TOWARD GENERATING COMMIT MESSAGES FOR SOFTWARE REPOSITORIES

by

Casey Casalnuovo

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Honors Bachelor of Science in Computer Science with Distinction

Spring 2013

TOWARD GENERATING COMMIT MESSAGES FOR SOFTWARE REPOSITORIES

by

Casey Casalnuovo

Approved: _____
Lori Pollock, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Vijay Shanker, Ph.D.
Committee member from the Department of Computer Science

Approved: _____
James Glancey, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____
Michael Arnold, Ph.D.
Director, University Honors Program

ACKNOWLEDGMENTS

First and foremost, I'd like to thank my Senior Thesis Advisor, Lori Pollock, for helping me through the research process and for all her assistance in writing my Thesis. I'd also like to thank my second and third readers, Vijay Shanker and James Glancey. Secondly, for assisting me with getting in contact with survey respondents and in improving the survey's content, I'd like to thank Jeff Carver, Nick Lacock, and Terry Harvey. I'd also like to thank both the University of Delaware students and developers who took the time to complete the survey. Finally, I'd like to thank my parents for their continuing support while I worked on completing my research and writing my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Poor quality documentation increases the time developers spend trying to understand and modify source code. Therefore, if documentation can be automatically extracted from words and phrases in source code, it can diminish maintenance costs when human-written documentation is poor. Commit messages are a type of documentation that specifically describes program change. While methods exist for both finding differences between versions and for extracting linguistic information from source code, there has been little work in producing output that uses both to produce natural language output similar to developer-written commit messages. In order to lay groundwork for such a model of output, in this Thesis we performed an observational study of commit messages from open source software projects to determine their linguistic and non-linguistic properties. We also sent out a survey to users of software repositories to learn about how they use commit messages, what kinds of commit messages they find useful, and to present an initial model of output for natural-language commit messages using verb phrases and their associated direct objects. We find this model is insufficient as it lacks important location information from the original commit messages, which is often found in prepositional phrases in the original messages. Finally, we performed an independent analysis on a distribution of DeltaDoc, a research tool which attempts to generate output to supplement developer written commit messages. We found this distribution to be too problematic to use as it is, but that its output has potential be extended using natural language techniques if the concerns about its usability and performance can be addressed.

Chapter 1

INTRODUCTION

A significant part of software engineering is *software maintenance*, which takes place throughout the software development cycle. It is defined in the IEEE standard as "the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage" [17]. Examples of such activities include optimizations of existing code, the tracing and removal of errors, or restructuring of code for easier use. Software maintenance comprises a significant component of software development, exceeding the time spent on the initial development of the system. Depending on the project, software maintenance takes anywhere between 40 to 80 percent of the developer's resources [12]. Therefore, improvements that affect the software maintenance process will have a significant positive effect on the overall quality of software.

1.1 Importance of Documentation and its Automatic Generation

Critical to assisting developers in understanding code, which is particularly needed during software maintenance is the human-written documentation surrounding the code. Previous research in software engineering has shown that when making changes to software, developers that are unfamiliar with the code being modified spend more time trying to understand code than actually implementing changes to the code[14]. The formal language of source code, designed for easy processing by machines, is not by itself designed to be easily read by humans. As such, software projects typically have a significant amount of documentation complementing their source code, which simplifies the task of understanding code. This documentation varies in scope, from low-level documentation such as comments embedded within the code, to higher

level documentation such as system requirements, use cases, etc. One type of documentation often used to describe changes between code versions are *commit messages*. These messages are typically no more than a few lines and allow developers to comment on changes made to the program. Since commit messages are used to specifically document program change, they relate closely to the various tasks carried out during software maintenance.

Although standards of documentation vary from project to project, documentation is often not very well written in practice [7]. It can be lacking or unavailable, or worse, misleading or out of date. Therefore, if documentation could be automatically created, developers would be able to make code changes effectively even in situations when human-written documentation is poor or missing.

Currently, while no research group has attempted to generate commit messages in natural language form, there has been significant work done in related areas. One is the area of file and version differencing, which seeks to extract the information about changes between two versions of a program[16, 18, 21, 31, 23, 3]. However, the output from such techniques are not as concise as developer-written commit messages and are not targeted at producing natural language output. Likewise, there has also been much work in extracting English language information from static source code distributions[22, 27, 14, 29], but these have not been applied to describing the changes between program versions. The most closely related work to automatic commit message generation in natural language is DeltaDoc, developed by Buse and Weimer [5]. However, the output of this program is closer to pseudocode than the English commit messages written by developers.

1.2 Thesis Contributions

This Thesis seeks to establish a grounding for generating meaningful natural language commit messages as a first step to summarizing the main intent of a commit. First, we performed a study of approximately 60,000 commit actions and their associated commit messages across 9 open source projects, investigating both their linguistic

and non-linguistic properties. We extracted verb phrases and their associated direct objects from these commit messages. Then, we launched a survey to users of software repositories and commit messages to learn about how commit messages are used. Survey respondents also evaluated the quality of commit messages and the effectiveness of the verb-direct object summarization as a potential form of natural language output. Finally, we performed an observational study on a distribution of DeltaDoc and its output. We evaluated the effectiveness of DeltaDoc's output in comparison to both our survey results and to the raw differences between the files summarized.

Chapter 2

BACKGROUND

2.1 Software Repositories

Software repositories are tools used by software developers to store source code and related materials (e.g. documentation, test cases, etc.) for a project. Software repositories are commonly used in software projects developed by more than one person, or in projects developed for long periods of time. Significant applications cannot be developed quickly and instead change over time as developers make incremental changes to the code. Documentation is updated as new features are added, errors are corrected, or code is optimized. While software develops over time, at any given time, a static set of code can be marked as a specific version of an application. Repositories assist developers by keeping track of all the code versions that exist over the lifetime of a project. The repository format provides a convenient method for tracking changes in the code and secondary artifacts made by members of a software development team, ensuring team members can access the changes made by others efficiently. Changes are recorded and developers can return to prior versions if a change introduces serious bugs. Examples of software repositories include free projects such as SVN, CVS, Git, Mercurial, as well as some proprietary systems.

2.2 Commit Messages

While there are many implementations of software repositories, all of them operate under roughly the same model. While the term software version often applies only to official releases of a software project, each change added to a software repository is called a *commit*, often referred to as committing a change. With each commit added to a repository, developers have an opportunity to document the changes they made in a

short message called a *commit message*. These messages are typically only a sentence or two, but they can be longer. The message should document the changes clearly and concisely, but often the messages are too short or vague to convey useful information about the commit. Below are a few examples of commit messages that qualitatively seem to be good or poor, respectively:

Examples of well-written commit messages from our data set:

- Added preliminary, ugly VectorKeyStrokeOptionComponent that can handle multiple keystrokes per action.
- Fixed ArrayIndexOutOfBounds exception on Player.stance[] when attacking an enemy privateer
- Logger now sanity checks the verbosity level param. Listener now has verbosity and log round the right way where it was crashing phex. SwarmingManager now makes backup copies of the download file. This should make us more robust against going down unexpectedly while writing this file.

Examples of poorly-written commit messages from our data set:

- XML and XSL stuff
- several updates
- fixed problems
- Refactoring

Additionally, commit messages can serve different purposes to different audiences. The primary audience for commit messages are other members of the development team working on the project. The commit messages act as a way for developers to quickly see what changes have been made. However, researchers studying software evolution also make use of commit messages to better understand how software develops over its lifetime. While commit messages can act as a record of change in a project, some software evolution researchers claim that the granularity of change summarized by commit messages is too large to be effective for their purposes. Additionally, the poor quality and sometimes the outright lack of commit messages hamper their ability to act as a record of the changes in a software's lifetime [24].

2.2.1 Alternatives to Commit Messages

Some researchers have considered using alternative methods to gain information about the changes software undergoes. For example, the Eclipse add-ons Syde and Scamp work by giving the smallest granularity possible of change possible. They broadcast changes made in real time to other developers on the project [20]. They are designed with the focus of increasing team awareness, i.e. its goal is to raise awareness about who is working on what files to help prevent conflicts. It is not clear that it is appropriate as a method to replace commit messages. Other groups have suggested alternative means to display the information made in a commit message. For instance, the tool Commit 2.0, created for SmallTalk, an object-oriented programming language, creates a visual graph of the commit. This visually displays the changes of a commit as an complement to developer-written messages. The program automatically identifies and displays both new, deleted, and modified sections of code. These can be seen at different granularities, such as the package, class, or method levels. Additionally, the tool tries to identify not only the areas that changed, but also other structures in code that use these changed areas. Including this context gives developers a better idea about what code might have been affected by the changes. However, this tool is intended to assist the understanding of existing documentation, not replace it [6].

2.3 Summary

While commit messages are known to be helpful, developers often do not write them or write minimal phrases are that not helpful to readers. There are some alternative techniques designed to assist understanding of code changes, but commit messages continue to be used as the most common method of change documentation in software repositories. However, how frequently commit messages are used is an important question, which we will return to address Chapter 5.

Chapter 3

STATE OF THE ART

The problem of automatically generating commit messages is essentially a combination of two distinct subproblems. The first problem is how to extract differences between two versions of an application. The second problem is figuring out how to display this information in an natural language format of no more than a few sentences. Both of these areas have received some attention in research, but their have been only a few limited attempts to combine them to generate human readable documentation.

3.1 File and Version Differencing

At the most basic level, file differencing works by comparing two files and creating some sort of output about the differences between the two files. It differs from the problem of creating commit messages in that it usually is not concerned with summarizing the changes in a few English language sentences, and in the general case, it doesn't necessarily involve source code. File differencing is often used to see what has changed between two versions of the same file. The simplest form of this technique is the UNIX *diff* command, which takes two files and outputs line by line the differences between two files, regardless of what is stored in the file.

Figure 3.1 shows a sample of the diff command's output for two very simple Java files, which were changed by adding one new line and removing another:

DiffExample1.java

```
public class DiffExample {

   public static void main(String args)
   {
      int x = 5;
      int y = x + 3;

      System.out.println(x);
      System.out.println(y);
   }
}
```

DiffExample2.java

```
public class DiffExample {

   public static void main(String args)
   {
      int x = 5;
      int z = x - 2;
      int y = x + 3;

      System.out.println(y);
      System.out.println(z);
   }
}
```

Example Diff output:

```
5a6
>       int z = x - 2;
7,8c8
<
<       System.out.println(x);
---
>
9a10
>       System.out.println(z);
```

Figure 3.1: Example output of diff for two test Java files

However, for the purposes of acting as a substitute for commit messages, UNIX diff output is lacking. A commit message summarizes changes between versions, and the changes being summarized may encompass multiple files and many changed lines of code. The diff output, on the other hand, gives virtually no summarization and is often confusing to read without having both files open to understand the context of what changed. The more changes made, the longer and less convenient the diff output becomes for tracking down changes.

Many researchers have noticed the limitations of diff output, and consequently, there have been several attempts to develop more sophisticated differencing algorithms to give better output for software developers interested in understanding code changes. Susan Horwitz addressed the problem of differencing source code rather than general files, and introduced an algorithm that could be used to find differences in simplistic languages [16], but is not sufficient for full object-oriented programs. *Semanticdiff* is another differencing tool, but its scope is limited to measuring the changes in input/output behavior of a single function [18]. Another group was able to determine file differences in C/C++ by translating the code into an XML representation called src XML and using existing XML analysis techniques to extract information about the source code differences [21].

In addition to these tools, there are some version differencing techniques that seek to specifically take advantage of the structure of object-oriented languages. One such differencing tool is UMLDiff [31], implemented via the Eclipse plugin JDEvAN, which focuses on file differencing between two different versions of the object-oriented language Java. It takes two separate versions of a Java project as input, and then builds a tree to represent the structural changes made between the two Java versions. It records what classes, methods, fields, and other attributes were added/removed, moved, or renamed between the two versions. However, it suffers from some issues when the scope of changes is quite large, and can take 30 - 50 minutes to record the changes between major releases of a program.

Ren et al, focused on *change impact analysis*, which is the practice of determining what effects a change has had on other parts of a program, and developed a tool, Chianti, for Java [23]. Chianti breaks down the changes to a Java program into a series of atomic changes. It requires the existence of extensive unit testing of the source code, as it maps how each of the changes it has extracted from the program will affect the existing tests.

Another version differencing algorithm focused on finding differences in object oriented source code is CalcDiff, which is implemented in the Eclipse add-on JDiff. [3]. CalcDiff operates by first identifying classes and interfaces and sorting out which are added/removed or modified. It then identifies changes within methods and traces program behavior within the context of both the execution of these methods and the larger context of overall object-oriented design. In this way CalcDiff focuses on lower level changes than UMLDiff, as it does not lean so heavily on structural modifications.

3.2 Comment Generation

Some work has been done in the area of creating documentation for a single static version of a program. Sridhara et al. created a tool for the purpose of generating Javadoc comments [29]. Javadoc comments are associated with a single method within a Java class, usually giving a descriptive summary of the purpose of the method. This technique uses the Software Word Usage Model (SWUM) for Java [14] as the basis for extracting meaningful words and phrases from source code and then applies a three step process to generate a comment. First, lines of significant code for the summary are selected using various heuristics. Then, for each line of code selected an English summary is generated. Finally, each of these phrases is combined and further summarized to generate a natural language summary comment for the overall method.

3.2.1 Natural Language Processing and SWUM

The techniques SWUM uses to parse source code falls under the larger area known as *natural language processing (NLP)*, the interpretation of language by machines. This is often domain specific, and prior research [22] has determined that there can be great benefit in using English words and phrases within source code to aid in program understanding. These techniques are called *Natural Language Program Analysis (NLPA)*, and work by combining knowledge of the structure of the English language with knowledge about the structure of source code, in order to meaningfully extract information from the code to aid in the process of software maintenance.

In terms of analyzing source code for linguistic structures, not all programming languages are equally useful, and depending on the structure of the language, techniques for mining information may vary significantly. In particular, Java has been a focus of research due to its characteristics [14]. For one, it is an example of *Object-oriented programming (OOP)*, which functions by using pieces of code called *classes* which serve as a template for the eponymous objects. These *objects* can be viewed as nouns in the context of the larger program, and they have various attributes associated with them, which may be simple data such as integers or other objects. Additionally, functions written in a class are considered the class's *methods*. These functions are equivalent to verbs or actions in OOP, and they take various objects (the nouns) as input and can return them as output [14].

