact contains any relevant information. Other times, typically when interacting with a lengthy software artifact, the developer has to read through large amounts of information and many details of the artifact.

In this dissertation, we proposed the use of automatically generated summaries of software artifacts to help a software developer more efficiently interact with them while trying to address an information need. Based on their content, software artifacts form a spectrum where software artifacts with mostly natural language content are at one end and software artifacts with mostly structured content are at the other end. We chose to focus on two different types of software artifacts to address both ends of this spectrum; bug reports as an example of natural language software artifacts and crosscutting code concerns as an example of structured software artifacts. Since addressing an information need might involve making sense of a collection of software artifacts as a whole, we also studied a case of multi-document summarization of natural language software artifacts; a chain of project documents related to a code change.

We developed summarization techniques for each of the three cases mentioned above. For bug reports, we used an extractive approach based on an existing supervised summarization system for conversation-based data. For crosscutting code concerns, we developed an abstractive summarization approach. For multi-document summarization of project documents, we developed an extractive supervised summarization approach. In each case, to establish the effectiveness of generated summaries in assisting software devel-

opers, the generated summaries were evaluated extrinsically by conducting user studies. Summaries of bug report were evaluated in the context of bug report duplicate detection tasks. The result showed that summaries helped participants save time completing a task without degrading the accuracy of task performance. Summaries of crosscutting code concerns were evaluated in the context of software code change tasks. The results showed that summaries helped participants more effectively find pertinent code related to the change tasks and that summaries made programmers perceive a task as less difficult and their performance as more successful. Summaries of chains of project documents were evaluated by a number of project experts who found summaries to contain information describing the reason behind corresponding code changes.

This dissertation makes a number of contributions. For bug reports:

- It demonstrates that it is possible to generate accurate summaries of individual bug reports.
- It reports on the creation of an annotated corpus of 36 bug reports chosen from four different open source software systems. The recent approches proposed for summarizing bug reports have used this corpus to evaluate automatically generated bug report summaries [56, 59].
- It demonstrates that while existing classifiers trained for other conversation-based genres can work reasonably well, a classifier trained specifically for bug reports scores the highest on standard measures.
- It reports on a task-based evaluation that showed bug report summaries can help developers working on duplicate detection tasks save time without impairing the accuracy of task performance.

For crosscutting code concerns:

- It introduces the concept of using generated natural language summaries of concern code to aid software evolution tasks.
- It introduces a summarization technique to produce such natural language summaries automatically from a description of the code that contributes to a concern.

- It reports on a tool developed to allow programmers to access and interact with concern summaries for Java programs.
- It reports on a task-based evaluation that showed such natural language summaries can help a programmer find pertinent code when working on software change tasks.

For project documents related to a code change:

- It proposes the use of multi-document summaries to assist developers in addressing an information need when different pieces of information from different artifacts needs to be put together.
- It introduces a summarization technique to generate a summary of a chain of project documents related to a code change.
- It reports on a user study in which summaries of chains of project documents were found to be effective in describing the reason behind related code changes as judged by project experts.

Overall, the main contribution of this dissertation to the field of software engineering is an *end-to-end* demonstration that reasonably accurate natural language summaries can be automatically produced for different types of software artifacts: bug reports (as an example of mostly natural language software artifacts), software concerns (as an example of mostly structured software artifacts) and a chain of documents related to a code change (as an example of inter-related natural language software artifacts) and that the generated summaries are effective in helping developers address their information needs. This contribution serves as a proof-of-concept that summarization of software artifacts is a promising direction of research in the field of software engineering and that summaries of software artifacts may help developers in a variety of software tasks.

Main directions for future work include improving the proposed summarization techniques by utilizing domain-specific structure and content of software artifacts. Other possible directions are to develop summarization systems for other types of software artifacts (e.g., project wiki pages, requirement documents, design diagrams) and to explore multi-document summarization of software artifacts when the input is an inter-linked network of

software artifacts of different types (e.g., a network of inter-linked bug reports and code elements).

Bibliography

[1] H. Abdi. Bonferroni and Sidak corrections for multiple comparisons. In N. J. Salkind, editor, *Encyclopedia of Measurement and Statistics*, 2007.

[2] Bram Adams, Zhen Ming Jiang, and Ahmed E. Hassan. Identifying crosscutting concerns using historical code changes. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 305–314, New York, NY, USA, 2010. ACM.

[3] Stergos Afantenos, Vangelis Karkaletsis, and Panagiotis Stamatopoulos. Summarization from medical documents: a survey. *Artif. Intell. Med.*, 33(2):157–177, February 2005.

[4] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM.

[5] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: a recommender system for debugging. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, 2009.

[6] Elisa L. A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design pattern rationale graphs: linking design to source. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 352–362, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Elisa L. A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *AOSD'02: Proceedings of*

the first international conference on Aspect-oriented software development, pages 120–126, 2002.

[8] Regina Barzilay and Kathleen R. McKeown. Sentence fusion for multidocument news summarization. *Comput. Linguist.*, 31(3):297–328, September 2005.

[9] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *CSCW'10: Proceeding of the ACM Conference on Computer Supported Cooperative Work*, pages 291–300, 2010.

