IMPROVING PROGRAM COMPREHENSION USING
NEURAL MACHINE TRANSLATION

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Siyuan Jiang

______________________________

Collin McMillan, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2018
This document is in the public domain.
Program comprehension is the task of understanding software projects. One effective way to help programmers comprehend code is documentation. However, manual documentation costs a significant amount of time and effort. Therefore, software documents are often incomplete and out of date. Automatic documentation generation is to address the problem of lack of documentation and to improve the quality of documentation.

In this dissertation, I summarize my work on improving program comprehension using neural machine translation to generate documents. First, I conduct two user studies about programmers’ behavior when they comprehend code before and after they make changes. Specifically, I study how programmers do change impact analysis, which is the task of finding source code that is affected by a change. The studies show that programmers do more change impact analysis before they make changes than after the changes. When programmers need to understand a change in a software repository, a summary of the change is an important component of the comprehension process. My first project of generating documents is to generate short summaries of software changes using neural machine translation. Neural machine translation (NMT) is a type of neural network for translating natural languages. This project demonstrates that NMT can also be used in translating from diff
(results of text differencing techniques) to English text. My second project focuses on topic labeling to generate descriptions of key functionalities in software projects. Topic labeling is the task of labeling hidden topics in topic models, which are often used in program comprehension tools and research to find key functionalities. In these tools and research, key functionalities in software projects are assumed to be the hidden topics in topic models. However, the topics are represented by lists of words with probabilities, which are difficult to interpret. Labeling topics is often a manual task. My project uses NMT to translate topics represented by lists of words to English text. The results show that NMT-generated descriptions are more helpful for programmers to understand software projects than the lists of words.
DEDICATION

To myself and Weijing Huang, Kenken Wu and Ping Jiang.
CONTENTS

Figures ........................................................................ viii

Tables ........................................................................ xi

Acknowledgments .......................................................... xii

Chapter 1: Introduction ..................................................... 1
  1.1 Programmers’ Behavior of Understanding Software Changes . . . . 2
  1.2 Documentation of Software Changes: Commit Messages .......... 2
  1.3 Documentation of Software Features ................................. 3

Chapter 2: Related Work .................................................... 5
  2.1 Empirical Studies about Software Changes and Change Impact Analysis (IA) .................................................. 5
  2.2 Empirical Studies about IA Practice of Programmers .......... 6
  2.3 Studies in Tool Adoption ................................................ 8
  2.4 Commit Message Generation Techniques ............................ 8
  2.5 Source Code Summarization Techniques ............................ 9
  2.6 Automatic Topic Labeling ............................................. 10
  2.7 Topic Modeling in Software Engineering (SE) ...................... 11
  2.8 Deep Learning in SE .................................................. 12
  2.9 Neural Machine Translation (NMT) in SE .......................... 12

Chapter 3: Study of Change Impact Analysis ............................. 14
  3.1 Introduction .................................................................. 14
  3.2 The Problem .................................................................. 16
  3.3 Background: Change Impact Analysis (IA) ......................... 17
    3.3.1 Classic IA Process .................................................. 17
    3.3.2 IA Techniques ....................................................... 19
    3.3.3 IA Activities ........................................................ 20
    3.3.4 IA Process, IA Techniques, and IA Activities ............... 21
  3.4 Background: Debugging ............................................... 22
  3.5 Empirical Study Design Overview .................................... 24
  3.6 In-Depth Study Design .................................................. 25
    3.6.1 Research Questions ................................................ 25
3.6.2 Methodology .................................................. 27
3.6.3 Subject Applications ........................................... 28
  3.6.3.1 PdfSam and the Subject Bug ....................... 29
  3.6.3.2 Raptor and the Subject Bug ....................... 30
3.6.4 Participants .................................................. 33
3.6.5 Threats to Validity .......................................... 33
3.6.6 Reproducibility .............................................. 35
3.7 In-Depth Study Data Collection ............................... 35
  3.7.1 Recorded activities .................................... 35
  3.7.2 Found change locations ................................. 35
  3.7.3 Found important times .................................. 36
  3.7.4 Measured distances ..................................... 37
3.8 In-Depth Study Results ........................................ 37
  3.8.1 Quality of Patches ..................................... 37
  3.8.2 Example Result ......................................... 40
  3.8.3 Aggregate Result ...................................... 40
  3.8.4 Difference between the study with the suggested fixes and the study without the suggested fixes .... 48
  3.8.5 RQ₁: Do programmers use any IA tools? ............ 48
  3.8.6 RQ₂: Do programmers navigate to the dependencies or the dependents of the first section of code they read? 50
  3.8.7 RQ₃: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of the first read and the first modification? 51
  3.8.8 RQ₄: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of modification and the first run of the altered code? .......... 52
  3.8.9 Qualitative Results ..................................... 53
    3.8.9.1 knowledge of IA .................................. 53
    3.8.9.2 practice of IA .................................. 53
  3.8.10 Comparison to Other Results of Code Changing Tasks .... 54
3.9 Breadth Study Design ........................................ 59
  3.9.1 Research Questions .................................... 59
  3.9.2 Methodology ............................................ 60
  3.9.3 Online Survey Format .................................. 61
  3.9.4 Participants ............................................ 61
  3.9.5 Threats to Validity .................................... 63
  3.9.6 Reproducibility ....................................... 63
3.10 Breadth Study Results ......................................... 63
  3.10.1 RQ₅: Knowledge of Term of Change Impact Analysis .... 63
  3.10.2 RQ₆: Self-reported Level of Change Impact Analysis .... 64
  3.10.3 RQ₇: Use and Knowledge of IA Tools .................. 64
  3.10.4 RQ₈: First Step after Having a Possible Fix .......... 65
3.1 The Process of Change Impact Analysis in the literature, described in Section 3.3.1 ................................................................. 18

3.2 The process of debugging according to the literature (see Section 3.4). This is the process that we expect our participants to follow in our studies. ................................................................. 23

3.3 All the research questions are introduced in Sections 3.6.1 and 3.9.1. In the in-depth study, RQ_1 helps to answer Objective 1 and Objective 2. RQ_2, RQ_3, and RQ_4 helps to answer Objective 3. In the breadth study, RQ_5 and RQ_6 helps to answer Objective 1. RQ_7, RQ_8, and RQ_9 helps to answer Objective 2. RQ_3: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they read a change location and the first time they alter the change location? RQ_4: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they alter code and the first time they run the altered code? ................................................................. 26

3.4 The original code before the fix in PdfSam. ........................................ 31

3.5 The code after the actual fix in PdfSam. ........................................ 31

3.6 The original code before the fix in Raptor. ........................................ 32

3.7 The code after the actual fix in Raptor. ........................................ 32

3.8 The distribution of the time lengths .................................................. 49

3.9 The distribution of the number of visited files .................................... 49

3.10 The distribution of the number of IDE functionality uses. .................. 50

3.11 Page 4 of the survey for our breadth study in Section 3.9.3. This page shows an image of an Eclipse IDE. The red boxes are the areas that participants can click. The areas include the window of Package Explorer, the button of Open Declaration in the context menu, and other menu options and Eclipse windows. We asked each participant to click on the area that she will go next after she makes a fix. This page is to answer RQ_9. ................................................................. 62
3.12 The results for RQ₅ to RQ₈ in our breadth study in Section 3.9. RQ₅: do programmers know the term of change impact analysis? RQ₆: do programmers think they do change impact analysis? RQ₇: do programmers use change impact analysis tools? RQ₈: after programmers make a possible fix, do they run the program first or check the impacts of the fix first? ................................. 67

3.13 The result of RQ₉ in our breadth study in Section 3.9. RQ₉: after programmers make a possible fix, what IDE functionality do they use? 67