In addition to being an object oriented language, Java is a good candidate for linguistic analysis for a couple of other reasons. First, it is a well-known and widely used language, ensuring a large corpus of code to analyze, but also ensuring that tools developed to assist Java developers will be relevant in a wide range of situations. Additionally, Java also has a reputation as a "wordy" language, with developers often using long and descriptive names for classes and methods. For the purposes of linguistic analysis, it makes Java a compelling source of information to retrieve, analyze, and display extracted information.

Hill et al. worked to develop a model of linguistic representation of Java code

called the Software Word Usage Model (SWUM), that improves on more simplistic frequency-based methods, like bag of words [14]. The SWUM model was designed to aid in searching source code for concerns, which are "anything stakeholders of the software consider to be a conceptual unit, such as features, requirements, design idioms, or implementation mechanisms" [25]. The key insight of the SWUM model is that the context in which words occur in software is extremely important. SWUM preprocesses the source code to accurately extract English words, using a tool called Samurai to parse words out of method names, even when they are not using standard camel case, e.g. getlist() instead of getList(). SWUM also expands common abbreviations within source code. Once these words are extracted, the model is built by considering how the words occur together in the code.

3.3 Commit Message Generation and Classification

In contrast to the more extensive research in version differencing and linguistic parsing of source code, the automatic generation of commit messages for software repositories is a topic that has received little direct attention. The most directly related tool, DeltaDoc, was created by Buse and Wiemer to create commit messages for Java source code [5]. Their tool operates by matching additions and removals at the class and method level, and then using symbolic execution to build control flow graphs for changed methods. From these, they extract *path predicates*, which describe the conditions under which code statements will be executed. They compare the predicates in the old and new versions and use the differences as a basis for their output. This output can be too long, so they then apply a series of lossy transformations to summarize the code changes. However, they use only very basic natural language processing techniques to improve the output, and what is produced resembles pseudocode more so than English. Although the study they performed found that these messages aided in understanding the changes and could supplement developer-written commit messages, they do not match what is typically understood when discussing commit messages. We perform a more detailed analysis of DeltaDoc with examples in Chapter 6.

One other tool for assisting in commit message generation is the plugin Mylyn [19]. It allows for commit messages to be created from user defined tasks. These tasks are conceptual units of work that developers complete while working on any project, such as adding a feature or fixing an error. However, since the messages generated from Mylyn are ultimately based on these developer-described tasks, it is subject to the same problems seen in other forms of human-written documentation - it merely pushes the issue to a different area. Moreover, the commit message generation feature is not the main focus of the plugin.

N. Dragan, et al. [9, 10, 8] developed a technique for categorizing commits based on distributions of *method stereotypes*. The stereotype of a method gives an abstraction of its purpose into one of several types. When a commit is made to a software repository, they measure how the method stereotypes have changed to create a *commit signature*, an estimation of the overall purpose of that commit. Rather than trying to identify the specific changes, these stereotypes provide insight into the overall purpose of the commit message. Some examples of the categories they created to describe commits include *relationships modifier commits*, which alter how classes relate to each other, *degenerate modifier commits*, which contain empty methods - possibly signaling a new feature will be added, or *state update modifier commits*, which signal a change in an object's internal behavior. These classifications differ from those generally found in the existing literature [15, 2] for categorizing types of maintenance changes, which are based around modifying or adapting categories of maintenance developed by Swanson in 1976 [30].

3.4 Limitations and Identified Improvements

While there are several techniques for extracting information about differences in files and versions of code, the output of such methods tend to be extensive. The output does not resemble what is usually viewed as developer-written commit messages. Likewise, while there exist techniques for extracting natural language information from source code, they are not currently targeted at measuring program changes.

Chapter 4

CHARACTERISTIC STUDY OF COMMITS AND COMMIT MESSAGES

We performed a study of commit messages on a variety of open source software projects to obtain an understanding the quality and form of commit messages. An important first step towards successfully generating commit messages in English is to understand both the general form and linguistic structure of developer written commit messages. This study was directed at answering the following questions:

- How are commits distributed among developers in open source projects? Are they spread evenly among many authors or are the majority of commits made by only a few developers?
- What is the appropriate length for a commit message? How long are the commit messages that developers typically write?
- What is the scope of program change for each commit? How many source code files are typically changed, added, or removed?
- What is the linguistic composition of commit messages? What parts of speech are used to begin messages? What does the structure of the whole commit message look like?

Finally, we present a technique for extracting verbs and their direct objects as a first step toward summarizing the content of developer written commit messages, and gather feedback from developers on those summaries. This feedback is presented in Chapter 5.

4.1 Subjects of Study

All the messages chosen for this study have been selected from a set of nine Java projects: Drjava, iText, Phex, JFreeChart, Jedit, Freecol, Squirrel-sql, Java-game-lib,

Table 4.1: General Software Repository Statistics

Project	Revisions*	Developers	Type	Downloads*	Lines of Code
Atunes	5000	3	Audio player	1627500	63000
DrJava	5400	59	Java code editing	1570600	91000
Freecol	7900	39	Game	1562800	117000
iText	4900	16	PDF utility	1562800	124000
Java-game-lib	3700	17	Game Library	674300	136000
Jedit	20400	119	Text	6942100	207000
JFreeChart	2400	4	Data Presentation	3309000	147000
Phex	3500	24	File Sharing	1127200	107000
Squirel-sql	6500	18	SQL Client	3926000	313000
Total	59700	297	—	23517500	130500

and Atunes. Table 4.1 summarizes some of the relevant characteristics of each project. Four of these projects (IText, Phex, JFreeChart, and Freecol) were selected from the set of 5 analyzed in the study of commits performed by Buse and Weimer as their baseline for building DeltaDoc, which were chosen in order to give us some ability to compare to their study on commit messages [5]. The other five projects that were selected also came from svn repositories. Moreover, all the projects chosen include Java code so that future studies on the source code will be able to use prior methods for linguistic analysis of Java. The other criteria used to select these projects were that the projects had a significantly sized repository of at least a few thousand commits and that the projects are sufficiently popular, i.e. they have had at least several hundred thousand downloads over their lifetimes. Furthermore, the projects were chosen from a wide variety of applications to ensure a range of potentially different commit messages. The commit messages were obtained from the projects by either checking out a local copy and parsing the svn log files, or by screen scraping the messages individually from Sourceforge's website [28].

4.2 Threats to Validity

As mentioned previously, all commit messages were obtained from svn repositories. However, it is possible that commit message content and form might differ in

Table 4.2: Classification of Developers by Number of Commits (x) Made Over All Projects

Project	# Developers	Categorization of Developers by Number of Commits Made				
		x<50	50<=x<100	100<=x<500	500<=x<1000	x>1000
Atunes	3	0	0	0	1	2
DrJava	59	38	7	10	4	0
Freecol	39	25	5	6	1	2
iText	16	10	0	4	1	1
Java-game-lib	17	12	0	3	0	2
Jedit	119	76	20	16	1	6
JFreeChart	4	3	0	0	0	1
Phex	24	21	1	1	0	1
Squirel-sql	18	11	3	1	1	2

from users of Git, CVS, or other software repositories. Additionally, all the projects selected used primarily Java code. Programmers who favor different languages may have different attitudes toward documentation, so this study may not generalize to repositories for all projects. Finally, all of the commit messages were chosen from open source projects. It is likely that closed sourced projects may have stricter rules about documentation than open source projects. Therefore, the results of this study cannot be applied to closed source projects.

4.3 Non-linguistic Analysis of Commit Messages

In the following subsections, we discuss our findings about the distribution of commit activity among authors, the length of commit messages, and the scope of changes made in each commit.

4.3.1 Developer Behavior

There were a couple of patterns in committing behavior across all projects, and Table 4.2 classifies the developers in each project by the number of commits they made to the project. First, in every case other than Atunes, the majority of the developers working on the project made less than 50 commits. However, the overall contribution of this group was small, with a majority of commits being made by a much smaller

group of developers. Additionally, a few of the applications seemed to be personal projects; they were dominated by one committer. This was most clear in the cases of JFreechart and Phex, where a single developer was responsible for 98.2% and 85.6% of the commits respectively. Although most of the commits were made by a subset of the total developers in the other projects as well, the degree to which this was focused on just a few developers varied depending on the project. For instance, DrJava and Jedit had the most spread out distributions, with more authors contributing equally to the project. These contrast with a project like Squirrel-sql, which had only 4 developers with over 100 commits.

In the case of larger open source projects, a detailed study on the structure of developer interactions was performed by Christian Bird, et. al. They discovered that developers in open source projects tend to self-organize into groups. Specifically, these groups were more modular when working on shared sections of the program and more broad when dealing with application-wide changes. Pairs of developers within each group tended to modify the same files more frequently than pairs of developers in different groups [4].

Therefore, since many open source projects are team based, and since team members tend to specialize into subgroups even in open source projects, it seems that developers whose files will be most affected by a commit will also be most familiar with the code changed. Therefore, commit message's core audience can be assumed to have a good knowledge of the code affected. However, as our data shows, there may be large numbers of developers who commit less regularly. Or, even if they do commit regularly, they may work in different sections of the code. As such, it would likely be useful to include information in commit messages that allows this secondary audience to assess how code changes may affect them without needing to know the details of the local implementation.

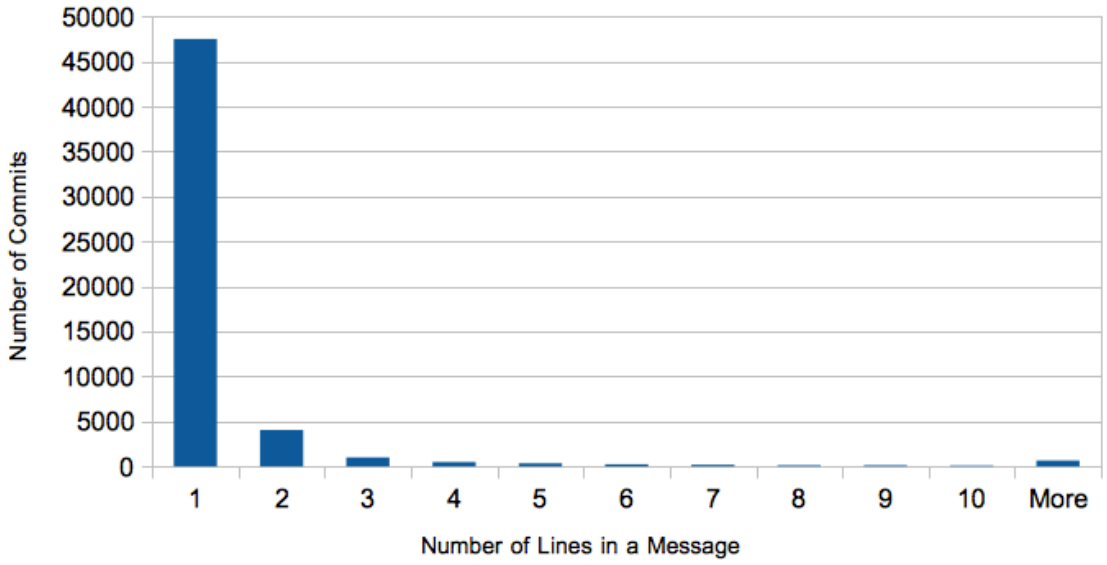

Figure 4.1: Distribution of Commit Message Lengths Over All Projects

4.3.2 Length of Commit Messages

One major consideration for creating commit messages is how long a commit message should be. To measure the length of a measure, we first used the metric of physical lines in the message log. Figure 4.1 shows the distribution of commit message size in lines across all projects. We found that 87.29% of the commit messages were only a single line, and 96.22% were 3 lines or less. The prevalence of single line commit messages agrees with the finding of Buse and Weimer's study [5]. One thing to note is the number of commit messages of over 10 lines, particularly in the projects Drjava and Jfreechart. Qualitative analysis of these longer commit messages showed that they were long and detailed developer-written commit messages, typically between 10 to 20 lines. In some cases this output exceeded 100 lines, but these constituted a very small portion of the commits. In these cases, the developers had usually copied a list of additional information into the commit message. For example, Drjava had a few extremely long commit messages caused by developers appending lists of changed files

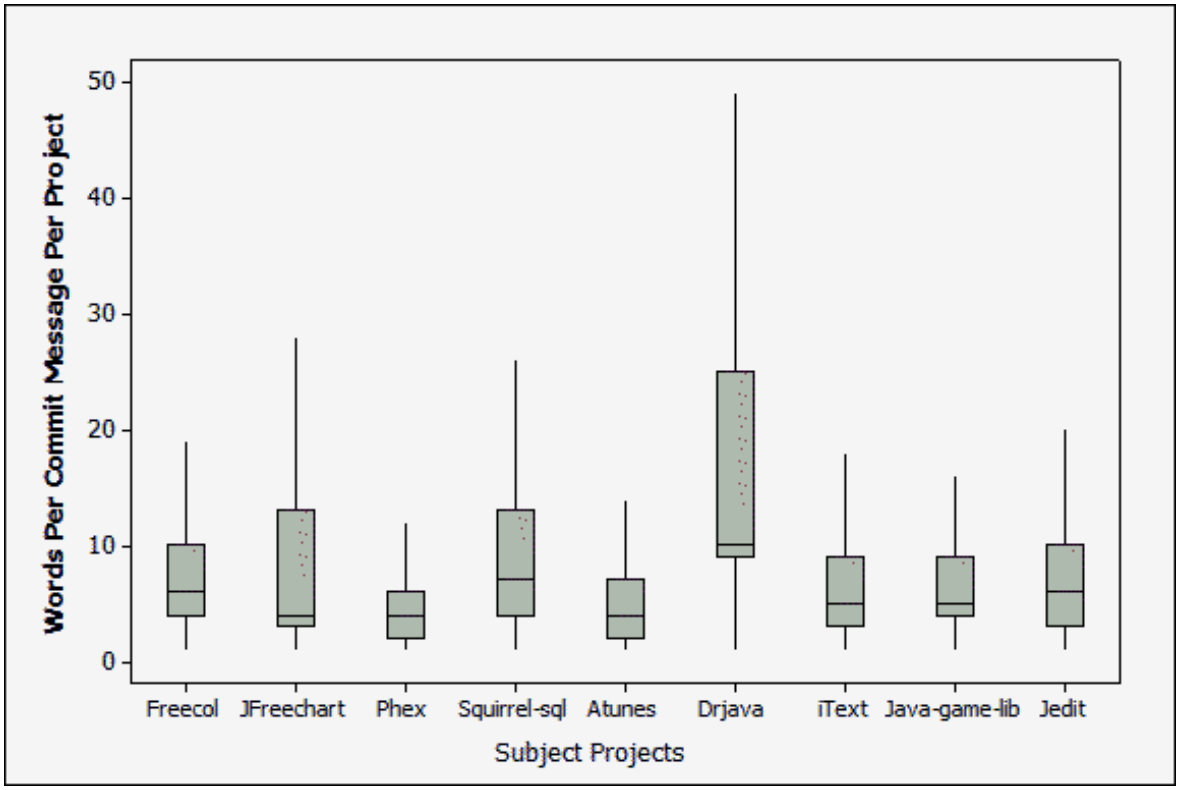

Figure 4.2: Boxplot of the Distribution of the Number of Words in Commit Messages in Each Project

or debug output. In JFreechart, occasionally long messages were the result of one developer's documentation style, which listed the entire directory path of each change on a new line. These messages seem to be produced under what might be an automated template, where the developer fills in the comments on each change. A short example is listed below:

- * source/org/jfree/chart/entity/AxisLabelEntity.java
  (AxisLabelEntity): Assign axis argument,
  (toString): New method override.