[10] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[11] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful; really? In *ICSM'08: Proceedings of the IEEE International Conference on Software Maintenance*, pages 337 –345, 2008.

[12] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[13] Alexander W. J. Bradley and Gail C. Murphy. Supporting software history exploration. In *MSR'11: Proceedings of the IEEE Working Conference on Mining Software Repositories*, pages 193–202, 2011.

[14] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *ASE'04: Proceedings of the International Confefence on Automated Software Engineering*, pages 310–315, 2004.

[15] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *CSCW'10: Proceeding of the ACM Conference on Computer Supported Cooperative Work*, pages 301–310, 2010.

[16] Shilpa Bugde, Nachiappan Nagappan, Sriram K. Rajamani, and G. Ramalingam. Global Software Servicing: Observational Experi-

ences at Microsoft. In *ICGSE'08: Proceedings of the IEEE International Conference on Global Software Engineering*, pages 182 –191, 2008.

[17] Giuseppe Carenini, Gabriel Murray, and Raymond Ng. Methods for mining and summarizing text conversations. *Synthesis Lectures on Data Management*, 3(3):1–130, 2011.

[18] Giuseppe Carenini, Raymond T. Ng, and Adam Pauls. Interactive multimedia summaries of evaluative text. In *IUI'06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 124–131, 2006.

[19] Giuseppe Carenini, Raymond T. Ng, and Xiaodong Zhou. Summarizing emails with conversational cohesion and subjectivity. In *ACL-08: HLT: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 353–361, New York, NY, USA, 2008. ACM.

[20] Jean Carletta, Simone Ashby, Sebastien Bourban, Mike Flynn, Thomas Hain, Jaroslav Kadlec, Vasilis Karaiskos, Wessel Kraaij, Melissa Kronenthal, Guillaume Lathoud, Mike Lincoln, Agnes Lisowska, and Mccowan Wilfried Post Dennis Reidsma. The ami meeting corpus: A pre-announcement. In *MLMI'05: Proceedings of Machine Learning for Multimodal Interaction: Second International Workshop*, pages 28–39, 2005.

[21] Y.-W. Chen and C.-J. Lin. Combining svms with various feature selection strategies. In *Feature extraction, foundations and applications*, pages 315–324. Springer, 2006.

[22] Brian de Alwis and Gail C. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE. In *VL-HCC'06: Proceedings of the International Conference on Visual Languages and Human-Centric Computing*, pages 51–54, 2006.

[23] Brian de Alwis, Gail C. Murphy, and Shawn Minto. Creating a cognitive metric of programming task difficulty. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Softtware Engineering*, pages 29–32, 2008.

[24] Scott Deerwester, Susan T. Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic

analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[25] Mark Dredze, Hanna M. Wallach, Danny Puller, and Fernando Pereira. Generating summary keywords for emails using topics. In *IUI'08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 199–206, 2008.

[26] Marc Eaddy, Thomas Zimmermann, Kaitin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[27] Noemie Elhadad, Min-Yen Kan, Judith L. Klavans, and Kathleen McKeown. Customization in a unified framework for summarizing medical literature. *Artificial Intelligence in Medicine*, 33(2):179–198, February 2005.

[28] Noemie Elhadad, Kathleen Mckeown, David Kaufman, and Desmond Jordan. Facilitating physicians' access to information via tailored text. In *In AMIA Annual Symposium*, page 07, 2005.

[29] Güneş Erkan and Dragomir R. Radev. LexPageRank: Prestige in Multi-Document Text Summarization. In *EMNLP'04: Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 365–371, 2004.

[30] Günes Erkan and Dragomir R. Radev. LexRank: graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22(1):457–479, December 2004.

[31] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[32] Tom Fawcett. ROC Graphs: Notes and Practical Considerations for Researchers. Technical report, HP Laboratories, 2004.

[33] J.L. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[34] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change

Extraction. *IEEE Transactions on Software Engineering*, 33(11):725 –743, nov. 2007.

[35] Linton C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1979.

[36] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 175–184, 2010.

[37] Zachary P. Fry, David Shepherd, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verbdirect object pairs in all the right places. *IET Software*, 2(1):27–36, 2008.

[38] Xiaodong Zhou Giuseppe Carenini, Raymond T. Ng. Summarizing email conversations with clue words. In *WWW '07: Proceedings of the 16th International World Wide Web Conference*, 2007.

[39] Jade Goldstein, Vibhu Mittal, Jaime Carbonell, and Mark Kantrowitz. Multi-document summarization by sentence extraction. In *NAACL-ANLP-AutoSum'00: Proceedings of the NAACL-ANLPWorkshop on Automatic summarization - Volume 4*, pages 40–48, 2000.

[40] Udo Hahn and Inderjeet Mani. The challenges of automatic summarization. *Computer*, 33(11):29–36, 2000.

[41] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE'10: Proceedings of the 17th IEEE Working Conference on Reverse Engineering*, pages 35 –44, 2010.

[42] S.G. Hart and L. Staveland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Human mental workload*, pages 139–183. P.A. Hancock and N. Meshkati (Eds.), Amsterdam: Elsevier, 1988.

[43] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *ASE'11: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527, 2011.