4.1 The overview of this chapter. .................................................. 78

4.2 The architecture of the encoder in RNN Encoder-Decoder ............ 81

4.3 The architecture of the decoder in RNN Encoder-Decoder ............ 82

4.4 The detailed process in Part A, Figure 4.1. There are three main steps: 1) filtering and preprocessing the data; 2) training a Neural Machine Translation model; 3) evaluating the model, which is Part B, Figure 4.1 83

4.5 The distribution of the lengths of diffs in the test set ................. 91

4.6 The distribution of the lengths of diffs in the training set ............ 93

4.7 An scoring example we gave to the participants in the survey study. .. 96

4.8 One of the pages that we ask the participants to score the similarity. There is an optional text box for the participants to write their justifications in the end of the page. This text box is omitted due to space limit. ................................. 97

4.9 The distribution of the median scores obtained in the human study. There are 983 scores in the figure. Each score is the median score of the scores made by one to three human experts for a generated message. The scores range from 0 to 7, where 0 denotes the generated message is not similar to the reference message at all, and 7 denotes the generated message is identical to the reference message. The most frequent scores are 0 and 7. There are 248 messages scored 0 and 234 messages scored 7. For the rest of the scores, the number of messages ranges from 68 to 100. ................................. 98

4.10 Outline of our cross-validation process. .................................. 100

4.11 The predict results of the cross evaluation of QA filter. QA filter reduced 108 messages that are scored 0, 45 messages that are scored 1, 42 messages that are scored 2, 32 messages that are scored 3, 32 messages that are scored 4, 21 messages that are scored 5, 18 messages that are scored 6, and 16 messages that are scored 7. We note that although we trained the QA filter with binary labels, “bad” and “not bad”, the evaluation result shows that QA filter is able to reduce more messages for lower scores. ............................................. 102
5.1 An overview of our approach, which we describe in Section 5.3. The upper layer (areas 1, 4 and 5) is the training process. The lower layer (areas 2, 3 and 6) is the process of creating topic labels after training.

5.2 The overall quality of learned topics of all projects, Web projects, and Android projects in terms of NPMI. They share similar distributions.

5.3 A screenshot of our survey page which asks participants whether the labels help them understand the projects.

5.4 Distributions of Scores on Helpfulness. Score 1 means the participants strongly disagree that the given label was helpful; 2 means “disagree”; 3 means “agree”; and 4 means “strongly agree”. In all the sub-figures, most baseline labels have scores of 2; most our labels have scores of 3.

5.5 The distribution of the similarity scores of 415 pairs of the labels. Each pair of the labels corresponds to a same topic. Score 1 means “completely different”; 2 means “different”; 3 means “similar”; and 4 means “identical”. There are 106 pairs of the labels scored as 2. In total, 213 pairs have scores that are larger than 2.
### TABLES

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Two open-source programs used in our study</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Participants in the in-depth study</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>The time labels</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Eclipse navigational functionalities</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Quality of the patches</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>The measured distances for one change location</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>The overall results for 18 debugging sessions</td>
<td>44</td>
</tr>
<tr>
<td>3.8</td>
<td>The average distances for the 31 changes</td>
<td>45</td>
</tr>
<tr>
<td>3.9</td>
<td>The average distances for the 17 changes for PdfSam</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>The average distances for the 14 changes for Raptor</td>
<td>47</td>
</tr>
<tr>
<td>3.11</td>
<td>IDE activities in debugging PdfSam</td>
<td>57</td>
</tr>
<tr>
<td>3.12</td>
<td>IDE activities in debugging Raptor</td>
<td>58</td>
</tr>
<tr>
<td>3.13</td>
<td>The counts of the programmers</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>BLEU scores (%) of MOSES and our models</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>BLEU scores (%) on diffs of different lengths</td>
<td>93</td>
</tr>
<tr>
<td>4.3</td>
<td>Example result</td>
<td>101</td>
</tr>
<tr>
<td>4.4</td>
<td>The changes of the results that have high scores</td>
<td>103</td>
</tr>
<tr>
<td>4.5</td>
<td>The changes of the results that have low scores</td>
<td>105</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of topic keywords</td>
<td>130</td>
</tr>
<tr>
<td>5.2</td>
<td>The statistics of the helpfulness scores for RQ₁ and RQ₃</td>
<td>131</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

This dissertation represents six years of my work at the University of Notre Dame. I am blessed with various help from many people, without whom this dissertation could never have happened.

- I would like to thank Collin McMillan, my advisor, for his guidance, understanding and strong support for my graduate study and future career.

- I would like to thank Raul Santelices, my previous advisor, for his guidance in the earlier stage of my Ph.D. study.

- I would like to thank Mark Grechanik, Jane Cleland-Huang, David Chiang, Douglas Thain, Gregory Madey, and Aaron Striegel for their committee service and recommendations.

- Personally, I would like to thank my dear friends, Jim and Barbara Blechl, and all the friends in the small group for their extreme kindness and support.

- I would like to thank my physical trainer, Neil MacDonald, from whom I learn to be stronger and more persevering.

- I would like to thank the staff at University Counseling Center at the University of Notre Dame.

- I would like to thank my uncle, who does not agree on my decision of pursuing Ph.D., but still has been caring and supporting me in these years.

- Above all, I would like to thank my husband, Weijing, and my mother, Kenken Wu, who have been with me through all the highs and lows.
CHAPTER 1

INTRODUCTION

Programmers rely on documentation to understand software projects [52, 98, 114], but documentation is very time-consuming to create manually. One major reason is that software changes over time. It costs much time and effort to update documents every time software changes. Therefore, software often lacks up-to-date documentation [43, 81, 114]. To address this problem, a growing research area is automatic generation of documentation. An existing method is to generate documents by automatically organizing code comments. For example, Javadoc [130] and Doxygen [168] are two popular tools that extract code comments to generate documents about source code. Although these tools are useful, they require source code comments, which are manually maintained, to be updated in order to generate up-to-date documents.

To further reduce the workloads for programmers to write documentation, over the last few years, researchers have proposed automatic generation of the natural language contents of documentation [62, 63, 114, 124, 158]. Specifically, many tools have emerged to generate or augment API documents automatically [28, 34, 77, 84, 161, 167]. For example, Buse et al. propose a technique of describing Java exceptions [28]; Treude and Robillard augment API documents with information from Stack Overflow [167]; Stylos et al. create Jadeite, which is a Javadoc-like API documentation system that allows users to add usage information to API documents.

In this dissertation, I focus on improving automatic documentation generation of software changes and software features. Towards this goal, I work on three questions. First, how do programmers understand changes? Second, how can high-level de-
criptions of software changes be generated automatically? Third, how can software features be described automatically? This dissertation covers the three questions in three chapters (from Chapter 3 to Chapter 5).

1.1 Programmers’ Behavior of Understanding Software Changes

One of the tasks for understanding software changes is change impact analysis. Given a software change, change impact analysis is the task of finding the software parts affected by the change. In literature, researchers often assume programmers do change impact analysis after they make changes to software [137]. However, there is little empirical evidence about programmers’ behavior of change impact analysis. To address this problem, I conducted two user studies: an in-depth study and a breadth study. In the in-depth study, I hired nine programmers to debug two bugs in two open-source Java applications. Then, I examined the programmers’ behavior in their debugging processes. My finding is that programmers do more change impact analysis before they make changes, which contradicts to the previous assumption: programmers do change impact analysis after they make changes. In the breadth study, I hired 35 programmers to fill a survey about their knowledge and behavior of change impact analysis. The results are consistent with the finding of the in-depth study. This finding explains why prototypes of change impact analysis techniques have not been adopted in practice.