However, since commit messages are often only a few lines, and since we are looking to mimic them with natural language output, we also measured message size in terms of words. We determine words solely by spaces, so file paths like "source/org/jfree /chart/entity/AxisLabelEntity.java" or method names like "AxisLabelEntity" in the

example above would both count as one word. Figure 4.2 shows the distribution of the number of words in each of the projects with the outliers (very large commit messages) removed. The median number of words in a commit messages ranged from 4 to 10 depending on the project, and in all projects the commit messages were less than 25 words at least 75% of the time. Therefore, we believe that a good target for the median length of generated natural language commit messages would be no more than 25 words.

Finally, it is notable that we found that virtually no log messages in these repositories were left blank. In some cases, the commit messages had very poor content, with only a few non-descriptive words, but they were not left blank. This agrees with the findings of Buse's study, which found that over 99% of commits were not empty [5].

4.3.3 Scope of Commits

There is no standard definition of what is granularity of change is too small or too large for a commit. Therefore, we measured the sizes of commits in terms of their Java source code files. We classify these changes into 3 categories: number of modified, added, and removed files. File differencing methods exist for modified files, but established differencing methods may not be suitable for describing wholly added or removed files. If these appear frequently, different techniques would need to be used to describe the changes.

In Table 4.3, we breakdown how frequently we see each type of change in commits. The percentages in the table are not exclusive; they document the percentage of commits with at least one modification, addition, or removal out of the total set examined. First, in every project, the most significant portion of change came from modifications to the Java source code files. Also note that in all projects, new file additions outpace the number of file removals. This makes intuitive sense - the source code grows as the the project matures. Moreover, source code modifications occur in at least half of all commit messages across all projects. Since the amount of source

Table 4.3: Percent of Commits Containing a Java Source Code Modification, Addition, or Removal

Project	Modified Files	Added Files	Removed Files
Atunes	72.49%	12.04 %	5.14 %
DrJava	53.38%	39.49 %	2.2 %
Freecol	80.76%	6.72 %	1.01 %
iText	65.65%	10.98 %	2.43 %
Java-game-lib	50.96%	15.56 %	1.97%
Jedit	59.96%	9.3 %	2.38 %
JFreeChart	91.75%	6.13 %	1.67 %
Phex	76.48%	14.52 %	4.57 %
Squirel-sql	56.04%	15.22 %	1.97 %

code file additions and removals are typically much smaller than the amount of modifications, any initial techniques to produce commit messages should focus primarily on the modifications to existing source code files. Finally, in some cases there were no Java files added, removed, or modified in a commit. We did not document how often this case occurs, but these commits include changes to documentation, libraries, and code changes to non-Java files.

Figure 4.3 summarizes the number of changes to already existing Java files, with very large changes removed for clarity. All distributions had large numbers of outliers, indicating rare occurrences when a large number of files in the repository were edited. Since these large changes are very infrequent and would be very difficult to summarize concisely and accurately in a few sentences, they should not be the first focus for an automated tool. Due to the number of large commits, we use the robust measure of the median and quartiles to determine the scope of a typical commit message. Using the interquartile range across all projects, we estimate that a commit will typically result in a modification of between 1 to 4 java source code files. Studies performed by other researchers also found that most commits do not affect a large numbers of files [5, 13]. Buse and Wiemer calculated the size of the diff results for a commit and found an average of 37.8 lines [5]. Since we did not measure the number of lines changed per

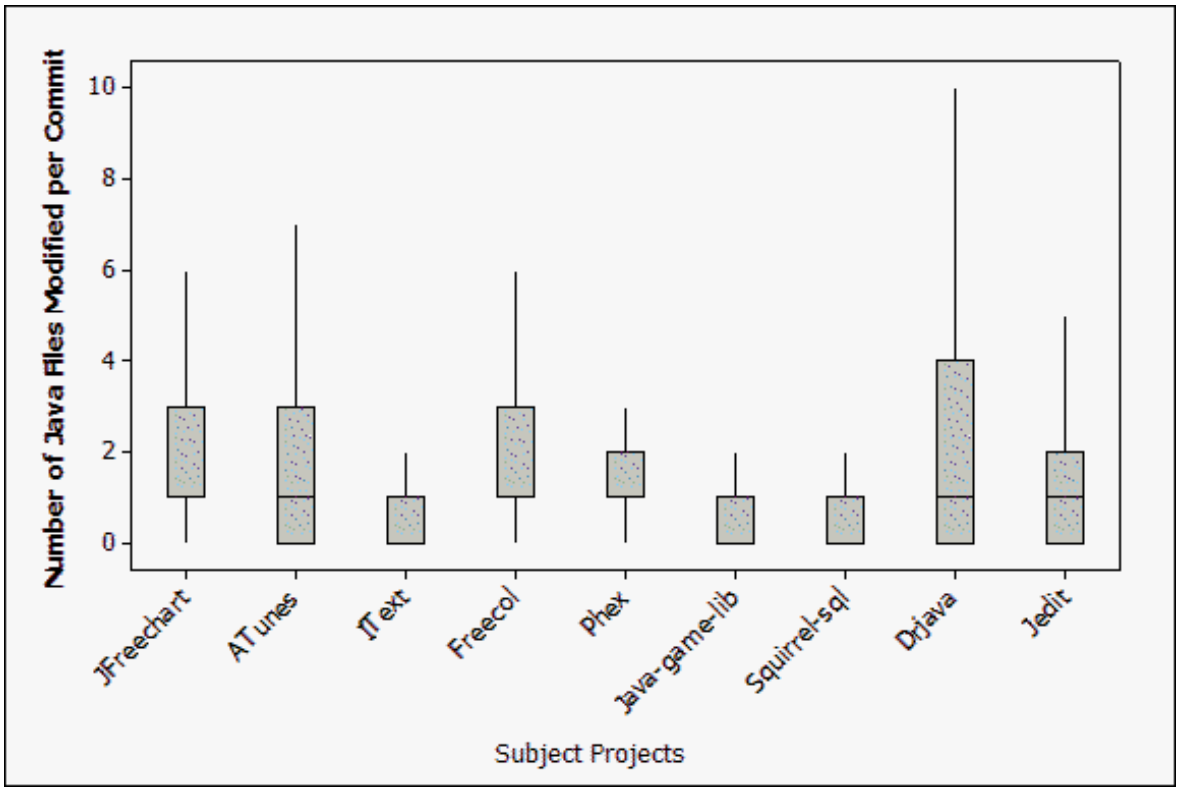

Figure 4.3: Distribution of Java Source Code Files Modified per Commit

file, we cannot verify the amount of change per file. It is possible that large changes were being made to only a few files. But, at least from the perspective of the number of changed source code files, the amount of change is not typically large.

4.4 Linguistic analysis of Commit Messages

We performed some basic linguistic analysis on the messages by extracting parts of speech and word phrases from the commit messages. In order to generate natural language output, we must first have an understanding for the linguistic structure of developer-written commit messages. In order to examine the linguistic structure, we used the OpenNLP [11] natural language processing library for Java to tag and chunk the commit messages. In a natural language context, tagging it involves the labeling of words with an associated part of speech. OpenNLP uses the Penn Treebank tag definitions [26] for its part of speech tagging. The second stage, chunking, uses the tagged words to identify larger constructions in a sentence, such as noun, verb, and prepositional phrases.

4.4.1 Linguistic Properties of Commit Messages

Due to the tedious nature of the work, we picked only a subset of 4 projects (Phex, JFreechart, iText, and Freecol) in our initial examination of the linguistic properties of commits messages. In each of these projects, we examined two things. First, we tracked what part of speech the commit messages began with. Second, we measured the composition of the whole commit message in terms of word phrases (chunks). While commit messages sometimes follow the format of English sentences, they also are prone to be filled with phrases instead of complete sentences, and parsing the messages for part of speech information gives a better sense of the structure of the commit message. Charts for the distributions of the parts of speech and phrases can be seen in the figures below.

First, we were concerned with only the part of speech of the first word in the commit. Examining the distributions by frequency, we can see that as a beginning

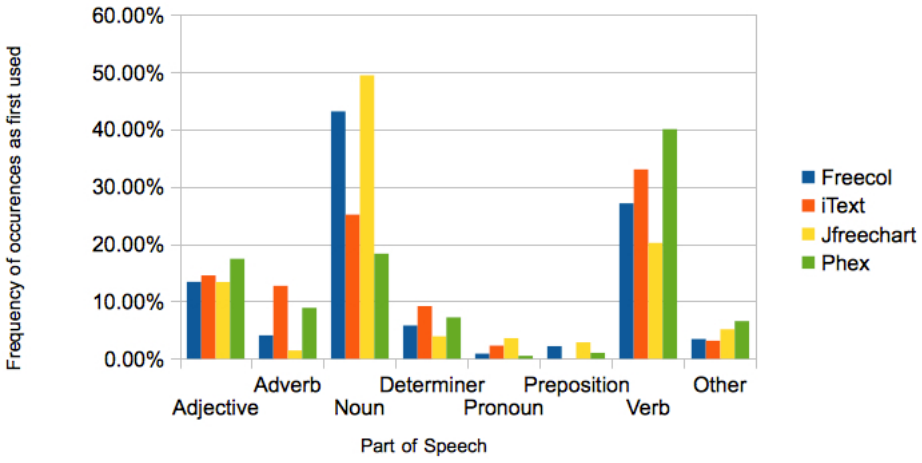

Figure 4.4: Frequency of each part of speech in first word position of commit message. Note that these parts of speech are a subset of the full set of Penn Treebank tags.

word choice of a commit, verbs and nouns are by far the most popular choice, although which of these is used more commonly depends on the project. After these, adjectives are the second most popular choice, which indicates that the starting word phrase of the commit is likely a noun phrase. Similarly, determiners (”a”, ”the”, ”this”, ”few”, etc.) modify nouns and indicate that the beginning of the commit message is a noun phrase. Adverbs also appear as a first word a notable number of times, which would indicate a verb phrase at the start of the commit. Taking these modifiers into account, it seems that noun phrases are a slightly more common opening section to a commit message than verb phrases. Within the distributions of the noun and verb phrases, there is a large degree of variance from project to project, with two exceptions. For nouns, singular nouns consistently were the most common starting word, as opposed to plural nouns. Likewise, past participles appear often in the verbs, as they capture the past tense terms many commit messages begin with, such as ”Fixed”, ”Added”, ”Changed”.

After examining the distribution of part of speech frequencies for each first word,

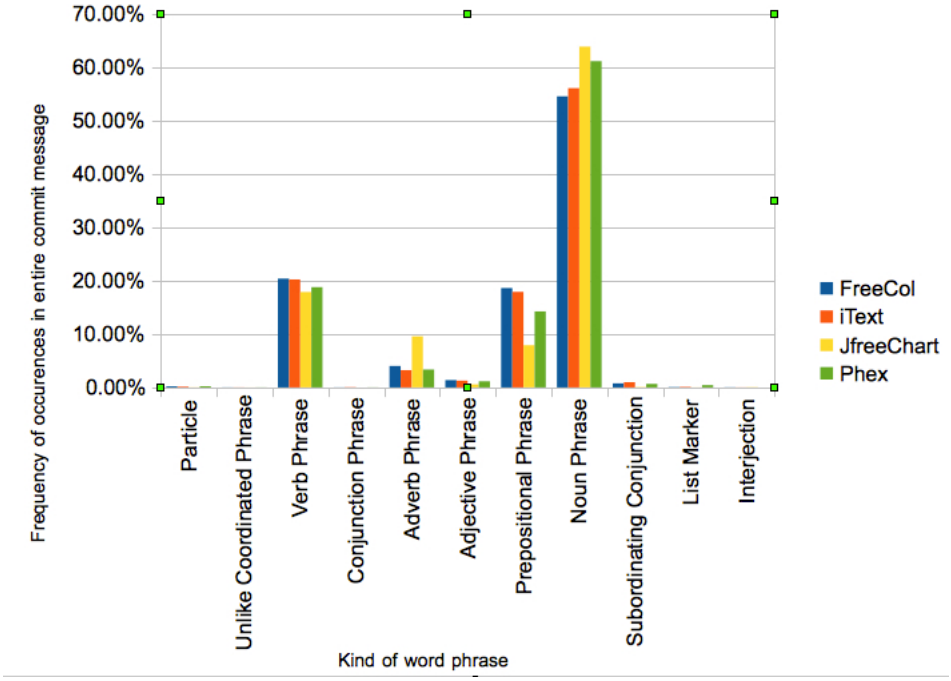

Figure 4.5: Distribution of Word Phrases in Open Source Projects

we also examined the distribution of word chunks across the entirety of the messages. Among the listed chunk tags, three types of phrases stand out: noun phrases, verb phrases, and prepositional phrases. Figure 4.5 presents the distributions of the word phrases across the commits in each of the examined projects. Noun phrases dominate the content of the commit messages, with verb phrases in a distant second by about 3 to 1. While it was expected that nouns and verbs would dominate the commit messages, this asymmetry suggests that more nouns phrases than verb phrases would be needed in a generated commit message. Perhaps the action of the commit can be summarized in a few verbs, but describing all affected objects, methods, and files would take several nouns. Finally, while prepositional phrases comprise a significant number of the word phrases in developer written commit messages, they are not part of the verb-direct output model that has been used for search and comment generation in some techniques [27, 29]. The information that prepositional phrases convey in commit messages will be explored further in the next chapter along with an evaluation of the verb-direct object technique as a way of extracting summaries from commit messages.