[44] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *ICSE*, pages 232–242, 2009.

[45] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *DSN'08: Proceedings of The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 52–61, 2008.

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[47] Bryan Klimt and Yiming Yang. Introducing the Enron Corpus. In *CEAS '04: Proceedings of the First Conference on Email and Anti-Spam*, 2004.

[48] A.J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353, 2007.

[49] B. Kolluru, Y. Gotoh, and H. Christensen. Multi-stage compaction approach to broadcast news summarisation. In *Interspeech '05-Eurospeech: Proceedings of the 9th European Conference on Speech Communication and Technology*, pages 69–72, 2005.

[50] Julian Kupiec, Jan O. Pedersen, and Francine Chen. A trainable document summarizer. In *SIGIR'95: Proceedings of the International ACM SIGIR conference on Research and development in information retrieval*, pages 68–73, 1995.

[51] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[52] Janice Langan-Fox, Chris Platania-Phung, and Jennifer Waycott. Effects of advance organizers, mental models and abilities on task and recall performance using a mobile phone network. *Applied Cognitive Psychology*, 20(9):1143–1165, 2006.

[53] Anna S. Law, Yvonne Freer, Jim Hunter, Robert H. Logie, Neil Mcintosh, and John Quinn. A comparison of graphical and textual presentations of time series data to support medical decision making in

the neonatal intensive care unit. *Journal of Clinical Monitoring and Computing*, 19:183–194, 2005.

[54] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In Stan Szpakowicz Marie-Francine Moens, editor, *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[55] Chin-Yew Lin and Eduard Hovy. The automated acquisition of topic signatures for text summarization. In *Proceedings of the 18th conference on Computational linguistics - Volume 1*, COLING '00, pages 495–501, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[56] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. In *ICSM'12: Proc. of the 28th IEEE International Conference on Software Maintenance*, 2012.

[57] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[58] Inderjeet Mani, David House, Gary Klein, Lynette Hirschman, Therese Firmin, and Beth Sundheim. The TIPSTER SUMMAC text summarization evaluation. In *Proceedings of the ninth conference on European chapter of the Association for Computational Linguistics*, EACL '99, pages 77–85, Stroudsburg, PA, USA, 1999. Association for Computational Linguistics.

[59] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 11:1–11:11, New York, NY, USA, 2012. ACM.

[60] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[61] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *WCRE'04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 132–141, 2004.

[62] Kathleen McKeown, Rebecca J. Passonneau, David K. Elson, Ani Nenkova, and Julia Hirschberg. Do summaries help? In *SIGIR'05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 210–217, 2005.

[63] Kathleen R. McKeown, Regina Barzilay, David Evans, Vasileios Hatzivassiloglou, Judith L. Klavans, Ani Nenkova, Carl Sable, Barry Schiffman, and Sergey Sigelman. Tracking and summarizing news on a daily basis with columbia's newsblaster. In *HLT'02: Proceedings of the second international conference on Human Language Technology Research*, pages 280–285, 2002.

[64] Qiaozhu Mei and ChengXiang Zhai. Generating Impact-Based Summaries for Scientific Literature. In *ACL'08: Proceedings of the Annual Meeting on Association for Computational Linguistics*, pages 816–824, 2008.

[65] Tom Mens, Juan Fernández-Ramil, and Sylvain Degrandsart. The evolution of Eclipse. In *ICSM'08: Proceedings of the 24th International Conference on Software Maintenance*, pages 386 –395, 2008.

[66] Rada Mihalcea and Paul Tarau. TextRank: Bringing Order into Text. In *EMNLP'04: Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 404–411, 2004.

[67] Gabriel Murray and Giuseppe Carenini. Summarizing spoken and written conversations. In *EMNLP'08: Proceedings of the 2008 Conference on Empirical Methods on Natural Language Processing*, 2008.

[68] Gabriel Murray, Giuseppe Carenini, and Raymond Ng. Generating and validating abstracts of meeting conversations: a user study. In *INLG'10: Proceedings of the 6th International Natural Language Generation Conference*, pages 105–113, 2010.

[69] Gabriel Murray, Thomas Kleinbauer, Peter Poller, Tilman Becker, Steve Renals, and Jonathan Kilgour. Extrinsic summarization evaluation: A decision audit task. *ACM Trans. Speech Lang. Process.*, 6(2):2:1–2:29, October 2009.

[70] Gabriel Murray, Steve Renals, and Jean Carletta. Extractive summarization of meeting recordings. In *Interspeech'05-Eurospeech: Proceedings of the 9th European Conference on Speech Communication and Technology*, pages 593–596, 2005.

[71] Gabriel Murray, Steve Renals, Jean Carletta, and Johanna Moore. Evaluating automatic summaries of meeting recordings. In *MTSE '05: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, Workshop on Machine Translation and Summarization Evaluation*, pages 39–52. Rodopi, 2005.

[72] Ani Nenkova and Kathleen McKeown. Automatic summarization. *Foundations and Trends in Information Retrieval*, 5(2-3):103–233, 2011.

[73] Ani Nenkova and Rebecca Passonneau. Evaluating content selection in summarization: the pyramid method. In *HLT-NAACL '04: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2004.