1.2 Documentation of Software Changes: Commit Messages

With revision control systems, each software change can be stored as a commit. When programmers submit a commit to a repository, they often write a message about the software change, which is the commit message. To generate commit messages automatically, Buse et al. create DeltaDoc, a tool that summarizes control flow changes [29]. Likewise, Linares-Vásquez et al. build ChangeScribe, which summa-
rizes method-level and class-level changes [103]. These tools are based on pre-defined rules and templates to generate summaries of specific types of software changes, for example, method name changes. To deal with more types of software changes and generate summaries that are similar to human-written ones, I introduced neural machine translation approach for generating descriptions of software changes. Neural machine translation (NMT) is a type of neural network for translating natural languages. To train such a neural network, I obtained thousands of commits from Github as the training and testing datasets. Each commit corresponds to a diff (the result of a text differencing tool for the commit) and a commit message. The test of my trained NMT model shows a promising result where the NMT model can generate short commit messages that are almost identical to what programmers write according to both BLEU score [134] and human evaluation.

1.3 Documentation of Software Features

The descriptions of software features can help users and managers understand software quickly. Like other documentation, software often lacks feature descriptions [120]. To address this problem, software engineering research community works on the area of automatic feature discovery, which is the task of identifying features in source code. Existing feature discovery solutions often treat this task as a task of finding topics in a collection of text files, where features are the topics and the source files are the text files [14, 115, 156].

To find abstract topics in text files, text mining techniques often use topic models, which are statistical models for discovering abstract “topics” from text corpora [20, 133]. A “topic” in topic models is defined as a list of words with probabilities. Each word has a probability, which represents the chances of the word being used for the topic. But topic models do not have labels or descriptions of the topics. For example, in Natural Language, if a topic is (“dog”, “cat”, “hamster”), each
word with a probability of 0.4, 0.4, and 0.2 respectively, the topic is probably about “pets”, but the topic model has no such label for the topic. One popular solution for labeling topics is to use the words with highest probabilities as the label of a topic, for example, “dog, cat” as the label of the example topic.

Like the practice in text mining techniques, in feature discovery projects, the features (i.e., the topics in the source code files) are often labeled with the words that have the highest probabilities [11, 36]. However, these labels are often not comprehensible because files in software projects are full of jargon and technical terms. To automatically generate natural language descriptions of topics, I trained a neural machine translation (NMT) model to translate from the word lists of topics to English descriptions. To train such a model, I obtained thousands of pairs of the topics and the class comments, which are assumed to be the descriptions of the topics. The test result shows that NMT can generate topic labels that are more comprehensible than the word lists.
CHAPTER 2

RELATED WORK

This chapter covers two types of related work. The first type of related work studies the research areas that this dissertation covers, including change impact analysis techniques, the practice of change impact analysis, tool adoption, automatic generation of commit messages, automatic generation of source code summaries, and automatic topic labeling.

The second type of related work includes the research projects in software engineering that have used methods similar to those in this dissertation, including topic modeling, deep learning, and neural machine translation.

2.1 Empirical Studies about Software Changes and Change Impact Analysis (IA)

Many empirical studies are conducted to study software changes, their sizes and how to predict changes. Gethers et al. created four benchmarks for IA from four open source Java systems [55]. In the total of 277 change requests, half of them were addressed by the changes within five methods. Seventy-five percent of the change requests were addressed by changing fewer than 13 methods. Gethers et al. introduced an integrated impact analysis approach including textual information analysis, evolutionary information analysis, and execution information analysis. The highest precision of this approach is 18%, and the highest recall is 75%. Ye et al. created the benchmarks from five Java projects [179]. The number of the collected bug reports in each benchmark ranges from 593 to 6,495. The maximum number of the files fixed for a bug report in each benchmark ranges from 87 to 587. However,
the median number of the files fixed in each benchmark ranges from one to three. In 70% of the bug reports in Eclipse Platform and Tomcat projects, the recommending system of Ye et al. includes at least one fixed file in the top ten ranked files.

For changes not in source code, McIntosh et al. mined the relationship between source code changes and build changes in four large systems [118]. They found that only 4% to 26% of the source code changes require build changes. Using code change characteristics, McIntosh et al. developed classifiers to explain when build changes are necessary.

There are also many empirical studies evaluating different types of IA techniques. Acharya and Robinson did an empirical study of IA based on static program slicing in ABB [5]. With the high setting defined in their paper, on 147 changes they ran, about 45% of the static slices had more than 300,000 lines of code, and about 46% of the static slices had fewer than 10,000 lines of code. Wu et al. explored the dependencies among programs and other binaries [174]. Without simplifying the dependency graph, a dependency graph for \texttt{wget} has 26 nodes, which means there are 25 binaries interacted with \texttt{wget}.

Rungta et al. introduced a change impact analysis technique, iDiSE, and applied this technique to \texttt{tcas} [148]. In the 16 version changes in the paper, at least 57% of the constraints in the changed versions are impacted by the changes. Dam and Ghose implemented an IA technique for agent systems [41]. The results showed that the precision of the technique ranges from 25% to 50% and the highest recall is 65%. Parande and Koru studied five KOffice products and found the dependencies concentrated more on smaller modules [135].

2.2 Empirical Studies about IA Practice of Programmers

De Souza and Redmiles went to two large software teams and studied the actual IA approaches—team strategies that handle software artifact-level dependencies [42].
From this study, De Souza et al. introduced impact management for team management of dependencies and changes, which models how developers inform impacts to others or prevent impacts from others. The difference between my study (in Chapter 3) and De Souzas study is that my study focused on how programmers do IA individually and their study focused on how programmers communicate IA results with others.

To discover important issues in IA, Rovegård et al. did an empirical study at Ericsson AB, Sweden [145]. They interviewed 18 people at different organizational levels. Rovegård et al. mapped three organizational levels in software engineering—technical, resource and product—to three levels in decision-making: operative, tactical, strategic levels. They found important IA issues from both organizational and personal views. For example, the issue “affected parties are overlooked” is seen as a high priority both in organizational and personal view. They also found that personal views affect organizational views on IA issues. Tao et al. did a large-scale online survey and follow-up email interviews in Microsoft [164]. From the study, they also identified a series of questions about understanding changes.

Several empirical studies have focused on individuals in software engineering. To study how programmers navigate through source code while they debug, Lawrance et al. recruited 12 programmers from IBM, and asked the programmers to debug an open-source project, RSSOwl[97]. Lawrance et al. required the programmers to think aloud when they debugged, and recorded videos and audios of their debugging process. Based on the videos and audios, they investigated how information foraging theory was applied to the navigation behaviors of programmers. In this dissertation, to answer the first question of how programmers understand software changes, I use an approach similar to what Lawrance et al. did. The difference is that they aim to study the debugging process, and I aim to study change impact analysis.

Wetzlmaier and Ramler also did a study on IA practice of individual program-
mers \cite{70}. They studied how well the experienced developers estimated changes. They found that the estimation of the developers is inaccurate. The difference between their study and my study is that their study finds out the accuracy of IA while my study finds out how developers do IA.

2.3 Studies in Tool Adoption

Storey reviewed the tools in program comprehension based on cognitive theories \cite{159}. In this review, Storey identified six aspects of tool support for program comprehension: documentation, browsing and navigation support, searching and querying, multiple views, context-driven views, and cognitive support. Additionally, Storey et al. did a study of 30 programmers using three tools to solve high-level program comprehension tasks \cite{160}. Based on their observation, they argue that the tools should support for different comprehension strategies, for example, bottom-up \cite{153} and top-down \cite{27}.