4.4.2 Extracting Verb Direct Object Summaries

As a first step toward generated commit message content, we extracted verbs and their matching direct objects automatically from commit messages to examine how well that information could capture the whole commit message. The method of extraction is described in detail below, and the evaluation of this output as a summary for commit messages and as a potential target for automated output is presented in Chapter 5.

Step 1. In order to tag and chunk the commit messages with the OpenNLP tagger [11], they must first be modified slightly to have the correct format for input to OpenNLP. The openNLP library is designed to handle English sentences, and in many cases the commit messages have poor grammar or do not otherwise follow the patterns of conventional English. One important first step is that certain kinds of punctuation must be removed because the tagger was observed to be performing poorly in some

cases. In preprocessing scripts before the openNLP tagger, contractions are expanded to avoid dealing with apostrophes. For example, can't becomes can not, won't becomes would not, etc. Additionally, file paths in commit messages have all occurrences of the '/' character removed so that they can be treated as a single word. Furthermore, file names are altered to contain only text in the following manner: if the file name is example.java, it becomes examplexjava for the tagger. Prior to making this modification, the openNLP tagger would assume that the '.' marked the beginning of a new sentence. This issue also arose with version numbers, which contain periods (e.g. 1.5.4). These numbers are specially encoded so that they will remain intact for the tagger, and they are reconstructed after passing through the tagger. The correctness of each modification was verified by closely examining the output files and checking the output from many different examples.

Step 2. The output files from Step 1 are then processed by the openNLP tagger and chunker. However, as the openNLP package expects to parse individual sentences, before the commits are tagged, they are divided up using the remaining punctuation as a guide. Then commit messages are run through the tagging software. The tagger is able to produce multiple part of speech taggings for each word, and ranks each tagging by probability of correctness. Manual accuracy studies were done on these parts of speech taggings to determine if they were correct. By examining 100 first words from the part of speech taggings, it was found that the tagger correctly identified the part of speech 87 times within the first three taggings. Therefore, each word was given three tags, which represent the three most likely part of speeches for each word in the context of that sentence. Next, the chunker was called for each commit message three times. The most probable chunking generated by openNLP for each tagging is stored, giving a total of three chunkings.

Below is an example of the output of the openNLP tagging and chunking for a commit message, divided up into each of its constituent words. The definitions of all tags used in this example are described in Table 4.4:

Table 4.4: Subset of Penn Treebank Word Chunks Tags Used in Example

Word Tag	Meaning
VBG	Gerund
VB	Verb, Base form
NN	Singular Noun
NNP	Singular Proper Noun
TO	Special tag for 'to'
Chunk Tag	**Meaning**
B-VP	Beginning of new verb phrase
I-VP	Continuing verb phrase from last chunk
B-NP	Beginning of new noun phrase
I-NP	Continuing noun phrase from last chunk

- Refactoring (B-VP, B-NP, B-NP, VBG, NN, NNP)
- method (B-NP, I-NP, I-NP, NN, NN, NN)
- extraction (I-NP, I-NP, I-NP, NN, NN, NN)
- to (B-VP, B-VP, B-VP, TO, TO, TO)
- improve ([I-VP, I-VP, I-VP, VB, VB, VB)
- code (B-NP, B-NP, B-NP, NN, NN, NN)

Each word has six tags associated with it. Consider the final three tags of each word first. These are the part of speech tags from the Penn Treebank set [26] associated with the word in decreasing order of likelihood. So for the word 'refactoring', it considers the most likely tagging to be 'VBG', or a gerund. Afterwords, it considers 'NN' and 'NNP', both of which are noun taggings. Now consider the first three tags. These are the chunk tags for each associated with each part of speech. They are divided into two parts, separated by a dash. The first part is either an 'I' or 'B', which indicates either that the chunk is a continuation of a word phrase started in an earlier chunk or the beginning of a new word phrase. So for 'refactoring', it thinks this is most likely the beginning of a verb phrase (VP). Alternatively, the model suggests that this word

might be the beginning of the noun phrase 'Refactoring method extraction', indicated with 'B-NP'. Notice that this is continued in the second most likely chunk tags for 'method' and 'extraction' with the phrase 'I-NP', which means the noun phrase is continued from the previous word. However, OpenNLP has selected that the most likely breakdown of this commit message is the following:

- Refactoring (Verb Phrase)
- method extraction (Noun phrase)
- to improve (Verb phrase)
- code (Noun phrase)

Step 3. After tagging the messages once, we modified the commit messages slightly to further improve the accuracy of the tagger and chunker. We examined the word level tags of the first word of each tagged message. Words tagged as past tense verbs and as past participles have the word 'I' appended to the front of the commit message and those messages with gerunds as their first tag have the words 'I am' appended to them. For example, we changed the commit message "finishing up configuration" to "I am finishing up configuration" and "fixed false jarfile bug" to "I fixed false jar file bug". Since commit messages sometimes exhibit poor grammar, we found this change often made messages more closely resemble full English sentences, which is the expected input for openNLP. This technique of adding a subject is similar to what Abebe and Tonelle use to improve the parsing of sentences created from words extracted from method names [1]. This modification is made whenever at least one of the three part of speech tags assigned to the first word was either in the past tense or a past participle, as we had noticed that OpenNLP had been tagging words incorrectly in this instance.

Step 4. The messages are run through the openNLP tagger and chunker a second time with the slight modifications.

Step 5. In this step, we summarize the commit messages and then output them next to the original message. In order to create a summary from the tagged messages, we

examine all three chunkings of every commit message. Recall that each word in the commit message is given three part of speech tags in order of decreasing probability. For each set of part of speech tags (1st most likely, 2nd most likely, 3rd most likely), the most likely chunking is used. This results in 3 separate chunkings, each based on a different set of part of speech tags. For each message, we decode the encoded version numbers and apply a conservative correction to the chunking. By qualitatively observing the output we found a pattern of where a word tagged as a verb was incorrectly labelled as part of a noun phrase in the chunk tag. To correct this mislabeling, for each noun phrase chunk, we examine each tagged word in the chunk. If a word in the chunk is tagged as VB or verb base in all three tags, the noun phrase chunk is divided into three pieces: anything before and after the verb becomes a new noun phrase, and the verb itself becomes a verb phrase. For example, this sentence in a commit message: "Let ListOption wrap options rather than raw values, making the UI more generic." After completing the tagging and chunking, openNLP had labeled "wrap" as a verb base in all three tags. However, in the chunking, it mistakenly described this word as part of a noun phrase: "ListOption wrap options". The change described above divides this into three parts, first a noun phrase, then a verb phrase, and then another noun phrase, which is the correct way to divide this phrase in the context of the sentence.

Finally, the rest of the commit message is examined for verb phrases. We start searching for verb phrases with the most highly rated chunking, and if nothing is found, we check the second and then the third rated chunkings. By using all three chunkings returned from openNLP, we increase our ability to extract verb phrases out of ambiguous commit messages. Depending on whether or not one is found, we do one of two things:

Step 6. If the chunker was not able to find a verb phrase, the commit message is put into file separate from those messages where we found verb phrases. Commit messages in this file can be one of two types. The first possibility is that the message does not contain any verb phrases. The second is that the tagger/chunker incorrectly

labelled the message, and the verb phrases have been tagged as another type of chunk. These messages are filtered for uniqueness and length, i.e., repeated messages are only printed once and messages with only one word are automatically discarded. We make the assumption that one-word commit messages are not effective summaries of changes in the code.

For messages where the tagger found at least one verb phrase, we apply our method of extracting the important actions of the commit by pairs of verb phrases with their associated direct objects. To do this, we attempt to identify the voice of each verb phrase. The difference between active and passive voice is that in active case the subject performs the action, and in the other an action is performed on the subject. The following commit message is an example that uses both the active and passive voice: "Fixed a bug where an extra list item was added if the list was in a table". The active voice verb 'Fixed' has 'bug' as its direct direct object, whereas the passive voice verb 'was added' refers to the 'extra list item' that comes right before it in the sentence.

The passive voice is most commonly associated with the past participle. If there is a past participle in the phrase, we search for a noun phrase before the verb, and label it the direct object. This search is halted when either a noun phrase is found, the start of the commit message is reached, or, in the case of multiple verb direct object pairs, when we reach a chunk previously used in another verb direct object pair. If no noun phrase is found in this case, the verb phrase is then treated as if it were in the active voice. In the case of active voice, we search for the first noun phrase after the verb phrase or until the next verb phrase is found. If no noun phrase is found, then only the verb phrase is reported. Once all verb phrases have been accounted for, we print them in the form <pair1 >- <pair2 >- ... - <pair n>for each commit message. Some examples of summaries with their associated commits messages are shown below in Table 4.5, and additional examples can be found in Table B.1 in Appendix B:

Table 4.6 summarizes the results of our extraction across all projects. The range of commits in which we identified verb phrases varied, from 64-65% in Atunes and Phex

Table 4.5: Examples of Commit Messages and their Extracted Phrases

Commit Message 1	Allow attack from ship?land if amphibiousMoves option is enabled, mitigating longstanding concerns regarding unconquerable islands.
Summary 1	Allow attack - amphibiousMoves option is enabled - mitigating longstanding concerns - regarding unconquerable islands -
Commit Message 2	Removed option to show ticks and labels in progress slider
Summary 2	Removed option - to show ticks and labels -
Commit Message 3	used new GUIActionPerformer for banning hosts, changed banning time from SESSION to one week new priority icons
Summary 3	used new GUIActionPerformer - changed banning time -

Table 4.6: Number of Identified Verb Phrases vs Uncategorized Commits

Project	Commits Categorized				Total Number of Commits
	Found Verb Phrase		Uncategorized		
	#	%	#	%	
Atunes	3243	64.18	788	15.59	5053
DrJava	5267	96.55	168	3.08	5455
Freecol	6253	78.43	1452	18.21	7973
iText	3407	68.41	996	20.00	4980
Java-game-lib	3067	82.07	532	14.24	3737
Jedit	16797	82.21	2182	10.68	20432
JFreeChart	1780	72.36	100	4.07	2460
Phex	2310	64.94	563	15.83	3557
Squirel-sql	5195	78.94	804	15.59	6581

Note: Since the Uncategorized Commits do not include single word commits or repeats, the percentages do not sum to 100% and the raw totals do not sum to the total commits column.

commits to 96.55% in the DrJava commits. Overall, 78.6% of commits were identified as having verb phrases. Therefore, it seems likely that verb phrases are necessary for good commit messages, but we will explore this further in Chapter 5.

4.5 Summary

We have examined a large subset of commits and commit messages over a range of popular open source Java projects. We found that although the distribution commits among developers varied, the majority of commits in open source projects are focused around a relatively small group of developers, with a larger group making small contributions. Both commits and commit messages tend to be relatively small in scope, with commits usually only modifying a few files, and commit messages typically not exceeding more than a few lines or more than 25 words. Finally, we analyzed the linguistic structure of commit messages and found that they typically begin with noun or verb phrases. Across the whole commit message, noun phrases dominate, although both verb and prepositional phrases occur regularly. We found that nearly 4 out of 5 commit messages have at least one verb phrase, indicating that they are likely important in developer-written commit messages. Finally, while we account for the role of noun and verb phrases in commits in our model, we did not account for the role of prepositional phrases, which will be examined more in Chapter 5.

Chapter 5

HUMAN OPINION SURVEY ON COMMIT MESSAGES

In order to design algorithms for generating commit messages, we need a clear view of what information is written in commit messages, when people read in commit messages, and what information is wanted in commit messages from others working on the project. Therefore, we created an online survey to ask users of software repositories both general questions about commit usage and about the usefulness of the extracted summaries described in Chapter 4. Since noun and verb phrases have clear mappings to objects and methods, respectively, in object-oriented programs, we wanted to know if the verb-direct object summarization technique would be sufficient as output for automatically generated commit messages, or if not, what information should be targeted in creating a commit message. The research questions we wish to answer and the design of the survey are explained in the following section, and the full survey can be viewed in appendix A.

5.1 Research Goals

A commit message can be viewed from two perspectives: the person writing the commit message and the person reading the commit message. Our survey's primary goal is to answer questions about what readers of commit message desire, although a few questions do address the writer's perspective. If there is a discrepancy between what developers want in commit messages and what they actually put in them, this would be a great target for a automated tool generating output that complements existing messages. Our research questions are:

- Do developers read other's commit messages?

- When developers read them, why do developers read them?
- Under what situations do developers read them?
- What kinds of information do they find most useful in commit message?
- Do useful commit messages have a similar linguistic form or similar set of linguistic forms? What about useless commit messages?
- Can a simple verb phrase capture the main action of the commit message? Are commit messages that don't contain verb phrases useful and if so, why?
- Is the technique of providing a summary with verb phrases and direct objects sufficient?
- Does certain kinds of information in a commit message typically appear in different parts of speech?
- What kinds of information do writers say they put in commit messages?
- How does what developers say they put into commit messages compare with what they want from commit messages?

To address these questions, the survey was divided into three sections, a section on general usage of commit messages, a section addressing the commit messages that OpenNLP successfully extracted a verb phrase from, and a section addressing the commit messages for which OpenNLP did not identify a verb phrase in.

5.1.1 General Usage of Commit Messages

First, respondents were asked questions about their current positions and general experience with programming. In this section, respondents were asked about how frequently they used commit messages and for what purposes they used them. Additionally, we asked questions about what kinds of information they put into commit messages, and what information they like to see in commit messages they read.

5.1.2 Commit Messages and Their Extracted Summaries

The second section of the survey presented respondents with commit messages that our tagger was able to extract verb phrases from. On the survey, we referred to commit messages of this type as group 1, and will continue using that naming method

here. This section asked questions about the effectiveness of the extracted verb phrases and direct objects as a summary of the original commit message. Summaries were only presented for commit messages that the evaluators deemed useful, and in the cases where they were not, we asked for an explanation of why the original commit message was not considered useful. In cases where we did present the summary, questions were asked about the overall effectiveness of the summary, including if it was misleading, or had too much or too little information in the summary.