[74] Paul Over, Hoa Dang, and Donna Harman. DUC in context. *Information Processing & Management*, 43(6):1506–1520, 2007.

[75] Franois Portet, Ehud Reiter, Albert Gatt, Jim Hunter, Somayajulu Sripada, Yvonne Freer, and Cindy Sykes. Automatic generation of textual summaries from neonatal intensive care data. *Artificial Intelligence*, 173(78):789 – 816, 2009.

[76] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511 – 526, 2005.

[77] Vahed Qazvinian and Dragomir R. Radev. Scientific paper summarization using citation summary networks. In *COLING'08: Proceedings of the 22nd International Conference on Computational Linguistics - Volume 1*, pages 689–696, 2008.

[78] Dragomir R. Radev, Timothy Allison, Sasha Blair-Goldensohn, John Blitzer, Arda Celebi, Stanko Dimitrov, Elliott Drabek, Ali Hakim, Wai Lam, Danyu Liu, et al. MEAD-a platform for multidocument multilingual text summarization. In *LREC'04: Proceedings of the International Conference on Language Resources and Evaluation*, 2004.

[79] Dragomir R. Radev, Hongyan Jing, Małgorzata Styś, and Daniel Tam. Centroid-based summarization of multiple documents. *Information Processing & Management*, 40(6):919–938, 2004.

[80] Dragomir R. Radev, Jahna Otterbacher, Adam Winkel, and Sasha Blair-Goldensohn. NewsInEssence: summarizing online news topics. *Commununications of ACM*, 48(10):95–98, October 2005.

[81] Owen Rambow, Lokesh Shrestha, John Chen, and Chirsty Lauridsen. Summarizing email threads. In *HLT-NAACL'04: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2004.

[82] Sarah Rastkar and Gail C. Murphy. Why did this code change? In *ICSE'13: Proceeding of the 35th International Confernce on Software Engineering*, 2013.

[83] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 103 –112, 2011.

[84] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 505–514, 2010.

[85] Ehud Reiter, Somayajulu Sripada, Jim Hunter, and Ian Davy. Choosing words in computer-generated weather forecasts. *Artificial Intelligence*, 167:137–169, 2005.

[86] Martin P. Robillard. Automatic generation of suggestions for program investigation. *SIGSOFT Softw. Eng. Notes*, 30(5):11–20, September 2005.

[87] Martin P. Robillard. Tracking concerns in evolving source code: An empirical study. In *ICSM'06: Proceedings of the International Confefence on Software Maintenance*, pages 479–482, 2006.

[88] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30:889–903, 2004.

[89] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.

[90] Martin P. Robillard and Frédéric Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *Proc. of the OOPSLA Workshop on Eclipse Technology eXchange*, pages 65–69, 2005.

[91] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, 2007.

[92] Robert J. Sandusky and Les Gasser. Negotiation and the coordination of information and activity in distributed software problem management. In *GROUP'05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 187–196, 2005.

[93] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD'07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224, 2007.

[94] H. Grogory Silber and Kathleen F. McCoy. Efficiently computed lexical chains as an intermediate representation for automatic text summarization. *Comput. Linguist.*, 28(4):487–496, December 2002.

[95] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE'10: Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[96] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE'11: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253 –262, nov. 2011.

[97] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug

report retrieval. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 45–54, 2010.

[98] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *SOSP'07: Proceedings of the twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[99] Simone Teufel. Task-based evaluation of summary quality: Describing relationships between scientific papers. In *Automatic Summarization Workshop, NAACL*, pages 12–21, 2001.

[100] Jenine Turner and Eugene Charniak. Supervised and unsupervised learning for sentence compression. In *ACL'05: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 290–297, 2005.

[101] Marian van der Meulen, Robert H. Logie, Yvonne Freer, Cindy Sykes, Neil McIntosh, and Jim Hunter. When a graph is poorer than 100 words: A comparison of computerised natural language generation, human generated descriptions and graphical displays in neonatal intensive care. *Applied Cognitive Psychology*, 24(1):77–89, 2010.

[102] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE'03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, 2003.

[103] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: Visualization of code evolution. In *Softviz'05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 47–56, 2005.

[104] Stephen Wan and Kathy McKeown. Generating overview summaries of ongoing email thread discussions. In *COLING'04: Proceedings of the 20th International Conference on Computational Linguistics*, pages 549–556, 2004.

[105] Xiaojun Wan and Jianwu Yang. Multi-document summarization using cluster-based link analysis. In *SIGIR'08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 299–306, 2008.

[106] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 461–470, 2008.

[107] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall. SEON: a pyramid of ontologies for software evolution and its applications. *Computing*, 94:857–885, 2012.

[108] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall. Supporting Developers with Natural Language Queries. In *Proceedings of the 32nd International Conference on Software Engineering*, page to appear. IEEE Computer Society, May 2010.

[109] Jin Yu, Ehud Reiter, Jim Hunter, and Chris Mellish. Choosing the content of textual summaries of large time-series data sets. *Nat. Lang. Eng.*, 13(1):25–49, March 2007.

[110] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, November 2009.