Bassil and Keller surveyed about software visualization tools with more than 100 participants \cite{15}. They found that the participants were satisfied with the current software visualization tools. In the data collected, the functionality of searching for graphical and textual elements is rated as the most useful one in visualization tools. Regarding practical aspects of visualization tools, the reliability and the ease of using the tools are rated as the two most important aspects. Kienle and Müller did a literature survey about requirements of visualization tools \cite{83}. They identified seven quality attributes, for example, usability, and seven functional requirements, for example, views and search.

2.4 Commit Message Generation Techniques

In this dissertation, I categorize the commit message generation techniques into two groups based on the inputs of the techniques. The first group uses code changes
of a commit as an input, and summarizes the changes to generate the commit message. For example, Buse et al. have built DeltaDoc, which uses symbolic execution to extract path predicates of changed statements, and follows a set of predefined rules to generate a summary of the code changes [29]. Linares-Vásquez et al. have built ChangeScribe, which uses Change Distilling (a tree-differencing algorithm [50]) to extract changes between two Abstract Syntax Trees [103]. Like DeltaDoc, ChangeScribe also generates summaries based on predefined rules and templates.

The second group looks for the documents that are related to a commit. For example, Le et al. have built RCLinker, which links a bug report to the corresponding commit message [99]. Rastkar and Murphy have proposed to use an extractive summarization technique to summarize related documents [139]. Integrating the ideas of the first and the second groups, Moreno et al. have developed a tool that combines the two types of information into release notes [125].

In contrast to the two groups of techniques, my technique in Chapter 4 uses diffs (output text of differencing tools, such as “git diff”) as inputs. My project supplements the first group in two ways. First, the techniques in the first group often generate multi-line summaries that contain pseudocode and template text. In contrast, my technique generates one-sentence descriptions, which can be used as a headline of the multi-line summaries. Second, my technique summarizes both code and non-code changes in diffs.

2.5 Source Code Summarization Techniques

Source code summarization techniques generate descriptions of source code [126]. The algorithms of the techniques can be adapted to generate summaries for changes in commits. Code summarization can be categorized into two groups: extractive and abstractive. Extractive summarization extracts relevant parts of source code and uses the relevant parts as a summary [63]. Abstractive summarization includes
information that is not explicitly in the source code. For example, Sridhara et al. have designed a Natural Language Generation (NLG) system to create summaries of Java methods [158]. First, the NLG system finds important statements of a Java method. Second, the system uses a text generation algorithm to transform a statement to a natural language description. This algorithm has predefined text templates for different statement types, such as return statements and assignment statements. Both DeltaDoc and ChangeScribe (commit message generation techniques discussed in Section 2.4) follow the similar NLG design.

Besides the NLG approach to generate abstractive summaries, there is a deep learning approach. Iyer et al. have built Code-NN, which uses an Neural Machine Translation (NMT) algorithm to summarize code snippets [74]. This project is similar to my project because my project also uses an NMT algorithm. There are two differences between my technique and Code-NN. First, the goal of Code-NN is summarizing code snippets, and the goal of my technique is summarizing changes. Second, Code-NN parses code snippets and removes all the comments. In contrast, the input of my technique is a diff file containing code, comments, and diff marks (e.g., “+” denoting insertion).

2.6 Automatic Topic Labeling

There are three ways of topic model labeling in terms of the granularity of the labels. The first way of topic labeling is using keywords as labels. One strategy is extracting the top-n words (that have the highest probabilities in a topic model) as the label. This labeling method is originally used in LDA [20] and is commonly used in other areas [11, 36, 115]. A similar strategy is to extract the words whose probability is higher than a specific threshold (e.g., 0.1) [18], and then, reorder the words within a topic [149, 154]. However, the topics in the form of keyword lists lack
the readability. For example, in my study, a topic learned from Eclipse Jetty project is represented by a list of keywords “socket web websocket jetty eclipse factory upgrade api policy driver”. Although this example is a high-quality topic, readers still need background knowledge to interpret the list.

The second way is to label topics using phrases. Lau et al. have proposed an approach, which uses phrases from the titles of Wikipedia articles that have the top words of a topic [95]. Mei et al. have proposed to treat topic labeling as an optimization problem, where they collect candidate labels from a reference text collection, and using a scoring function to rank candidates [121].

The third way of topic model labeling is at sentence level, which labels a topic using a sentence. For example, McBurney et al. have tried multiple extractive approaches, which label a topic by extracting the sentences that are textually similar to the topic’s top words. They found that the existing extractive techniques are not adequate to generate meaningful labels [115]. Mei et al. also argue that sentences are too specific to be topic labels [121].

2.7 Topic Modeling in Software Engineering (SE)

Topic models have been widely adopted by the software engineering research community in the last decade for multiple tasks, such as source code analysis [18] [49] [56], bug report analysis [57] [75] [77], concept localization [14] [56], and software traceability [11] [54]. Topic models are widely used in software engineering domain because each software artifact file reflects several software features and contains the words that are highly related to the features. Each artifact file can be seen as a document in topic modeling, and the features can be mined as topics. Chen et al. surveyed the applications of topic models in the research domain of software
engineering [37] and found that LDA is the most used topic model.

When applying LDA in software engineering, the typical setting is to use a project as a corpus and a class file as a document. The tendency to run on the class-level documents (i.e., using classes as documents) rather than on the method-level documents (i.e., using methods as documents) is also confirmed by the study from the machine learning community [163], which points out that the topic models perform not well on short documents. In the existing literature, the number of topics varies from 10 [115] to 300 [57]. Grant et al. have shown that the optimal topic number is proportional to the number of code files [57].

2.8 Deep Learning in SE

Deep learning algorithms are becoming more prevalent in Software Engineering research. Deep learning algorithms, as applied to software, automatically learn representations of software artifacts. For example, to detect code clones, traditional approaches predefined the representations of code fragments (some techniques use token sequences to represent code [82]; others use graphs [35, 89]). In contrast, the deep learning approach introduced by White et al. learns the representations of code automatically [172]. Similarly, deep learning algorithms are introduced in bug localization [91], software traceability [61], and code suggestions [171].

Besides using deep learning algorithms on software projects, researchers have used deep learning algorithms on other artifacts that are relevant to software engineering, for example, recognizing user actions from tutorial videos [40] and linking related knowledge in developer forums [176].

2.9 Neural Machine Translation (NMT) in SE

As a prevalent deep learning technique, NMT has been adopted for software engineering tasks. There are two ways to use NMT in software engineering: (1)
translating natural languages to code and (2) translating code to natural languages. By translating natural language descriptions to programming languages, NMT is used for code generation [13, 104, 136]. Additionally, NMT is also trained to generate code-like sequences. For example, Gu et al. propose an NMT to generate API sequences from a natural language query [59].

By translating code to natural language descriptions, NMT is used in program comprehension tasks. Iyer et al. propose Code-NN, an neural attention model to summarize code [74]. Alexandru et al. use NMT to tokenize source code and to annotate source code tokens [8]. Jiang et al. create an NMT to generate commit messages [76]. Our NMT approach is also for software comprehension, but our approach is the first attempt to use NMT to tackle the topic labeling problem.
CHAPTER 3

STUDY OF CHANGE IMPACT ANALYSIS

This chapter represents work published at Empirical Software Engineering 2017 [78].

3.1 Introduction

Change impact analysis (IA) is the task of finding the consequences of an alteration to software [140 141]. From a programmers perspective, those consequences are typically the components of source code that would need to be modified in order to make a change. For example, if function $A$ in a program is modified, and function $B$ depends on function $A$, then function $B$ may also need to be modified.

IA is important because features in programs tend to be implemented across many components in source code [48 105 119], and a change to any one of the components may affect several of the others.