5.1.3 Uncategorized Commit Messages

In the third section, we presented the commit messages in which our tagger was not able to find a verb phrase. These commit messages are considered to be in group 2. Recall that this case occurs for one of two reasons, either the tagger mislabeled the commit, and there actually is a verb phrase, or the commit message did not contain a verb phrase. Regardless, we asked them to classify the commit message as useful or not useful, and in cases where the commit message was both useful and did not contain a verb phrase, we asked them to explain why the message was useful without a verb.

5.2 Survey Design

The survey was designed to be easy for respondents to understand and use and to avoid biasing their responses. The questions came in three types - text responses, multiple choice/check boxes, and scaled questions from one to five. Since they cause the greatest inconvenience for survey takers, the number of text response questions was limited, and typically they were used in situations where the respondents could optionally provide additional information. Questions where we wanted either a positive or negative response were posted on a range to give a more nuanced view of the evaluators' opinions. Finally, questions about the usage of commit messages had preloaded answers from which respondents could select any number of responses. All evaluators who took the survey received an account, so that they could stop at any time and return later to finish it at their convenience. Although the survey was designed to take

about 20 to 30 minutes to complete, since the judgements on the commit messages and their summaries could get tedious, the questions about the general usage of commit messages was placed first so respondents would at the very least complete this section. This also served the purpose of getting them to think about commit messages and their usage before making judgements on the quality of actual commit messages and the extracted summaries.

The commit messages used in this survey were randomly selected from the software repositories of the 9 Java projects discussed in the previous section. In total, there were 10 commit messages and summaries presented to each evaluator from Group 1, and 10 commit messages from Group 2. For each commit message in the two groups, the message was presented to 3 different respondents. By having multiple evaluators for each commit message, we strengthen the judgements on the quality of the commit messages and extracted results as it allows us to obtain majority opinions for each of the commit messages.

5.3 Refining and Testing the survey

After building an initial framework for the survey in 2012, we went through a process of improving the content, scope, and clarity of the survey before properly launching it. After a draft of the online survey and database had been tested for correctness, a student from the University of Delaware was asked to complete the survey and provide commentary on the clarity of the presentation and mention of any bias or misleading terms in the questions. Additionally, the survey content was examined by Jeff Carver, an empirically-based software engineering researcher at the University of Alabama, who offered feedback on the style and content of the questions that were being asked. Finally, in the week before the survey was launched, several more students were asked to complete the survey in a trial run and their answers were monitored to ensure that they were understanding the questions correctly. Once this was completed, the survey was released to the rest of the correspondents.

5.4 Targeted Demographics

The survey was targeted at people who had experience working with software repositories and software projects in a team environment. Therefore, we targeted both software developers as well as undergraduate and graduate students at the University of Delaware with experience working with software repositories. Jeff Carver assisted us in getting into contact with some software developers. The students chosen had completed at least one course in software engineering. These courses simulate working in software development teams, requiring students to work in groups to create a substantial software project. In each of these projects, the students had to use a software repository to manage their projects, which included the use of commits and commit messages as a form of documentation. The list of students asked for this survey was obtained with the assistance of Professor Terry Harvey.

5.5 Results

After running our survey for approximately two weeks from the end of January to early February 2013, we received a total of 26 responses, with 18 individuals completing the whole survey. Every respondent completed the first section of the survey, but there were varying degrees of completion of the last two sections. In total, respondents looked at 90 commit messages in group 1, and 90 commit messages in group 2. Respondents were asked to state their current position(s), and since their was some overlap, 18 identified as undergraduates, 3 as graduate students, 8 as being in the workforce as developers/software engineers, a 1 computer science graduate not currently working as a developer. Of these correspondents, 16 listed their experience as less than 5 years, 9 as having 5 to 10 years, and 1 with 10 to 15 years experience as a developer.

5.5.1 Purposes for Reading and Writing Commit Messages

First and foremost is the question: How relevant are commit messages as a form of documentation? If commit messages are not used by developers, then it would not be useful to attempt to automatically generate them. These results of this question

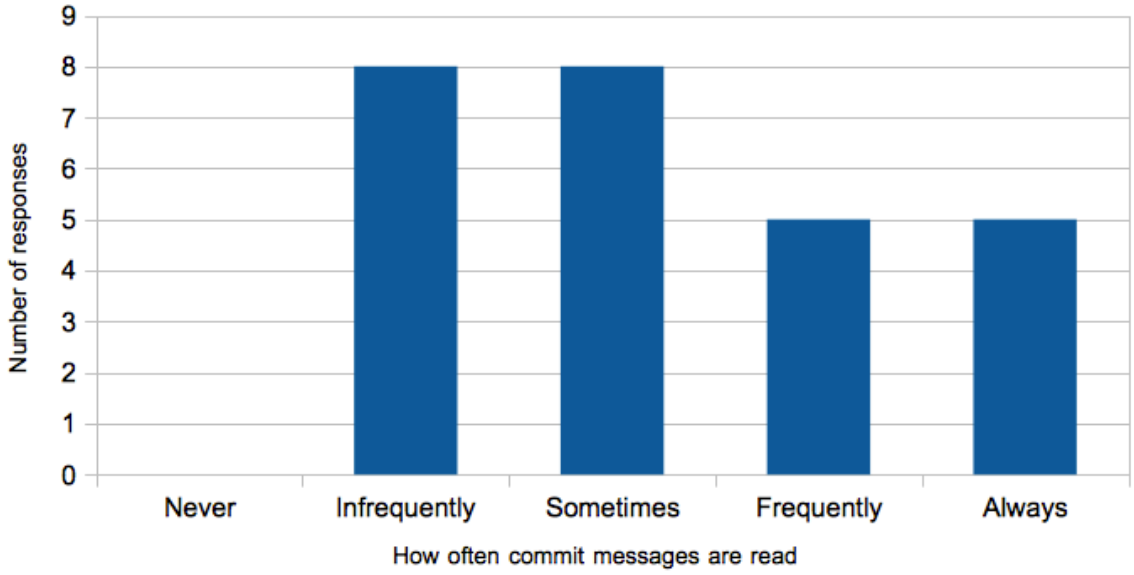

Figure 5.1: Frequency of Reading Commit Messages

are presented in Figure 5.1. Notably, all respondents to the survey answered that they at least occasionally read commit messages. The distribution of answers skews toward reading commit messages infrequently, but a significant component read commit messages always or at least frequently.

Survey respondents said that they most often read commit messages to get the intent of the changes, while seeking to understand the commit's purpose in the larger context of the program is an important, if secondary goal. The results of the survey suggested error tracking is also a use for commit messages, with 65.4% of respondents saying they read commit messages in this scenario. Here are some survey comments to that affect:

- "Usually, I read commit messages and use vimdiff to isolate differences in code such as bugs. I usually want to isolate some module that didn't work properly and go back to a version before it was implemented for comparison."
- "If after updating, function A no longer works, I want the commit messages to tell me what functions they edited so I can find the commit that broke function A."

While all but one of the respondents thought a short summary of the changes made were important, only 42.3% thought commit messages should be explicitly related to a project goal, and 26.9% thought that important to include information on how the changes were implemented. Therefore, any method for summarizing code changes to create a commit message should avoid outputting too low level of an explanation. Additionally, it seems that better commit messages avoid putting in too much higher level content, such as relating the change to project goals. For example, one respondent said the following, suggesting that such information is better suited to other forms of documentation: "When working on a team, goals should have already been communicated and I don't like reading an essay on what I already know. I much rather prefer a short but thorough description on what was changed, and if I don't understand a change I investigate it further, and as a last resort I contact the person directly."

The information respondents reported including in their personal commit messages matches what they want while reading them fairly closely. 92% of respondents identified explaining what changed in the commit as an important goal when writing messages, as opposed to 34% who said explaining the implementation details as important. Regarding this, one respondent said, "Specific support would be in code comments", suggesting that the comments are the best place to explain implementation details. Approximately 58% said that they link the changes to project goals, which is slightly higher than the number who found it useful while reading. Finally, one comment mentioned, "I also put in line numbers and file names when available." While this would be fairly easy to implement, these features are already available in some repository implementations as additional information to the actual message.

5.5.2 Useful and Non-Useful Commit Messages

In order to determine whether a commit message is useful or not, we use a majority opinion of the evaluators who looked at each message. As the usefulness of a commit message is ranked from 1 to 5, for a single evaluator, a rank of 1 or 2 is

considered useless, while 3 or above is useful. If at least 2 evaluators ranked a message as useful or useless, then it is considered as such, regardless of whether the message was evaluated by 2 or all 3 people. Messages that received only one response, or had only two evaluators who disagreed are considered as indeterminate and are not considered in the counting.

Applying this metric, there were 115 commit messages with 2 or more evaluators in agreement about the usefulness of the commit message out of a total of 180 commit messages presented. Of these, evaluators stated that 78 were useful and 35 were useless across both groups, giving a measure of 67.8% useful and 32.2% useless commit messages across all the open source projects. However, there was significant variance in the opinion of the usefulness of evaluated commit messages among respondents. Figure 5.2 shows the differences between the highest and lowest opinions of all 124 commit messages with at least 2 evaluations. While complete disagreement about the usefulness of a commit message was uncommon, in the most common case the evaluators had moderate disagreement on how useful the commit message was - a difference of 2 on a 5 point scale. Therefore, while our results suggest that at best about 2/3 of open source commit messages are useful, it is difficult to be conclusive as a different set of evaluators may have given different results. We are not sure what factors may contribute to having a stricter standard of a useful commit message.

The data on useful and non-useful commits were also manually categorized into groups to determine the overall purpose of the commit message and to see if there was any correlation between the content of the message and its usefulness. These categories were error fixes, refactoring, feature additional, feature removal, general modification, updates to supporting materials (anything other than source code), and indeterminate for commit messages whose category could not be identified. However, there was no obvious correlation between category and usefulness. In the useless commit message category, the most common categorization was indeterminate. These commit messages were too vague to determine even a general idea of the commit's purpose, including for example, "Next version", "Small changes", etc. Otherwise, bug fix commits were most

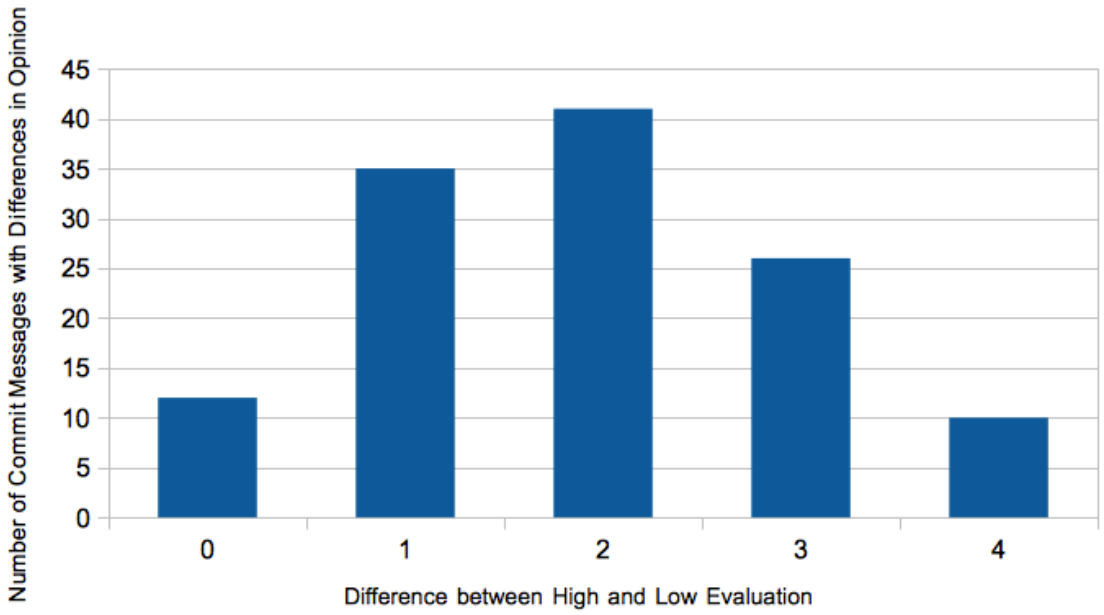

Figure 5.2: Variation in Opinion on Commit Message Usefulness

common in the useful commits and second most common in the useless commits, but no trend was significant enough to draw a positive conclusion.

5.5.3 Appropriateness of Verb-Direct Object Summarization for Commit Messages

The histograms in Figures 5.3, 5.4, 5.5, 5.5 detail the average responses to the four questions we asked correspondents about how well the technique of verb-direct object summarization maintains the meaning of the original commit message. The graphs measure the average opinion for these questions on each pair of commit message and extracted summary from the set of respondents who rated the original commit message in that pair useful.

The evaluators comments and ratings on the effectiveness of the verb-direct object summarization technique suggest that it is not appropriate as a summarization technique for commit messages. The evaluators gave the overall technique only slightly above 3 on average, meaning that it only somewhat maintained the information in

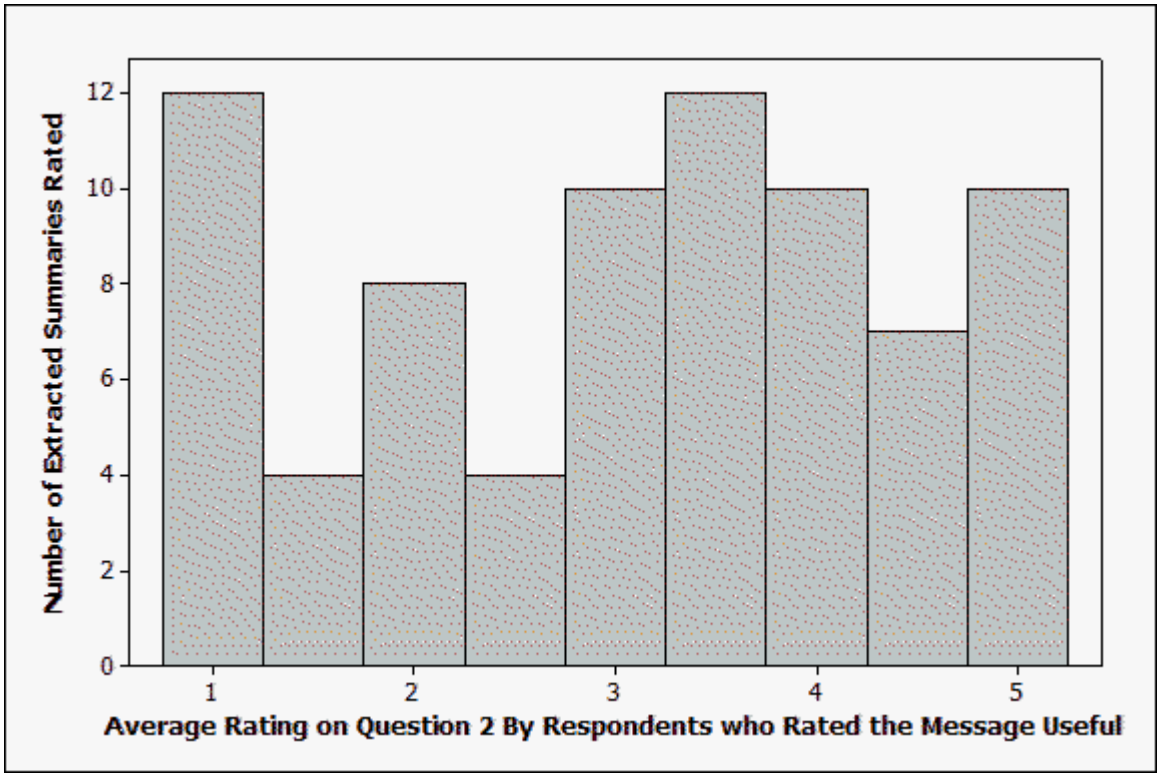

Figure 5.3: Distribution of average opinion among evaluators for each commit message to the question: "Overall, do the extracted clauses convey the main action(s) portrayed by the commit message?". The responses range from 1 - "Not at all" to 5 - "Completely".