[111] X. Zhu and G. Penn. Summarization of spontaneous conversations. In *Interspeech '06-ICSLP: Proceedings of the 9th International Conference on Spoken Language Processing*, pages 1531–1534, 2006.

Appendix A

Bug Report Corpus: Annotator Instructions

In this appendix, we provide the detailed instructions used by the annotators to annotate our bug report corpus (Section 3.2).

Annotator Instructions

General Task Overview:

In this task, your goal is to summarize bug reports and to annotate individual sentences for certain phenomena described below.

The Data:

The bug reports you will be summarizing and annotating are from a few open source software projects. Bug reports are categorized into two groups:

1. Defect: these bug reports report a defect when the system is not working as specified.
2. Enhancemnet: these bug reports are used to request a new feature in the system or a modification in the specified behavior of an existing one.

Some of the bug reports might have technical details referring to the code base of the underlying project. But you are not required to be familiar with that specific project in order to complete the task.

Annotation Details:

You will be presented with a bug report consisting of a description and several comments. Before carrying out any annotation, read through the bug report in its entirety to get a feel for the discussion. Once you have made a preliminary read-through of the bug report, you will proceed with annotation in three steps.

1. In the first step we ask you to write a summary of the entire bug report, using the text box at the top of the browser. You will author a single summary for the bug report, not a summary of each individual comment. The summary must be no longer than 250 words. It is therefore important to capture the important information of the bug report in a concise manner.

For each sentence in your written summary, we ask you to provide example sentence(s) from the original bug report that are related to the summary sentence. You can do this by indicating the bug report sentence IDs in brackets at the end of each summary sentence. For example:

"This error seems to happen when SessionStore is disabled or not correctly initialized.[1.1,1.3]"

"A one line code fix has been suggested for solving the issue.[2.2,5.3]"

Here the numbers 1.1, 1.3, 2.2, and 5.3 are sentence IDs from the bug report itself. Each sentence in your summary should have at least one such linked sentence at the end of it. There is no limit to how many sentences you may link in this fashion.

2. You will next have to produce another summary by selecting the important sentences in the original text; that is, the sentences from the bug report that you think are most informative or important. You can do this by clicking on the important sentences to highlight them. You can select as many sentences as you feel are necessary. There may be overlap with the sentences you linked in step 1, while you may also select sentences that were not linked in step 1. It is recommended that you at least select one third of all the sentences.

3. In the next step, you will mark individual sentences related to certain speech acts and other phenomena, which we will now define. Speech acts are sentences that perform a particular function. There are six types of

sentences to annotate.

1. **PROBLEM(Prob):** These are the sentences that desctibe a problem (an undesired observed behaviour) in the system. The following sentences would both be labeled as 'Prob':

   "JavaScript cannot be disabled with the new UI."

   "Scrolling inside iframe is not smooth."

2. **SUGGESTION(Sug):** In these senteces someone is expressing some idea on the ways in which the system should be modified. These modifications are either needed to fix a problem (in case of 'Defect' bug reports) or to add a requested feature (in case of 'Enhancement' bug reports). SUGGESTION sentences can take the form of a question, especially when the speaker is not sure that the idea or suggestion will be accepted. Examples:

   "It would be nice to have a warning when a feature.xml contains a plugin that depends on an unresolved plugin."

   "Haw about applying the Compose > HTML font selection to the Body Tag as a base font?"

   "Another approach is replacing StyledText.doDeletion() method."

3. **FIX(Fix):** These sentences are talking about how the problem has been actually fixed or how the requested feature has been added. For example:

   "This bug was marked as fixed because, on Bidi platforms, we were able to get RTL and LTR to work for Dialog (independent of the parent)"

   "Re-installing CDT fixed the problem(admittedly the daily build)."

4. **AGREEMENT(Agr):** These are the sentences when someone is making a comment in agreement with something that someone else has said. For example:

   "Correct, there is currently no way of undoing Show Filtered Children without selecting a node."

5. **DISAGREEMENT(Dis):** This is the opposite of 'Agr'. These are the sentences when someone is making a comment in disagreement with something that someone else has said.

   "I think you are going down the wrong path by assuming the solution is to re-implement TextMergeViewer from scratch."

6. **META(Meta):** Finally, we also ask you to annotate meta sentences. These are sentences that refer explicitly to this bug report:

   "If we continue this thread, it should be a new bug report."

   These six phenomena can be annotated simply by clicking the relevant buttons next to each sentence. Note that a given sentence can receive multiple annotations â€“ for example, it can be both a suggestion and an agreement sentence. When a button is clicked, it will change color to indicate your selection.

4. Finally, you will fill out a questionnaire about the bug report that you have just summarized and annotated.

Quotations:

You will notice that some comments contain both new text and quoted text. Quoted text is text from a preceding comment and is often indicated by the '>' character. For example:

> "
> *> If you see items in the location bar after you've deleted the history, don't*
> *> forget that bookmarks are now also shown in the popdown. That confuses people*
> *> that are used to older versions of Firefox.*
>
> *Is there a way to turn off this behavior for the location bar? Specifically, I*
> *don't want to see bookmarks in there.*
>
> "

Because of the presence of quoted text, some sentences will occur in several different comments in the bug report. When linking sentences in your summary, it is not necessary to link every occurrence of a given sentence in the bug report. Also, be aware that sometimes new text occurs in the middle of quoted text.