In theory, programmers will do change impact analysis prior to making any changes to the source code. Consider the example of repairing a bug. The consensus in the literature is that programmers 1) localize the bug behavior to a function or other set of statements, 2) determine a change to that code that will repair the bug, 3) do change impact analysis to determine the effects of that change, and 4) implement and test that change [26 80 96 141]. Several change impact analysis tools have been created based on this consensus, including static [26], dynamic [96], and conceptual [80]. While these tools have been shown to be effective, in practice no procedure has emerged as a standard accepted by a majority of programmers. This situation is akin to that pointed out for program comprehension tools by Roehm et
What Roehm et al. found was that program comprehension tool support was rich and growing, but that in practice programmers did not use these tools. The reality was that the tools assumed a different usage scenario than the programmers were actually following. Change impact analysis tools may be in a similar situation. Programmers may not do change impact analysis in the way that the literature suggests. For example, it is possible that programmers implement changes as quickly as possible, and repair negative effects of the change as they occur, skipping change impact analysis. Or, it is possible that programmers are simply unaware of the state-of-the-art techniques available.

In this chapter, I present an empirical study of change impact analysis by professional programmers. We conduct this study in two phases. In the first phase of our study, we recruited nine programmers to read two bug reports from two open-source systems, and repair the bugs by modifying the systems source code. We recorded videos of the repair processes. The objective was to observe the programmers behavior as they fixed the bug. We then analyzed this behavior to determine whether the programmers used any tools for change impact analysis, and if not, whether they performed change impact analysis through manual inspection.

In the second phase, we surveyed 35 programmers who had professional experience at different organizations. The intent of the survey was to determine the knowledge that the programmers had of change impact analysis, and to query the programmers about their current IA procedures. In addition to the knowledge questions via an online survey, we presented the programmers with an actual bug report from an open-source system, along with screenshots of an IDE. On the screenshots, the programmers selected the components of the IDE that they would use to repair the bug.

We designed three research objectives to answer the question whether programmers do change impact analysis. We will describe the research objectives in Sec-
From the research objectives, we designed several research questions in each phase. By answering the research questions, we found evidence of a gap between theory of change impact analysis and the practice. We came to this conclusion based on several findings, because the programmers did not know the term “Change Impact Analysis” and they did not use change impact analysis tools. Moreover, we found the programmers did static impact analysis before they made changes and dynamic impact analysis after they made changes.

3.2 The Problem

The problem we target in this chapter is a gap in the current literature regarding how programmers accomplish change impact analysis (IA) in practice. During code change activities (e.g., fixing a bug, adding a feature, etc.), it is generally assumed that programmers conduct IA prior to modifying source code. A rich and diverse set of tools has been created based on this assumption. However, there is little empirical evidence confirming this assumption. While IA procedures are recommended in textbooks and other educational resources, it is possible that programmers follow different strategies. For example, it is possible that programmers make changes first based on intuition, and test the consequences of those changes. This behavior would be in line with current literature surrounding “opportunistic” programming and similar concepts [23, 24, 45, 64], which are part of a broader consensus that programmers only try to comprehend the minimum amount of source code required to make a change [49, 90, 93, 144].

This chapter has a direct impact on the design of change impact analysis tools for code change activities, because it improves the understanding of how programmers will use those tools. For example, if a majority of programmers do not do IA prior to making changes, then tool support could potentially be directed to monitoring the effects of changes already made, rather than predicting effects from possible changes.
This potential impact is not unprecedented, as similar implications have been discovered in other areas of software engineering, e.g., program comprehension [144] and bug report authorship [73].

3.3 Background: Change Impact Analysis (IA)

Change impact analysis (IA) is a general process to identify elements that are indirectly or directly affected by a change [100, 141, 173]. IA has two types of application scenarios. The first application scenario is upfront. In requirement management, given a specific change request, IA is to identify the files and the models that may need to be modified for the requested change. A change request can be a bug report, a request of adding a feature, etc. The second application scenario is in source code change activities, such as debugging, refactoring, feature implementation, etc. When programmers debug, they may do IA to find out whether a change in code fix the bug. In refactoring, programmers may do IA to ensure that the changes will not have any effect on the outputs of the program.

The following subsections will describe three terms related to IA: IA process, IA techniques, and IA activities. We will also discuss the difference among the three terms in Section 3.3.4.

3.3.1 Classic IA Process

Petrenko and Rajlich [137] introduced an interactive IA process at the granularity of all components. This process can be used in both application scenarios (in requirement management and code change activities). Based on this interactive IA process, we illustrated the IA process at the granularity of statements in Figure 3.1. The objective of the IA process is to obtain a Changed Set, that is, the statements that need to be changed. Ideally, the Changed Set will always contain just the statements that must be changed. However, in practice, the Changed Set will not always be
To begin doing IA, the programmers must identify a location to change. An example of a location would be a statement of code. This statement is called an initial change location. First, the programmers put the initial change location into the Changed Set (Figure 3.1 Area a).

Second, the programmers follow different procedures to find code that is affected by the Changed Set (Figure 3.1 Area b). For example, they may follow the dependencies from the initial change location (we will list four possible strategies in Section 3.3.2). Then, the programmers add the affected code into the Possible Changed Set (Figure 3.1 Area b). Note that the Propagation Set in Figure 3.1 Area b will be defined later.

Third, the programmers look at every statement in the Possible Changed Set. For each statement, if the programmers think the statement needs to be updated,
they put this statement into the Changed Set (Figure 3.1 Area e). Otherwise, the programmers decide whether the statement propagates the effects of the change, such as changed values. If the statement propagates the effects to other statements, the programmers put it into the Propagation Set (Figure 3.1 Area f).

Fourth, with the new Propagation Set and the updated Changed Set, the programmers start at Figure 3.1 Area b again. The programmers stop the IA process when there are no more statements to be added into the Propagation Set and the Changed Set. The output of IA is the Changed Set, that is, the statements that need to be changed because of the initial change location.

3.3.2 IA Techniques

Although the IA process contains any phase described in Figure 3.1, current IA techniques focus on automating the task in Figure 3.1 Area b, which is finding the code that may be affected by a given statement. According to Li et al. [102], IA techniques can be broadly categorized as: 1) static dependency analysis, 2) dynamic execution information analysis, 3) software repository mining, 4) coupling measurement, or 5) combined approaches.

Static Dependency Analysis typically does reachability analysis on a graphical representation of a program derived from the source code [25], such as, call graphs, control flow graphs, or dependency graphs. Hattori et al. [65] analyzes reachability of a call graph, i.e. whether a method can be called by another method. If a method can be called by another method, the first method can be affected by the later one.

Dynamic Execution Information Analysis uses data generated from the execution of the programs, rather than only the source code [102]. There are different types of execution data [96, 131]. For example, Law and Rothermel use execution traces of function calls [96]. The intuition is that the function called first may affect the functions called later.
Software Repository Mining analyzes not only the source code and the execution data, but also the logs of software version control systems. Many kinds of logs can be used for IA [31, 182]. For example, Zimmermann et al. [182] apply data mining to changes made in the history to see what methods are usually changed together.

Coupling Measurement aims to calculate the degree of dependency between any two of program modules, such as methods. This degree of dependency indicates the possibility of one program module being affected by another module. The majority of coupling metrics are based on interactions between two modules. For example, Beszédes et al. [16] use metrics based on Execution-After relations [10]. Different from measuring the interaction between two modules, Poshvyanyk et al. [138] proposed conceptual coupling using information retrieval on identifiers and comments from code to measure relationship between two modules.

Combined Approaches integrate multiple kinds of analyses to gain better accuracy. For example, Breech et al. [25] combined static and dynamic approaches to obtain higher precision. Another example is that Gethers et al. [55] integrate dynamic analysis and repository mining, in order to choose an appropriate technique in a specific situation. Similarly, Kagdi et al. [80] integrate coupling measurement with software repository mining.