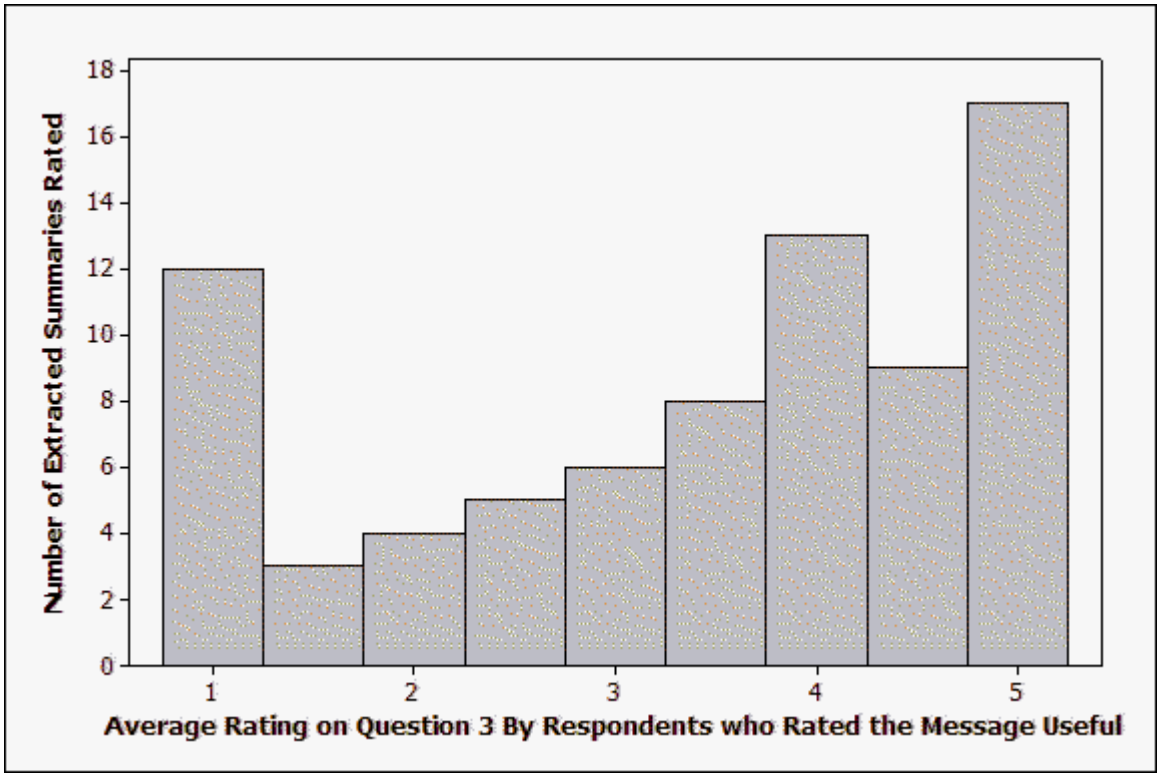

Figure 5.4: Distribution of average opinion among evaluators for each commit message to the question: "Does the summary created by the extracted clauses contain misleading information?". The responses range from 1 - "Very Misleading" to 5 - "Not Misleading".

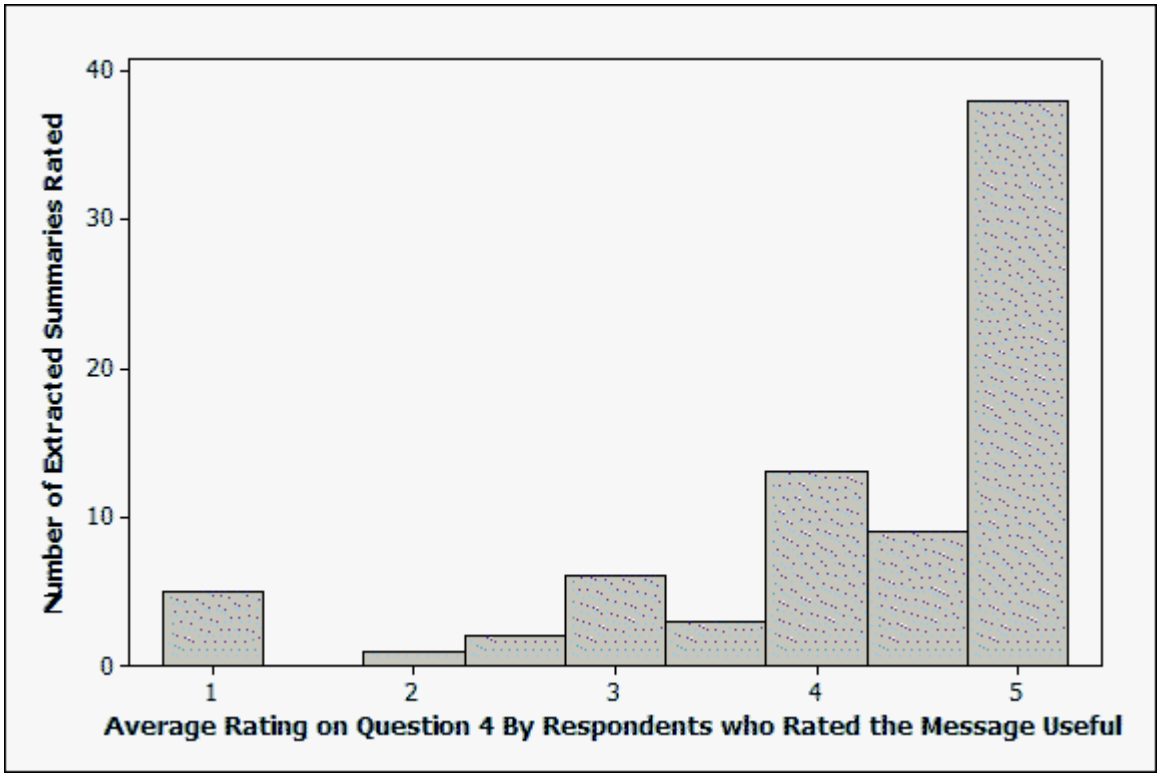

Figure 5.5: Distribution of average opinion among evaluators for each commit message to the question: "Do the extracted clauses contain unnecessary information?". The responses range from 1- "A lot of unnecessary information" to 5- "No unnecessary information"

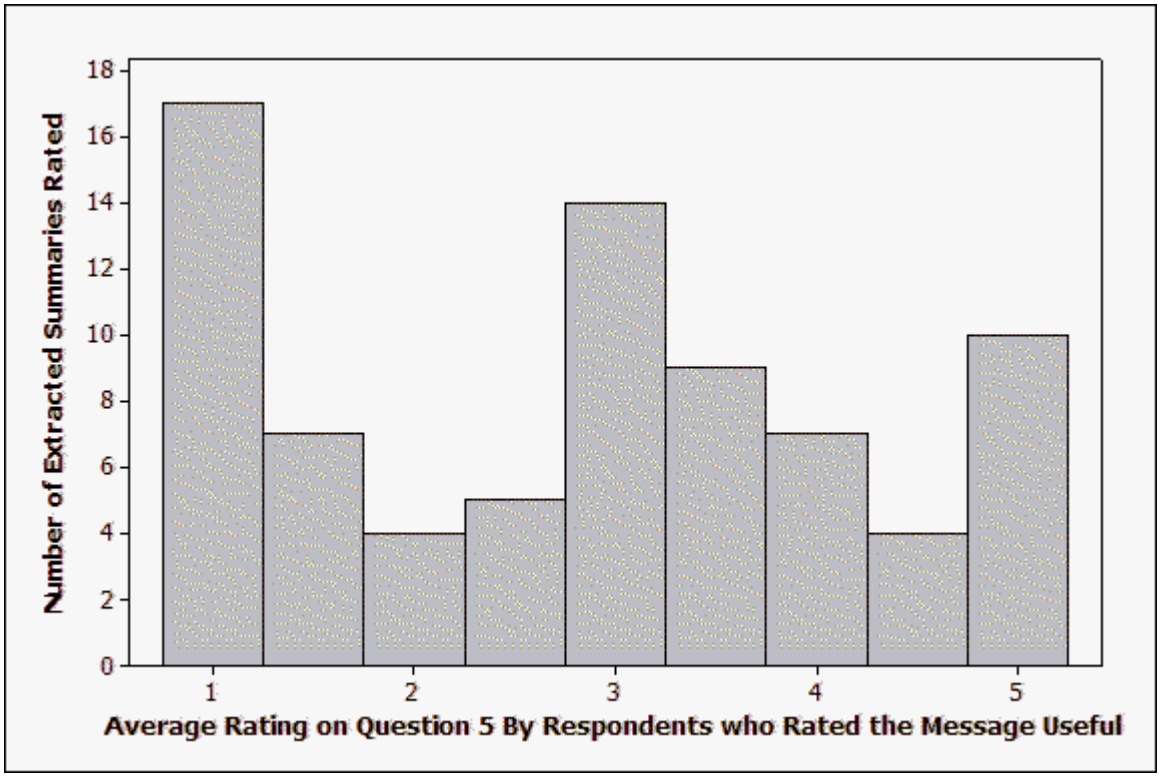

Figure 5.6: Distribution of average opinion among evaluators for each commit message to the question: "Does the extracted summary leave out any important information?". The responses range from 1- "Missing a lot" to 5 - "No information missing"

the original commit message. Additionally, on average, respondents believed that the summaries were somewhat misleading, and in some cases, as shown in Figure 5.4, it did very poorly.

The problem was not that the extracted phrases contained too much information, but instead that they were lacking information. Figure 5.6 shows their ratings on how much information each commit message left out. Disregarding examples where the technique created a distorted summary due to incorrect tagging, the extracted phrases usually missed out on identifying where a change took place. This information is not contained in the verb or its direct object. One of the most clear examples of this is the commit message: "Added minimum size to EditPreferencesDialog" and the extracted phrase: "Added minimum size - ". Here, the change is mentioned, but there is a loss of context to where the change is occurring. Also, the location mentioned may not be a file or method, but also could be a location within the application: "search button added to tool bar". We observed that this information is typically found in prepositional phrases, which were not included in the extraction technique's model. In English, prepositional phrases are used to modify nouns or verbs, restricting their scope. Although we must be cautious about our claims due to the small data set, it seems that these phrases are often used to identify where a change has taken place.

5.5.4 Usefulness of Commit Messages without Verb Phrases

Since the size of the data set here is very small, we can only make observations, but our respondents seem to suggest that a commit message must at least have an implied verb phrase in order to be useful. After separating out the group 2 commit messages that had been tagged improperly, among the commit messages with no verb phrase still identified as useful, we saw two cases. First, the verb was not explicitly mentioned, but it could be implicitly understood from the context of the commit message. In these cases, the implied verb was typically either some form of addition or removal, for example, "A new edit mode, minor abbrev tweak" or "No more JAXB". In other cases, we observed the verb was hidden in the form of a noun phrase. The

word 'cleanup' in this example message shows this behavior: "Misc minor build script cleanup". Finally, as one respondent mentioned, "If there is no action, then the commit message isn't useful... Then the code didn't change...". While it seems that commit messages without an explicit verb can sometimes be useful, for the purposes of an automated natural language tool, the most straightforward approach would be to keep the verb explicit.

5.6 Implications for Automated Tool Output

In Buse and Weimer's paper on DeltaDoc they describe commit messages as generally having one or two types of information, *what* and *why*. *What* information summarizes the actual changes made, and *why* information explains the rationale for making the change. They say that extracting high level *why* information from the code is often difficult, and instead focus on extracting *what* information [5]. Our survey results suggest that the information users of software repositories most want is in fact the *what* information, with *why* information being a useful, if secondary goal. However, the survey results also suggest that what is least desired is *how* information. This distinction is subtle, but *how* information details the implementation of the change, and is perhaps too low level to be appropriate in a commit message. Therefore, the concern with any tool generating commit messages from code is to avoid outputting information about how the code changed, and instead obtaining a greater level of abstraction provided by *what* information.

Therefore, combining these insights with the developer comments on our verb-direct object extraction technique, we can build a preliminary model of a natural language output for commit messages. There are six questions that are standardly asked about any situation: *who*, *what*, *when*, *where*, *why*, and *how*. In the case of a commit message, we can disregard *who* and *when* information as they are already part of the commit message's meta information. *Why* information is useful, but as a secondary goal. Since it is a difficult to produce, it is not appropriate to develop for an initial tool. *What* information should be the primary goal of the commit message, but

any automated tool generating commit messages should be careful to abstract the code changes enough not to produce *how* information instead. If the message becomes too descriptive in implementation of the change, it is not as useful to readers. Comparing to how actual commit messages are written, it seems that *what* information is summarized well by the verb-direct object model we examined in this chapter. Finally, the message should also include *where* change takes place. Fortunately, in simple cases, this can be added easily by including the name of the package, file, and method in which the change takes place. In more complex cases, location information may also be a feature or part of the running application, which falls under the field of feature location. Either way, when modeling human written commit messages, *where* information seems most appropriately represented in the form of a prepositional phrase. Finally, since commits sometimes represent more than one change to the repository, each of these changes should be represented with a different sentence. This will help readers distinguish individual changes from each other, and since the scope of change in commits are typically small, this method will not usually be too verbose.