Recap:

Write your summary of 250 words or less. Create links between your summary sentences and sentences from the email thread. Then create another summary by selecting the most important sentences in the original bug report text.

Then annotate sentences for the following 6 phenomena:

1. **Problem**
2. **Suggestion**
3. **Fix**
4. **Agreement**
5. **Disagreement**
6. **Meta**

Figure A.1: Instructions used by the annotators of the bug report corpus

Appendix B

Task-based Evaluation of Crosscutting Code Concern Summaries: Supporting Materials

This appendix provides materials used in the task-based evaluation of crosscutting code concern summaries (Section 4.4), including the task descriptions used by the participants in the study and the concern summaries related to each task.

Task: Autosave

This task involves planning a change to jEdit's autosave feature. You will need to first familiarize yourself with JEdit and one of its features: *autosave*.

Overview of JEdit, Buffers, and Autosave

1. To run JEdit, right click on
   `org.gjt.sp.jedit`
   `JEdit.java`
   and select: Run As -> Java Application. Do not close JEdit.

2. In JEdit, an opened text file is called a *Buffer*. The following is an extract from the JEdit Manual (section 2.1):

   > "Several files can be opened and edited at once. Each open file is referred to as a *buffer*. The combo box above the text area selects the buffer to edit. Different emblems are displayed next to buffer names in the list, depending on the buffer's state; a red disk is shown for buffers with unsaved changes, a lock is shown for read-only buffers, and a spark is shown for new buffers which don't yet exist on disk."

3. JEdit has an *autosave* feature. The following is an extract from the JEdit Manual (section 3.3.1):

   > "The autosave feature protects your work from computer crashes and such. Every 30 seconds, all buffers with unsaved changes are written out to their respective file names, enclosed in hash ("#") characters. For example, `program.c` will be autosaved to `#program.c#`."

JEdit will also generate *backup* files, which are terminated with a tilde (~) character. These have nothing to do with your task in this study. You can completely ignore them.

Saving a buffer automatically deletes the autosave file, so they will only ever be visible in the unlikely event of a JEdit (or operating system) crash.

If an autosave file is found while a buffer is being loaded, jEdit will offer to recover the autosaved data. The autosave feature can be configured in the **Loading and Saving** pane of the **Utilities>Global Options** dialog box.

4. In the Loading and Saving pane, set the autosave frequency to 5 seconds.
5. Open the file `C:\Temp\test.txt`
6. Add a character to the file and do not save the file.
7. Look in `C:\Temp`. You should see the autosave file.
8. Save the test buffer in JEdit. The autosave file should disappear.
9. Add a character to the test buffer and do not save it. Wait 5 seconds.
10. Kill JEdit using the terminate button on the Eclipse console (the button with the red square).

11. Launch jEdit again. JEdit will attempt to recover the autosave file. Click yes. **ATTENTION:** A bug in the code of JEdit will cause the program to hang if you do not click yes or no in the recovery dialog before the time in the autosave frequency. To avoid this, just click yes or no before the 5 seconds (or whatever) of the autosave frequency are over. If the program hangs, you can kill it using the terminate button on the console. You do not have to worry about this bug for the study.

From a user perspective, that's all there is to the autosave feature. You can close JEdit now.

Change Request

You are to create a plan for performing the task described below. The plan should include the relevant program elements that need to be changed and how they should be changed. **NOTE:** You are not actually required to perform the changes. Rather you should identify the particular classes and methods to be used and describe any new classes or methods required. Use a text file (in e.g. Wordpad) to record your plan.

Change Task:

Modify the application so that the users can explicitly disable the autosave feature. The modified version should meet the following requirements:

1. jEdit shall have a checkbox labeled "Enable Autosave" above the autosave frequency field in the Loading and Saving pane of the global options. This checkbox shall control whether the autosave feature is enabled or not.
2. The state of the autosave feature should persist between different executions of the tool.
3. When the autosave feature is disabled, all autosave backup files for existing buffers shall be immediately deleted from disk.
4. When the autosave feature is enabled, all dirty buffers should be saved within the specified autosave frequency.
5. When the autosave feature is disabled, the tool should never attempt to recover from an autosave backup, if for some reason an autosave backup is present. In this case the autosave backup should be left as is.

During the task:

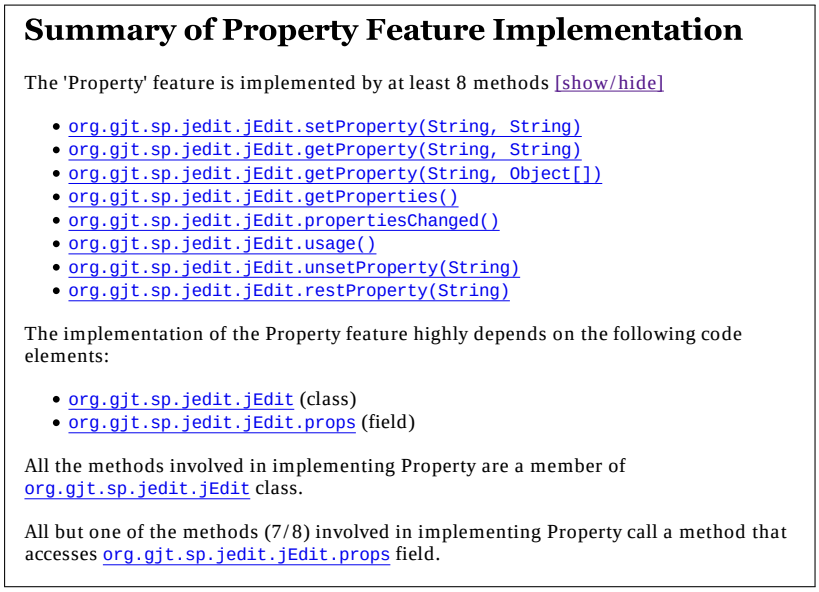
1. You must make **no change** to the source code. You are not allowed to perform temporary changes, or try out different alternatives.
2. Do not use the debugger.

Expert Knowledge

The starting point:
A checkbox should be added to `org.gjt.sp.jedit.options.LoadSaveOptionPane` to enable/disable the autosave.

Please notify the investigator when you are ready to commence.

Figure B.1: Description of the jEdit/Autosave task given to the participants in the study



Summary of Property Feature Implementation

The 'Property' feature is implemented by at least 8 methods [show/hide]

- org.gjt.sp.jedit.jEdit.setProperty(String, String)
- org.gjt.sp.jedit.jEdit.getProperty(String, String)
- org.gjt.sp.jedit.jEdit.getProperty(String, Object[])
- org.gjt.sp.jedit.jEdit.getProperties()
- org.gjt.sp.jedit.jEdit.propertiesChanged()
- org.gjt.sp.jedit.jEdit.usage()
- org.gjt.sp.jedit.jEdit.unsetProperty(String)
- org.gjt.sp.jedit.jEdit.restProperty(String)

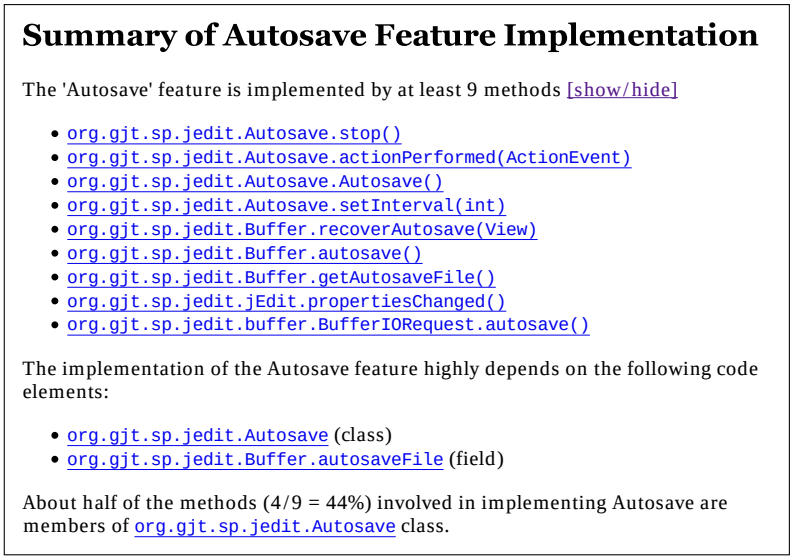
The implementation of the Property feature highly depends on the following code elements:

- org.gjt.sp.jedit.jEdit (class)
- org.gjt.sp.jedit.jEdit.props (field)

All the methods involved in implementing Property are a member of org.gjt.sp.jedit.jEdit class.

All but one of the methods (7/8) involved in implementing Property call a method that accesses org.gjt.sp.jedit.jEdit.props field.

Figure B.2: Summary of the 'Property' crosscutting concern



Summary of Autosave Feature Implementation

The 'Autosave' feature is implemented by at least 9 methods [show/hide]

- org.gjt.sp.jedit.Autosave.stop()
- org.gjt.sp.jedit.Autosave.actionPerformed(ActionEvent)
- org.gjt.sp.jedit.Autosave.Autosave()
- org.gjt.sp.jedit.Autosave.setInterval(int)
- org.gjt.sp.jedit.Buffer.recoverAutosave(View)
- org.gjt.sp.jedit.Buffer.autosave()
- org.gjt.sp.jedit.Buffer.getAutosaveFile()
- org.gjt.sp.jedit.jEdit.propertiesChanged()
- org.gjt.sp.jedit.buffer.BufferIORequest.autosave()

The implementation of the Autosave feature highly depends on the following code elements:

- org.gjt.sp.jedit.Autosave (class)
- org.gjt.sp.jedit.Buffer.autosaveFile (field)

About half of the methods (4/9 = 44%) involved in implementing Autosave are members of org.gjt.sp.jedit.Autosave class.

Figure B.3: Summary of the 'Autosave' crosscutting concern

Task: Undo

This task involves planning a change to the JHotDraw program.

Overview of JHotDraw

JHotDraw is a two-dimensional graphics framework for structured drawing editors.