3.3.3 IA Activities

According to Figure 3.1 in the scope of this chapter, we define IA activities as the actions that programmers do to find and understand the code that may belong to the Possible Changed Set (Figure 3.1 Area b), the Propagation Set (Figure 3.1 Area f), or the Changed Set (Figure 3.1 Area e). By this definition, some examples of IA activities are file navigation, source code reading, and call graphs examining. In fact, any program comprehension activities are seen as IA activities in this chapter,
because program comprehension is one possible way to do change impact analysis. Note that we do not consider any program comprehension tool as an IA tool, because their purpose is program comprehension in general. Similarly, we do not consider IDE navigational functionalities as IA tools. However, they can be used in IA activities.

An extreme example of IA activities is running programs. By running a program and check its result, programmers may decide to continue the IA process or not. If the result is correct, some programmers may feel confident to commit the changes they made. If the result is incorrect, programmers know there is some code needs to be changed. If the value of the result is meaningful, programmers may locate the code to be changed by the incorrect value.

3.3.4 IA Process, IA Techniques, and IA Activities

In this section, we describe the differences among the IA process, IA techniques and IA activities. The IA process is a general procedure defined in academia that programmers do to find consequences of a source code change. We restrict our discussion of the consequences within the consequences in source code, i.e. the source code locations that need to be changed because of the initial change location. Note that IA process can be started with an initial change location with or without a specific change, which means, programmers can do IA before or after they make a change. We do not make any assumption about whether programmers do IA before they make a change.

IA techniques are designed specifically for a task in the IA process, which is Figure 3.1 Area b. There are IA techniques for post-change, which means that they assume programmers do IA after they make a change. There are also IA techniques for pre-change, which means that they can work without a specific change.

Any activity, whether they occur before or after programmers make a change, if the activity can help programmers in any phase in Figure 3.1, we consider they are
IA activities. IA activities are not necessarily related to IA techniques. IA activities can occur in various tasks, such as debugging, refactoring, reverse engineering, and effort estimation.

3.4 Background: Debugging

In this section, we describe a general debugging process based on the scientific debugging method introduced by Zeller [181]. The debugging process is shown in Figure 3.2. Given a bug report, first, programmers reproduce the bug and observe the failure. Then, based on the observation, programmers create a hypothesis about the cause of the failure. In the next step, programmers test the hypothesis. The methods of testing a hypothesis are various. Sometimes, programmers can print a value at a specific location to determine whether to reject the hypothesis or not. Often, programmers may make a change to the source code and observe whether the program runs as expected. If the observed value is not expected or the program gives unexpected result, it means the hypothesis is rejected. If the hypothesis is rejected, programmers need to make a new hypothesis about the failure cause. If the hypothesis is supported and the bug is fixed, programmers stop the debugging process. If the hypothesis is supported, but the bug is not fixed, programmers refine the hypothesis and test it again.

An example of a hypothesis about a failure cause is “If variable Xs value at line 10 is 1, the program should have a correct output.” Based on the hypothesis, programmers added a line at line 10 which assigns 1 to X. Then, programmers run the program to test the hypothesis. If the failure is gone, this test supports the hypothesis. If there is a failure, the hypothesis is rejected. The hypothesis being supported does not necessarily mean that the bug is fixed. For example, always assigning value one to X at line 10 may cause errors in other tests, and there may be a deeper cause for the faulty status of X.
Figure 3.2. The process of debugging according to the literature (see Section 3.4). This is the process that we expect our participants to follow in our studies.
The steps in the process of debugging may be skipped by some programmers. For example, programmers may skip reproducing the bug and create a hypothesis based on a bug report. Some hypotheses are not necessarily about what the failure cause is. They can be any hypothesis that is related to the program and help programmers narrow down the problems.

Note that fault localization falls into the step where programmers try to make hypotheses about the failure cause. In this case, the hypotheses are about the locations of faults. Also, regression testing can be in the beginning and in the end of a debugging process. Regression testing can reveal a bug, which leads to a bug report. The bug report becomes the beginning of a debugging process. Additionally, the results of regression testing may help programmers form hypotheses about the bugs. After programmers fix a bug, regression testing can help programmers ensure the quality of the fix, which occurs in the end of debugging.

Similarly, IA can occur in different phases in the debugging process. Programmers can do IA during fault localization. If a source code location affects the result, programmers may create a hypothesis that this location is a faulty location. Programmers may do IA when they test hypotheses. For example, in order to test the hypothesis source code line 10 should not affect the output of the program, programmers need to investigate the effects of any change in line 10, which is considered as an IA process.

3.5 Empirical Study Design Overview

The objective of this chapter is to begin to answer the question *do programmers do change impact analysis?* In particular, we want to study whether programmers do IA when they do code change activities. Towards the problem we stated in Section 3.2, we designed three research objectives.

**Objective 1** What knowledge do programmers have of research activities in IA?
Objective 2 What technologies do programmers use to do IA if they do any?

Objective 3 At what phases of debugging, do programmers do IA if they do any?

The first objective helps us to answer whether there is a gap between research and industry. In studying the second object, if we find technologies used for IA, it is not only the evidence of programmers doing IA, it also provides information about the IA process in practice. In studying the third object, if we find evidence that they do IA before they make changes, this will be the empirical evidence for the assumption of various IA tools.

We target the three research objectives with a “depth” study and a “breadth” study of programmer behavior. In the “depth” study we recorded the behavior of nine programmers repairing actual bugs in source code. The “breadth” study is a survey of 35 programmers. We designed four research questions in the in-depth study and five research questions in the breadth study. We illustrate the overview of the objectives and research questions in Figure 3.3

3.6 In-Depth Study Design

This section will describe our in-depth study, in which we hired programmers to solve actual bugs in software. Following, we will cover our research questions, methodology, subject applications, participants, and threats to validity.

3.6.1 Research Questions

In this section, we have specified research questions to study the specific measurable behaviors related to the research objectives. We pose the following Research Questions (RQs) towards our objective of determining whether programmers do IA. We formulate these RQs with the idea of recording the actual behavior of programmers during debugging.
Figure 3.3. All the research questions are introduced in Sections 3.6.1 and 3.9.1. In the in-depth study, $RQ_1$ helps to answer Objective1 and Objective2. $RQ_2$, $RQ_3$, and $RQ_4$ helps to answer Objective3. In the breadth study, $RQ_5$ and $RQ_6$ helps to answer Objective1. $RQ_7$, $RQ_8$, and $RQ_9$ helps to answer Objective2.

$RQ_3$: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they read a change location and the first time they alter the change location?

$RQ_4$: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they alter code and the first time they run the altered code?
RQ₁ Do programmers use any IA tools?

RQ₂ Do programmers navigate to dependents or dependencies of the first section of code they read?

RQ₃ How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities to navigate dependencies between the first time they read a change location and the first time they alter the change location?

RQ₄ How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities to navigate dependencies between the first time they alter code and the first time they run the altered code?

RQ₁ is for Objective₁ and Objective₂. If programmers use IA tools, this indicates that programmers know the concept of IA. Using tool or not is an evidence of how programmers do IA. The rationale behind RQ₂ is that the first section of code the programmers read is the beginning of the debugging process. If a programmer reads code along the dependencies of that code, we think this indicates programmers doing IA in the beginning of debugging. So RQ₂ helps to answer Objective₃. Similarly, RQ₃ and RQ₄ are for Objective₃, too. The more activities programmers do before or after they make a change, the stronger is the evidence of programmers doing IA during the corresponding period.

3.6.2 Methodology

We followed an observational study methodology [9], under guidelines for case studies recommended by Runeson et al. [147] and Robillard et al. [142]. Our overall procedure is as follows.