5.7 Summary of Properties of Well-Written Commit Messages

Pulling from the results of the survey and manually examining the 180 evaluated commit messages, we can speculate on some properties that contribute to a good commits and commit messages. While the size of the commit might vary, it should be possible to describe the changes made in a few sentences. If the changes cannot be adequately summarized into a few discrete ideas to describe *what* changed, perhaps the commit is too large and should have been divided into smaller commits. It also seems important for readers of commit messages to be easily able to identify what type of change occurred in the commit, but using the type of change alone is likely insufficient. The general type of change described should be related to the specifics in the application. This includes information about the location of the changes, but also more details about the type of change than just a general category. For instance, writing that a bug was fixed does described *what* changed, but saying what kind of

bug it was and what areas of the program were affected seems to be more useful for other developers who may not be familiar with the affected code.

5.8 Limitations and Future Work

The validity of judgements about the usefulness of commit messages is one potential limitation of this study. Opinions about commit message usefulness will likely vary depending on the audience. Those familiar with the source code affected will likely require less information in order for the message to be useful. However, as mentioned in Chapter 4, research shows that even in a open source environments not all developers will work in the same area of a large project [4]. For automated messages, it would be better to output information that will be useful to a larger audience so long as the message does not become too verbose. Our evaluators might be giving a stricter evaluation of commit message usefulness because they are unfamiliar with the context of the changes and may need more information to claim a message is useful. However, if our evaluators believe that the message is useful, it is likely to be useful to those more familiar with the project. Nevertheless, their lack of specific knowledge about the projects does prevent them from being able to give accurate judgements about the contextual usefulness of the message. Therefore, their opinions about the usefulness of the commit messages should be taken as a judgement about whether the messages contain the right kinds of information with the appropriate level of detail based on their own experience working in team software projects.

Another limitation on our survey is that the respondent demographic were largely made up of undergraduate students, only a few of which had worked as actual developers. However, this is mitigated by the fact that all students asked have already used software repositories while working on team projects that simulate software development. Also, since not everyone finished the survey, some of the commit messages examined did not have 3 opinions on their usefulness and the quality of their summaries. However, there were enough complete responses to draw conclusions from the data. Finally, since the overall set of the data is small, we are cautious about our

conclusions, and as such they are intended to be suggestions for a preliminary model of an automated commit message generating tool.

Ideally, the results of this survey could be used to develop a larger survey on a more diverse group to obtain a better idea of the standards and practices of developer-written commit messages. Additionally, it might be useful to have perform a survey where software engineering students judge the usefulness of commit messages from their own projects. This would permit questions about the usefulness of commit messages within the context of the projects themselves. Though the standards used on the project would differ somewhat from open source projects, using student projects would be more feasible than trying to contact developers of specific open source projects for opinions about the usefulness of their own documentation.

Chapter 6

INDEPENDENT ANALYSIS OF DELTADOC

Among available techniques for measuring program change, DeltaDoc's [5] output is the closest to modeling actual commit messages. Unlike other methods mentioned in Chapter 3, it generates output intended to supplement developer written commit messages. However, the output seems to resemble pseudocode and they apply only a minimal amount of natural language transformations. It does not resemble developer written commit messages. Raymond Buse, one of the creators of the algorithm has made a distribution of the source code available. Therefore, we performed an independent analysis of the source and its functionality to answer the following questions:

- How does the distribution of DeltaDoc compare with the algorithm described in the paper?
- How well does DeltaDoc summarize source code changes compare to the raw UNIX diff in regards to correctness and conciseness?
- Based on the results from our survey, does DeltaDoc output compare with the content and scope that users of software repositories want in a commit message?
- Is DeltaDoc a good platform to extend to achieve the kinds of commit messages developers desire? Can its output be enhanced using natural language processing techniques?

6.1 Overview of DeltaDoc

The DeltaDoc distribution was initially designed to take 2 Java source code files and output a summary of the differences of the files by using path predicates. It has a few different options for output display, but for the purposes of this study, we only consider the default plain text output intended to approximate textual commit

messages. DeltaDoc requires Java 7 to run, and uses the Eclipse Abstract Syntax Tree (AST) Parser to extract and represent information in the source code. ASTs are data structures representing the syntactic components of source code, such as loops, conditionals, and variable declarations. They are simplified representations of the actual source code, and are commonly used for program analysis.

6.2 Test Inputs

We used two sets of input to test the functioning and output of DeltaDoc. One set consisted of 4 pairs of manually created Java files. Each pair was designed to test different parts of the output described in the DeltaDoc paper. The second set of inputs, 57 in total, came from Java source code files extracted from the open source projects JFreechart, Atunes, DrJava, and Phex. These commits were chosen arbitrarily from the projects, although there were some limitations on which files we were able to extract. For instance, when we tried to extract commits from early in the project's development on some repositories, svn was unable to locate and export the files in the commits. Then, these source code files were altered by a preprocessing script to remove structures in the code that DeltaDoc was unable to handle. These structures, and the method for handling them, are described in the section "Input and Performance Limitations of DeltaDoc".

6.3 Comparing Distribution to Documentation

Although we used the DeltaDoc paper as a guide for using the distribution, we double checked the source code and the inputs and outputs to see how closely the distribution matched what was described in the paper.

6.3.1 Brief Source Code Review

The code of the DeltaDoc algorithm seems to follow what is described in the paper somewhat closely, although there are some differences. The program takes two Java source code files as input. Then the Eclipse AST parser is run on each file to create a syntax tree for the file's source code. The parser separates out the classes,

methods, and variables in the file, and different types of code structure are tagged and put into the tree. Once the tree is built, DeltaDoc does a symbolic execution of the code to build the control flow graphs and path predicates for each method in the files.

After the setup stage, the algorithm begins to calculate the differences between the two files. However, the first stage is a filtering of relevant statements from irrelevant statements. Statements that are considered relevant by the algorithm are return statements - which contain the result of the called method, error throwing statements, and method invocation statements. These filters are fairly close to those described in the paper, although the paper suggests they added a filter to remove accessor methods, i.e. "methods of the form get[Field]()" [5]. After the filters have been applied, DeltaDoc begins searching for differences between the files. This process starts at the class level and proceeds inwards towards methods and then finally program statements. At the highest level, added and removed classes are documented, then modified classes are examined. Within the modified classes, added and removed attributes and methods are documented, and then modified methods are examined. When comparing relevant statements inside the method, DeltaDoc uses the generated path predicates to determine how to classify the statement. The four possibilities are listed below:

1. If there is a statement in the new version no longer reachable by any path, it is classified as removed.
2. If there is a statement in the new version that was not reachable by any path in the old version, it is classified as added.
3. If there is a program path that leads to a statement that it did not previously, it is associated with output of the form "If X Do Y instead of Z" statement.
4. If there is a statement that is only reached from paths different than it used to be, it is associated with output of the form "If X Do Y instead of When Z" statement.

After computing the differences between the two files, DeltaDoc converts them to a series of formatting objects. These objects are used for formatting the output and for applying further summarization. Specifically, the type of summarization being applied finds and removes shared subexpressions within the path predicates. This process of

removing common subexpressions and simplifying is only one of the summarizations mentioned in the DeltaDoc paper. The previously mentioned statement also reduces the length of the output. In fact, regarding the effectiveness of these summarizations, Buse and Wiemer's own study found that filtering what statements were relevant gave by far the largest reductions in size [5]. However, we did not find that the final summary transformation described in the paper was being applied to the output. In the paper, when the output was longer than 10 lines, a more general summarization was applied; it listed only key statements to give a general sense of the affected code areas. We did not see this final summarization being performed for the output in the provided distribution.

6.3.2 Input and Performance Limitations of DeltaDoc

When running the DeltaDoc code, the initial phase performed by the AST parser at first crashed on virtually all input from actual open source projects, including files from commits that were explicitly mentioned as being tested in the DeltaDoc paper. For example, the program did not first complete on the files from JFreechart's revision 1155 to 1556, which was used as an example in the paper[5]. The problem was that certain types of Java code structures were unable to be interpreted by the program. Upon contacting Raymond Buse about this issue, he confirmed that the provided version of DeltaDoc was later extended, although that implementation was not being provided. Therefore, in order to handle these structures and run DeltaDoc, a small program was written to remove these structures from the program. In the case of statements that did not affect control flow, the statements were changed into comments. In the case of problematic content in statements that do affect control flow, the content was replaced instead. For 'if' statements, since DeltaDoc would be unable to interpret the content of the statement anyway, the 'if' statement was changed to always resolve as false, and hence the control flow path would be cut short. The set of statements that the DeltaDoc distribution did not parse included:

- Bitwise operators such as ">>", "|", "^", and "&"

- The "?" operator, used to express conditionals concisely
- "Do while" loops.
- Calls to superclasses.
- Any line referencing the .class object

This list was gathered experimentally, and is not intended to be a complete list of all Java structures the distribution is unable to parse.

Secondly, while running Deltadoc, we discovered performance issues on some forms of input. In the DeltaDoc paper, it is was mentioned that the program typically finishes each set of inputs within a few seconds. However, the authors did note that since their algorithm uses symbolic execution to generate path predicates, it is potentially exponential [5]. We observed that in many cases, including all the manually created test files, DeltaDoc completed within a few seconds. However, in some cases, the program performance was poor. Some file pairs extracted from JFreeChart took several hours to be processed by DeltaDoc. PiePlot.java, for example, which changed between revision 132 to 133, had to be left running overnight before it completed. By examining debug output, we were able to confirm that this slowdown was the result of large numbers of control flow paths (over 10000 in one method) being explored during symbolic execution, and also from the time necessary to compute the differences and summarize them in. In the case of PiePlot.java, the path predicates that were being summarized were several pages long. Comparatively, predicate generation completed faster than the process of summarizing the output. It is uncertain if these slowdown issues are the result of poor implementation and can be fixed, or if they are potentially a problem with any implementation of summarization based on path predicates and symbolic execution.

6.4 Comparing DeltaDoc Output to Diff Output

For convenience in comparing DeltaDoc output to the file diff output, we divided DeltaDoc output into three categories: no change found, change but no modified

methods, and change inside methods. The first category of output is produced when DeltaDoc deems that there is no significant change to the file, output as "[No Appreciable Change]". 22 out of the 57 test cases fell in this category. We observed several different cases in the diff files when this occurred. In a few cases, this was because the preprocessing script had commented out the changed lines. Nevertheless, the distribution of DeltaDoc would not be able to handle these anyway. In the other cases, we observed that DeltaDoc identified no change when there were changes to modifiers in variables (public/private, or final), changes to the human written comments in the source code, and changes to import statements at the top of the file.

Additionally, DeltaDoc did not document changes to local variable assignments, which are described as being left out in the DeltaDoc paper [5]. For instance, DeltaDoc did not document a line "boolean isAdded = trackedAddresses.add(address);" which changed to "boolean isAdded = trackedAddresses.add(pong.getPongAddress());". In another case, DeltaDoc did not report changes in object type used in the file. For instance, an object had its type changed from "DefaultDestAddress" to "DestAddress". Additionally, there where some other cases were DeltaDoc left out changes in variable assignment, such as when "caughtHostsContainer = new CaughtHostsContainer();" changed to "caughtHostsContainer = new CaughtHostsContainer(hostFetchingStrategy);". While the paper mentions changes to methods as important to the algorithm [5], perhaps constructors are considered differently from other method invocations.

In the remaining 35 cases where DeltaDoc did report change within the code, we divided them into cases where DeltaDoc produced path predicate outputs and cases where it did not. The path predicate summaries were used only in cases where DeltaDoc identified statements changing within methods. 19 of these outputs had no path predicate output and the other 16 did. Disregarding 4 cases where the predicates had been clearly generated incorrectly, the median length of the DeltaDoc output was 8 lines versus the median diff output which was 28 lines, so DeltaDoc does give a more concise summary of the changes to the file on average.

Finally, we give a few comments on the structure of output in these cases. In

the case without file predicates, DeltaDoc's performance was accurate; it correctly identified removed and added fields and methods in the file. Cases where DeltaDoc failed to document all changes fell into the set of examples described in the no change case. When documenting changes within methods, DeltaDoc's path predicate output exhibited few oddities. First, the output often described the change in terms of a conditional if statement, even when this was unnecessary. The following output is a good example:

when calling AboutDialog

if TRUE

instead of

setSize(500, 400)

do

getRootPane().setDefaultButton(_okButton)

setSize(550, 400)

From examining the diff output and actual files, we found no conditional statement surrounding these statements within the method. It seems that DeltaDoc sometimes defaults to describing change in terms of an 'if' statement, even if there was no such statements in the original method being summarized.

Additionally, while DeltaDoc usually identified what changed correctly, it did not always output the changed statement in the correct form. On one of the manually created test cases, it reversed the meaning of a boolean expression from "return subfunctionZ() >= test;" to "return test >= subfunctionZ();" Other than examples of this type, the observed output of the predicates behaved as described, giving a more concise summary than the diff output.

6.4.1 Comparing DeltaDoc Output to the Survey Results

Recall in Chapter 5 that software repository users who responded to our survey indicated that the information most desired in commit messages is *what* changed as opposed to *how* the change was implemented. In the DeltaDoc paper, the stated goal of DeltaDoc is to display this *what* information as opposed the higher level *why* information [5]. However, no distinction was made in the paper between *what* and *how* information. The difference between the two is the level of abstraction, with *what* information being a overview not considered with local implementation specifics. The concern is that if DeltaDoc output is too low level, it will not capture the desired level of abstraction in a commit message.