1. To run a sample JHotDraw application, right click on
   `org.jhotdraw.samples.javadraw`
   `JavaDrawApp.java`
   and select: Run As -> Java Application.
2. Create a new file (using the File menu).
3. Try to add a few figures.

Change Request

You are to create a plan for performing the task described below. The plan should include the relevant program elements that need to be changed and how they should be changed. **NOTE:** You are not actually required to perform the changes. Rather you should identify the particular classes and methods to be used and describe any new classes or methods required. Use a text file (in e.g. Wordpad) to record your plan.

Change Task:

In the drawing editor, a user can change attributes of a figure using the Attributes menu. Your task is to implement the undo functionality for changing a figure's attributes. To check that undo is currently not supported:

1. Change the fill color (or any other applicable attribute) of a figure using the Attributes Menu (you have to first select the figure).
2. Try to undo the change by choosing Undo Command from the Edit menu.

During the task:

1. You must make **no change** to the source code. You are not allowed to perform temporary changes, or try out different alternatives.
2. Do not use the debugger.

Please notify the investigator when you are ready to commence.

Page **1** of **1**

Figure B.4: Description of the JHotDraw/Undo task given to the participants in the study

Summary of 'Undo' Feature Implementation

The 'Undo' feature is implemented by at least 22 methods [show/hide].

- org.jhotdraw.standard.SelectAllCommand$UndoActivity.undo()
- org.jhotdraw.figures.ConnectedTextTool$UndoActivity.undo()
- org.jhotdraw.figures.TextTool$UndoActivity.undo()
- org.jhotdraw.standard.SendToBackCommand$UndoActivity.undo()
- org.jhotdraw.standard.ConnectionTool$UndoActivity.undo()
- org.jhotdraw.figures.FontSizeHandle$UndoActivity.undo()
- org.jhotdraw.figures.UngroupCommand$UndoActivity.undo()
- org.jhotdraw.standard.ChangeConnectionHandle$UndoActivity.undo()
- org.jhotdraw.figures.RadiusHandle$UndoActivity.undo()
- org.jhotdraw.figures.PolyLineHandle$UndoActivity.undo()
- org.jhotdraw.contrib.PolygonScaleHandle$UndoActivity.undo()
- org.jhotdraw.standard.DragTracker$UndoActivity.undo()
- org.jhotdraw.figures.InsertImageCommand$UndoActivity.undo()
- org.jhotdraw.standard.AlignCommand$UndoActivity.undo()
- org.jhotdraw.standard.CutCommand$UndoActivity.undo()
- org.jhotdraw.contrib.TextAreaTool$UndoActivity.undo()
- org.jhotdraw.standard.DeleteCommand$UndoActivity.undo()
- org.jhotdraw.standard.PasteCommand$UndoActivity.undo()
- org.jhotdraw.figures.GroupCommand$UndoActivity.undo()
- org.jhotdraw.standard.ResizeHandle$UndoActivity.undo()
- org.jhotdraw.figures.BorderTool$UndoActivity.undo()
- org.jhotdraw.contrib.TriangleRotationHandle$UndoActivity.undo()

This feature provides 'Undoing' functionality for 'Select All Command', 'Connected Text Tool', etc.

The implementation of the Undo feature highly depends on the following code element(s):

- org.jhotdraw.util.UndoableAdapter.undo()
- org.jhotdraw.util.Undoable.undo()

All of the methods involved in implementing 'Undo':

- are named 'undo'
- override method org.jhotdraw.util.UndoableAdapter.undo()
- override method org.jhotdraw.util.Undoable.undo().
- are a member of a class named 'UndoActivity'
- appear with a method that overrides org.jhotdraw.util.Undoable.redo()
- appear with a method that overrides org.jhotdraw.util.UndoableAdapter.redo()
- appear with a method named 'redo'

All but one of the methods involved in implementing 'Undo':

- statically-calls org.jhotdraw.util.UndoableAdapter.undo()
- are a member of a class that extends-class org.jhotdraw.util.UndoableAdapter
- appear with a method that statically-calls org.jhotdraw.util.UndoableAdapter.UndoableAdapter(DrawingView)
- appear with a method that calls org.jhotdraw.util.UndoableAdapter.setRedoable(boolean)
- appear with a method that calls org.jhotdraw.util.UndoableAdapter.setUndoable(boolean)

All but one of the methods involved in implementing 'Undo':

- are a member of a class created by a method named 'createUndoActivity'

More than 2/3 of the methods appear with a method which calls one or more of the following methods:

- org.jhotdraw.framework.FigureEnumeration.nextFigure()
- org.jhotdraw.util.UndoableAdapter.getAffectedFigures()
- org.jhotdraw.framework.FigureEnumeration.hasNextFigure()
- org.jhotdraw.util.UndoableAdapter.isRedoable()

Around half of the methods call one or more of the following methods:

- org.jhotdraw.framework.DrawingView.clearSelection()
- org.jhotdraw.util.UndoableAdapter.getAffectedFigures()
- org.jhotdraw.framework.FigureEnumeration.hasNextFigure()
- org.jhotdraw.util.UndoableAdapter.getDrawingView()

Figure B.5: Summary of the 'Undo' crosscutting concern