1. Identify bug reports. We identified two bug reports in two different open source programs.

2. Set up development environment in a virtual machine (VM). For each bug report, we built a VM with Ubuntu 14.04 as the guest operating system. Inside the VM, we installed Eclipse and set up the open source program inside Eclipse. We set up the development environment inside a VM so that we can send the environment to the participants who cannot be in our lab.
3. **Install recording software in the VM.** We installed SimpleScreenRecorder\(^1\) inside the VMs to record videos of the screens.

4. **Conduct study with participants.** We gave each participant the two bug reports and asked them to fix the bugs. In total, we obtained two videos from each participant. In this study, we do not introduce IA concept to the participants to minimize the bias. There are ten participants. For seven participants, we hired them for one hour per bug. If they are not able to fix one bug in one hour, they can give up or continue at will. For the remaining three participants, we hired them to fix the bugs, so there is no time limitation. To minimize the bias of our working setting, we explicitly asked all the ten participants to install any plugin or tools that they used in their normal working environment.

5. **Complete study.** We repeated step 4 for ten participants. For four participants that did experiments in our lab, one author was present during the study to assist with technical problems, but that author did not communicate with the participants about the bug or code. The author sat away from the participants and did not watch them, to avoid observer bias. For the other participants who did the experiments remotely, the author was available online for potential technical problems.

3.6.3 Subject Applications

The two subject applications we used in our study are “PDF Split and Merge” (PdfSam\(^2\) and Raptor\(^3\). In each Java program, we chose one bug report for our study. The sizes of the programs are listed in Table 3.1. We chose these projects because they are real, of non-trivial size, and the purposes of the projects are clear and easy to understand. We chose the two bug reports because they were fixed and can be easily understood by the programmers who do not know the programs.

With each bug report, we presented to the participants some information to let them know the program and the bug faster and easier. The information includes the purpose of the program, the entry point of the program, an input of the program that reveals the bug. For the seven participants who only have one hour, we also provided

\(^1\)http://www.maartenbaert.be/simplescreenrecorder/
\(^2\)http://www.pdfsam.org/
\(^3\)https://code.google.com/p/raptor-chess-interface/
a suggested fix because of two reasons. One reason is that it took us about 2 hours to understand the bug and locate related source code. We do not want participants spend all the time in understanding the application and have not time changing anything in one hour. The second reason is that when programmers do their daily job, they often have some ideas of where might be the problem. Therefore, giving a suggested fix matches this scenario. Note that we did not provide the suggested fix for the other three participants who were required to fix the bugs no matter how much time they spend.

3.6.3.1 PdfSam and the Subject Bug

PdfSam is a PDF file editor. The bug we chose (id: 100) was reported in June 2014. A user reported that PdfSam failed to rotate a PDF file. The actual fix is shown in Figures 3.4 and 3.5. The cause of the bug was that the developer used RandomAccessFileOrArray to load pdf files into memory. In this API function, some information about pdf file is lost. This lost information caused the rotation not being applied to a new pdf file. RandomAccessFileOrArray is called in method readerFor. To fix the bug, the developer added another method

<table>
<thead>
<tr>
<th>Methods</th>
<th>KLOC</th>
<th>Java Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>PdfSam</td>
<td>2686</td>
<td>31.2</td>
</tr>
<tr>
<td>Raptor</td>
<td>1136</td>
<td>13.1</td>
</tr>
</tbody>
</table>

a KLOC: K lines of code reported with all comments removed.
called `fullReaderFor`. The `fullReaderFor` is the same with `readerFor` except `fullReaderFor` calls `FileInputStream` instead of `RandomAccessFileOrArray`. Then, inside the method that handles rotation, the developer replaced `readerFor` with `fullReaderFor`. The fix will only affect `PdfSam` when the pdf files are requested to be rotated. Our suggested fix is directly replacing `RandomAccessFileOrArray` with `FileInputStream` inside method `readerFor`. The suggested fix potentially affects all the methods that calls `readerFor`. Additionally, the participants do not know whether the suggested fix works.

3.6.3.2 Raptor and the Subject Bug

`Raptor` is a graphical user interface for a Chess Server (`FICS`). The bug report we used (id: 1) was reported by a developer of this program in September 2009. The developer reported that the clocks did not tick down when they should. The actual fix is shown in Figures 3.6 and 3.7. The reason for the bug is that the states of the game were not correctly updated. The cause is in the method `updateNonPositionFields`, which is supposed to update the states of the game. The problem is that the game’s state is not updated with `IS_CLOCK_TICKING_STATE` when the game is at `EXAMINING_STATE`.

To fix this bug, a developer added a patch in the method `updateNonPositionFields`. By adding an `if` statement in `updateNonPositionFields`, `Raptor` is forced to updated the `IS_CLOCK_TICKING_STATE` every time `updateNonPositionFields` is called. The suggested fix we provided is a change that committed in the repository with this fix. The change is related to updating states at `EXAMINING_STATE`, but the change does not fix the bug. The change is in Class `ExamineController`, which is the chess board controller when the game is in `EXAMINING_STATE`.

public class RotateCmdExecutor{
  ...
  public void execute(AbstractParsedCommand parsedCommand){
    pdfReader = PdfUtility.readerFor(fileList[i]);
  }
}

public final class PdfUtility {
  ...
  public static PdfReader readerFor(PdfFile file){
    PdfReader reader = new PdfReader(
      new RandomAccessFileOrArray(file.getFile().getAbsolutePath()), …);
    unethical(reader);
    return reader;}
}

Figure 3.4. The original code before the fix in PdfSam.

public class RotateCmdExecutor{
  ...
  public void execute(AbstractParsedCommand parsedCommand){
    pdfReader = PdfUtility.fullReaderFor(fileList[i]);
  }
}

public final class PdfUtility {
  ...
  public static PdfReader readerFor(PdfFile file){
    PdfReader reader = new PdfReader(
      new RandomAccessFileOrArray(file.getFile().getAbsolutePath()), …);
    unethical(reader);
    return reader;}

  public static PdfReader fullReaderFor(PdfFile file) ... { 
    PdfReader reader = new PdfReader(new FileInputStream(file.getFile()), …);
    unethical(reader);
    return reader;}

  private static void unethical(PdfReader reader) ... { 
    [code block 1]}
}

Figure 3.5. The code after the actual fix in PdfSam.
public class FicsUtils implements GameConstants {
  public static void updateNonPositionFields(Game game, Style12Message message) {
    switch (message.relation) {
      case Style12Message.EXAMINING_GAME_RELATION:
        game.setState(Game.EXAMINING_STATE);
        break;
      case ...
        break;
    }
    if (message.isClockTicking) {
      game.addState(Game.IS_CLOCK_TICKING_STATE);
    } else {
      game.clearState(Game.IS_CLOCK_TICKING_STATE);
    }
  }
}

Figure 3.6. The original code before the fix in Raptor.

Figure 3.7. The code after the actual fix in Raptor.
3.6.4 Participants

We recruited seven programmers for one hour per bug in our study. One participant was not familiar with Eclipse, so we discarded the videos of the participant. The remaining six participants are listed as Participants 1 to 6 in Table 3.2. Additionally, we hired another three programmers to fix the bugs with no time limit and no suggested fixes, who are listed as Participants 7 to 9 in Table 3.2. Most of the participants are experienced programmers. The average experience level of the nine programmers is 10 years on paid positions. Due to time and resource limits, we decided not to hire more programmers.

3.6.5 Threats to Validity

Like any study, our study has threats to validity. First, the two programs in the study may not be representative. We mitigate this threat by choosing two types of programs. One program is an online application, and the other is offline. Therefore, our results are not limited in one program type. Second, the two bug reports may not be representative. We mitigate this threat by choosing real bugs from the real repositories, so that the bugs are at least realistic and may happen in practice. Third, we chose Eclipse as the working environment, which may affect the behavior of the programmers in debugging. However, we think Eclipse is very widely used, and can represent general IDEs.