In the case of small changes of only a few lines, it is difficult to divorce the implementation of how something changed from describing what changed and in these cases DeltaDoc output is better suited to describing the change. But when many statements change, listing path predicates seems to be too closely tied to the code, requiring either good knowledge of the code within the file or requiring the reader to read the files in addition to the output. Buse and Weimer try to address larger scale changes with higher level summarizations in their paper, which forgo path predicates to list changes in invocations, returns, etc. [5]. Unfortunately, this summarization technique was not seen in this distribution, and regardless is not always applicable to longer output, such as when large numbers of methods are added as in the example below:

ImportToRepositoryAction
added field : IRepositoryHandler repositoryHandler
added field : IErrorDialogFactory errorDialogFactory
added field : IFrame frame
added field : IMultiFolderSelectionDialogFactory multiFolderSelectionDialogFactory
added field : ILookAndFeelManager lookAndFeelManager
added method : executeAction

added method : setRepositoryHandler

added method : setErrorDialogFactory

added method : setMultiFolderSelectionDialogFactory

added method : setFrame

added method : setLookAndFeelManager

removed method : actionPerformed

Finally, in regards to identifying *where* information, DeltaDoc does identify affected classes, fields, and methods. Since DeltaDoc only operates between two files, it is not necessary for it to output location information at a larger scope. While simple to include, our survey results indicated information of this sort was important to include in commit messages. However, no attempts were made to relate the path predicates or changed fields to a particular aspect or feature of the application.

6.4.2 Conclusions and Capacity for Extension

As a whole, this distribution of DeltaDoc seems to be too problematic for practice. Some of the summarization capabilities described in the paper weren't used, and we experienced many bugs while trying to run the tool. While the range of acceptable inputs was limited, that part was relatively easy to handle. However even if the program worked correctly, the potential of exponential time performance suggests that this method of documenting change may not be appropriate for documenting changes at the level of a commit. Commits typically only modify a few files, and its unclear if it is necessary to generate large path predicates with symbolic execution in order to document changes of this size.

However, if the output of DeltaDoc can be generated without bugs and the performance issues are not a problem, the output can be improved with natural language processing techniques. For instance, path predicates can be long and difficult to interpret for developers not familiar with the code in a particular file. Consider the

following output from DeltaDoc:

when calling fromOtherViews

if TRUE

do

indeterminateProgressDialogFactory.newDialog(frame,

lookAndFeelManager).setTitle(I18nUtils.getString("PLEASE_WAIT"))

new DeleteFilesWorker(indeterminateProgressDialogFactory.newDialog(frame,

lookAndFeelManager), new LocalAudioObjectFilter().getLocalAudioObjects(

navigationHandler.getFilesSelectedInNavigator())).execute()

instead of

new DeleteFilesWorker(new LocalAudioObjectFilter().getLocalAudioObjects(

navigationHandler.getFilesSelectedInNavigator())).execute()

Using this output as an example, we demonstrate cases where extracting natural language could significantly improve the readability of the output. The method developed by Sridhara et al. to create natural language summaries for code statements could be applied here [29].

Likewise, it is questionable if DeltaDoc's method of merely giving method names that have been added and removed is appropriate for describing the level of change in the added code. Information extracted from their associated Javadoc comments or generated from statements within the code [29] might be more effective in explaining the change to all the potential audiences of a commit message, not just the few developers intimately familiar with the code in that particular file. Finally, when many fields and methods are added or removed, simply listing them seems to be tied too closely to implementation. If we extracted words and phrases from these methods, identifying patterns would enable a more concise and abstract summary to be written.

6.5 Threats to Validity and Future Work

This observational study is intended to be a cursory examination of DeltaDoc and its output. As such, its conclusions should be considered only a starting point. However, the difficulties of using this distribution of DeltaDoc make it difficult to do further studies without significant work. However, if we were to more definitively classify the qualities of the output, including whether it focuses too much on implementation level information, a formal study with independent evaluators could be performed. Likewise, the performance issues mentioned were only observed in practice and not measured systematically. Anyone looking to use DeltaDoc should consider experimenting to determine what situations in code are correlated with exceptionally long run times, and evaluate how frequently these situations occur.

Chapter 7

CONCLUSIONS AND FUTURE WORK

When making changes to software, developers spend more time trying to understand code rather than implementing changes. Critical to assisting developers in understanding code is human-written documentation. Unfortunately, in many contexts the documentation is not as good as it could be. Therefore, if linguistic information can be extracted from code and presented as documentation, it provide support for developers when in cases where documentation is lacking.

Commit messages are a type of documentation closely tied to changes in software. While work has been done on both extracting information about changes between software versions and on parsing linguistic information from static versions of source code, relatively little has been done in combining the two areas. While one program, DeltaDoc [5] attempts to generate commit messages, they apply relatively little in the way of natural language transformations to their output.

7.1 Conclusions

In this Thesis, we sought to create a grounding for what output would be most appropriate for mimicking developer-written commit messages. We first performed an observational study of commits and their associated messages in open source projects to examine their linguistic and non linguistic properties. We found that commits were typically modifications of no more than a few code files and that commit messages were typically only a few lines in length and usually not more than 25 words long. Linguistically, verb, noun and prepositional phrases dominated the commit messages.

We also extracted verbs and their associated direct objects from these commit messages and presented this as a preliminary model of output to users of software

repositories. While we found that commit messages were read at least infrequently by all survey respondents, the method of verb-direct object summarization was often misleading and left out important information. It would leave out the context of *where* in the code the changes occurred. By observation, we discovered that this information was often located in prepositional phrases, which had not been included in the original model. Additionally, we found that users generally preferred that implementation details be left out of commit messages. On the other hand, we also found that higher levels of abstraction in commit messages, such as relation to project goals, should at best be supplemental information to a concise summary of *what* changed. Evaluators also rated the usefulness of existing commit messages and found about two thirds of them useful, although they often disagreed on how useful a commit message was.

Finally, we performed an independent evaluation on a distribution of DeltaDoc [5]. We found this distribution difficult to use on code extracted from actual open source projects, and it deviated somewhat from what was described in the paper. It required additional preprocessing scripts to remove certain unsupported code structures, and in some cases we saw very poor time performance. However, if a more stable distribution of DeltaDoc was used and underlying concerns about the potentially exponential performance of the algorithm can be addressed, then DeltaDoc's output can be extended with existing natural language techniques to improve the readability of the output.

7.2 Future work

As this Thesis establishes a target for potential commit message output using verb, noun, and prepositional phrases, the next step would be to begin developing an initial version of a tool to produce this output. This could proceed one of two ways. One approach would be to debug DeltaDoc and try to limit performance errors so that the distribution models the behavior described in the paper more accurately. Then techniques developed in SWUM [14] and Javadoc comment generation [29] could be used to improve the output readability. Or, these linguistic models of word use in source code could instead be used as a starting point instead. Output could be

generated based on finding changes in the model from version to version. Either way, the output from this process should be presented to a larger set of evaluators and compared with existing commit messages, diff output, and potentially with DeltaDoc output as well. Further changes and refinements to the model would depend on the results of this evaluation. Ideally, this would eventually become a basis for a tool or add-on to repository software to generate commit messages in order to supplement existing developer written messages.

BIBLIOGRAPHY

[1] S.L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 156–159, 2010.

[2] Juan Jose Amor, Gregorio Robles, Jesus M. Gonzalez-barahona, and Alvaro Navarro. Discriminating development activities in versioning systems: A case study ?, 2006.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, March 2007.

[4] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA, 2008. ACM.

[5] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[6] M. D'Ambros. Commit 2.0: enriching commit comments with visualization. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 529 –530, may 2010.

[7] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, SIGDOC '05, pages 68–75, New York, NY, USA, 2005. ACM.

[8] Natalia Dragan, Michael L. Collard, Maen Hammad, and Jonathan I. Maletic. Using stereotypes to help characterize commits. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 520–523, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. Reverse engineering method stereotypes, 2006.

[10] Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. Using method stereotype distribution as a signature descriptor for software systems. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 567–570. IEEE, 2009.

[11] Isabel Drost, Grant Ingersoll, Benson Margulies, Thilo Goetz, Jrn Kottmann, Jason Baldridge, James Kosin, Thomas Morton, William Silva, Boris Galitsky, and Aliaksandr Autayeu. Apache opennlp.

[12] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Softw.*, 18(3):112–111, May 2001.

[13] L.P. Hattori and M. Lanza. On the nature of commits. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 63 –71, sept. 2008.

[14] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration.* PhD thesis, University of Delaware, Newark, DE, USA, 2010. AAI3423409.

[15] A. Hindle, D.M. German, M.W. Godfrey, and R.C. Holt. Automatic classication of large changes into maintenance categories. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 30 –39, may 2009.

[16] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[17] IEEE. International standard - iso/iec 14764 ieee std 14764-2006 software engineering 2013; software life cycle processes 2013; maintenance. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 1–46, 2006.

[18] D. Jackson and D.A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 243 –252, sep 1994.

[19] Mik Kersten. *Focusing knowledge work with task context.* PhD thesis, University of British Columbia, Vancouver, BC, Canada, 2007.

[20] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 202 –211, march 2010.

[21] J.I. Maletic and M.L. Collard. Supporting source code difference analysis. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 210 – 219, sept. 2004.

[22] Lori Pollock, Vijay Shanker, David Shepherd, Emily Hill, Zachary Fry, and Kishen Maloor. Introducing natural language program analysis. In *In 7th ACM SIGPLAN-SIGSOFT Workshop of Program Analysis for Software Tools and Engineering*, June 2007.

[23] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448. ACM Press, 2004.

[24] Romain Robbes, Michele Lanza, and Mircea Lungu. An approach to software evolution based on semantic change. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 27–41, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), February 2007.

[26] Beatrice Santori. Part-of-speech tagging guidelines for the penn treebank project (3rd revision, 2nd printing), 1995.

[27] 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 *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 212–224, New York, NY, USA, 2007. ACM.

[28] Sourceforge. http://sourceforge.net/, 2013.

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

[30] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[31] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

Appendix A

HUMAN OPINION SURVEY ON COMMIT MESSAGES AND EXTRACTED SUMMARIES

The following appendix displays the instructions and questions asked in the online survey in approximately the form they were display to the survey respondents. Since the survey dynamically loaded commit messages and different questions depending on the respondents answers, only the questions are displayed.

A.1 Background:

Our current research goal is to identify an effective way of representing commit messages found in software repositories such as svn, with the ultimate goal of being able to generate commit messages from code that meaningfully summarize information developers would find useful. This purpose of this study is to determine if the approach of extracting verb phrases from commit messages and pairing them with their direct objects adequately retains the information found in a commit message.

A.2 Instructions:

Participation in this survey is completely voluntunary, and completing the survey should take approximately 20 to 30 minutes.

First, you will be asked several questions about your software engineering experience and general usage of commit messages. Then, you will be asked a few questions about 20 commit messages, separated into 2 groups of 10. Group 1 includes messages that our tool summarized with a verb-direct object summary. Group 2 contains messages that were not successfully summarized.

In order to keep your comments grouped together as a single user, you will be asked to input a username and password before starting. If you do not finish the survey, the results you have submitted will be saved and you may return later.

Finally, please note that once you submit your answers for a commit message, you cannot go back and change them. Make sure your answers are what you intended before submitting.

A.3 Overview

How many years of programming experience do you have?

1. 0-5 years
2. 5-10 years
3. 10-15 years
4. 15-20 years
5. over 20 years

How would you describe your current position:

1. Undergraduate student
2. Graduate student
3. Software Engineer
4. Developer
5. Professor
6. Other:

1. What sorts of information do you put in commit messages? (Select all that apply.)
   - An explanation of how I implemented a change.
   - An explanation of what changed in the code.
   - An explanation of what features the changes intend to support.
   - Other:

2. Do you read commit messages when working on a software project?

1 (Never) - 5 (Always)

3. Why do you read commit messages? (Select all that apply.)

- To confirm my correct understanding of the code changes.
- To get a higher level view of the changes.
- To get the intent/goal of the changes.
- Other:

4. Under what scenarios do you read commit messages? (Select all that apply.)

- When it is a commit message I wrote earlier, and I need a quick summary of what I did.
- When I want a high-level view of what work is being accomplished.
- When I need to trace down a new error in the application.
- Other:

5. When working in a development team, what do you find most useful in commit messages? (Select all that apply.)

- A short summary of the changes made.
- What project goal the changes are supporting.
- How the changes in this revision were implemented.
- Other:

A.4 Group 1

1. How useful is the commit message in conveying information about how the code changed? (Without having to look at the code.)

1 (Not at all)-5 (Very useful):

if Question 1 has a response of 3, 4, or 5:

2a. Overall, do the extracted clauses convey the main action(s) portrayed by the commit message?

1 (Not at all) - 5 (Completely)

3. Does the summary created by the extracted clauses contain misleading information?
1 (Very misleading) - 5 (Not misleading)

4. Do the extracted clauses contain unnecessary information?
1 (A lot of unnecessary information-5 (No unnecessary information)

5. Does the extracted summary leave out any important information?
1 (Missing a lot)-5 (No information missing)

5b: If information is missing explain:

if Question1 has response of 1 or 2:

2b. Please briefly explain why the original commit message does not contain useful information:

A.5 Group 2

1. How useful is the original commit message in conveying information about how the code changed? (Without having to look at the code.)
1 (Not at all)-5 (Very useful):

if Question 1 has a response of 3, 4, or 5:

2a. If the commit message contains at least one action, please enter what you believe to be the main action of the message. Note that the action may be in the form of a noun (e.g. Addition, other words ending in -tion, etc):

3. If there is no action, what do you think makes the commit message useful?

if Question1 has response of 1 or 2:

2b. Please briefly explain why the original commit message does not contain useful information:

Appendix B

ADDITIONAL EXAMPLES

B.1 Commit Messages and their Extracted Summaries

The table on the following page lists some examples of the commit messages and extracted summaries that we presented to survey correspondents:

Table B.1: Additional Examples of Commit Messages and their Extracted Phrases used on the Survey

Commit Message 1	Minor changes while writing chapter 14
Summary 1	writing chapter 14 -
Commit Message 2	Removed redundant entry, since it is merged into 4.3.
Summary 2	Removed redundant entry - it is merged -
Commit Message 3	Defend playerExploredTiles against NPEs.
Summary 3	Defend playerExploredTiles -
Commit Message 4	PdfContentByte.addImage() draws the borders
Summary 4	draws the borders -
Commit Message 5	Added Line Enumeration (FR #508676), but still buggy
Summary 5	Added Line Enumeration FR 508676 -
Commit Message 6	Moved the plugin from SVN to git.
Summary 6	Moved the plugin - to git -
Commit Message 7	Adds Michael's fix for updating the menu bar after changing the language.
Summary 7	Adds Michaels - fix the menu bar - changing the language -
Commit Message 8	Removed the child column index creation when creating a FK. Informix does this automatically when a FK is created.
Summary 8	Removed the child column index creation - creating a FK - does this - a FK is created -
Commit Message 9	Added a label to the logviewer dockable.
Summary 9	Added a label -