Fourth, the participants are not the developers of the selected programs. Therefore, their behavior may differ from the real developers of the applications. We mitigate this threat by offering some information about the programs and the bugs. Although the participants have some information about the application and the bug, the participants still may do more activities for program comprehension than the real developers of the applications. So there may be more IA activities done in the study than in the real world.
<table>
<thead>
<tr>
<th>Participant</th>
<th>Java Experience</th>
<th>Years Exper.</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant 1</td>
<td>2</td>
<td>4</td>
<td>Ph.D. program University of Notre Dame</td>
</tr>
<tr>
<td>Participant 2</td>
<td>0 (5 years in OO)</td>
<td>5</td>
<td>Computing resource center University of Notre Dame</td>
</tr>
<tr>
<td>Participant 3</td>
<td>12+</td>
<td>12+</td>
<td>withheld for privacy</td>
</tr>
<tr>
<td>Participant 4</td>
<td>10</td>
<td>5</td>
<td>Ph.D. program University of Notre Dame</td>
</tr>
<tr>
<td>Participant 5</td>
<td>15</td>
<td>30</td>
<td>withheld for privacy</td>
</tr>
<tr>
<td>Participant 6</td>
<td>7</td>
<td>9</td>
<td>Ph.D. program Peking University</td>
</tr>
<tr>
<td>Participant 7</td>
<td>5</td>
<td>4</td>
<td>A Bank IT department</td>
</tr>
<tr>
<td>Participant 8</td>
<td>5</td>
<td>5</td>
<td>withheld for privacy</td>
</tr>
<tr>
<td>Participant 9</td>
<td>12</td>
<td>12</td>
<td>A financial service software company</td>
</tr>
</tbody>
</table>
3.6.6 Reproducibility

To ensure reproducibility by independent researchers, we put all the data via an online appendix, including the VM images, the videos, the logged activity sequences, the recognized change locations, the scripts for data analysis, and the results: http://nd.edu/~sjiang1/IA-study.htm

3.7 In-Depth Study Data Collection

Before we processed the videos, we discarded the two videos of one participant because he was not familiar with Eclipse. Then, we processed the remaining 18 videos as follows.

3.7.1 Recorded activities

We watched the videos and recorded all activities with time-stamps, such as “open a file”, “edit a line of source code”, and other activities in IDE. Additionally, if an activity modifies source code in any way, we also recorded the file name and the line number that the activity modifies. For a complete list of the activities, please refer to the online appendix in Section 3.6.6. All the activities we recorded in a video formed one sequence of activities. For 18 videos, we obtained 18 sequences.

3.7.2 Found change locations

We found the change locations in the sequences. To find change locations, we first found all the activities that modify source code in the sequences. We removed the modifications that do not have any impact. These modification includes: adding/deleting comments, formatting (adding/deleting spaces and lines), modifications that were undid immediately (for example, in the middle of the modification, the programmer realized that this modification would not work and undid the whole
modification.) It is possible that there are some changes that may seem to have impacts, but actually they do not have any impact. We took these changes into account, because without inspection, programmers do not know that the changes have no impact either.

Besides the changes that have no impact, we also excluded a special case of the modifications, which is undo. Often the programmers undid changes they made because of the two reasons: 1) they ran the program and the change did not have expected impact on the output of the program, and 2) the changes are made accidentally. We ignored these “undo” changes because their impacts are discarding the impacts of the changes, which programmers may know about.

There are another two special changes that we did not take into account. One programmer created a new test project. The other programmer created a new test class. Because our scope of the changes is limited to the changes to the existing code, we excluded these two changes in our results.

From the remaining modifications, we collected the file names and the line numbers. If some lines of code are successive in the same file, we grouped these lines into one “change location”. Even the lines of code that were modified at different times, we grouped them into one “change location” because these changes are related to each other. In one video, a programmer may have multiple change locations.

3.7.3 Found important times

First, we logged the first times programmers read a change location ($FReT$). Then, we logged the times when each change location was edited. Programmers may modify the location multiple times and the number of modifications varies. For our research questions, we only logged the first modification time ($FMT$) and the last modification time ($LMT$). Furthermore, for each modification time, we logged the first time that the program was executed after that modification ($FRT$ and $LRT$).
In summary, Table 3.3 lists all the time labels we marked.

3.7.4 Measured distances

We measured elapsed time in seconds, the number of files visited, and the number of times that Eclipse functionality is used between three periods of time, which are from $FReT$ to $FMT$, $FMT$ to $FRT$, and $LMT$ to $LRT$. We counted all the IDE functionalities that help programmers navigate dependencies of a code element, including “open call hierarchy of” a method, “open declaration of” a class, and so on. We listed all the navigational functionalities in Table 3.4. Note that although “search” is listed in Table 3.4, “search” is not counted as the IDE functionality that help programmers navigate dependencies, so we did not count “search” in answering our research questions.

3.8 In-Depth Study Results

In this section, we will describe the quality of the patches that our participants created. Then, we will begin to answer the question do programmers do change impact analysis? From a high level, we found the evidence that the programmers did IA but did not use IA tools: 1) the programmers did not use IA tools; 2) the programmers did IA before they made changes; 2) the programmers ran the programs after they made changes. In the rest of the sections, we will explain the statistical details of how we came to this conclusion.

3.8.1 Quality of Patches

For PdfSam, Participants 1 to 6 applied the exact patch of the suggested fix to the program. Participant 7 figured out a way to fix the problem in a different method. Participant 8 made a similar patch to the actual fix, so this patch has less impact than the other patches. Participant 9 applied the exact patch of the suggested fix.


TABLE 3.3

THE TIME LABELS

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FReT$</td>
<td>The first time that a programmer reads a change location.</td>
</tr>
<tr>
<td>$FMT$</td>
<td>The first time that a programmer modifies a change location.</td>
</tr>
<tr>
<td>$FRT$</td>
<td>The first time that a programmer runs the program after $FMT$.</td>
</tr>
<tr>
<td>$LMT$</td>
<td>The last time that a programmer modifies a change location.</td>
</tr>
<tr>
<td>$LRT$</td>
<td>The first time that a programmer runs the program after $LMT$.</td>
</tr>
</tbody>
</table>

* It is impossible to know the change location before the location is altered. So we found $FMT$ first, then we found $FReT$. 
<table>
<thead>
<tr>
<th>Category Name</th>
<th>Eclipse Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call hierarchy</td>
<td>open call hierarchy</td>
<td>show the callers of a method</td>
</tr>
<tr>
<td>Type hierarchy</td>
<td>open type hierarchy</td>
<td>show the supertype/sub-type of a class</td>
</tr>
<tr>
<td>Declaration</td>
<td>open declaration</td>
<td>open the definition of a class/method/variable</td>
</tr>
<tr>
<td>Implementation</td>
<td>open implementation</td>
<td>open the implementation of a method</td>
</tr>
<tr>
<td></td>
<td>open super implementation</td>
<td>open the super implementation of a method</td>
</tr>
<tr>
<td>References</td>
<td>find references</td>
<td>show all the references in Workspace/Project</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of a class/method/variable.</td>
</tr>
<tr>
<td>Search</td>
<td>text search</td>
<td>search the exact text in Workspace/Project</td>
</tr>
<tr>
<td></td>
<td>java search</td>
<td>search all the occurrences of a java element in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Workspace/Project</td>
</tr>
<tr>
<td></td>
<td>text find</td>
<td>find the exact text in the current file</td>
</tr>
</tbody>
</table>
For Raptor, Participant 1 to 6 did not fix the problem. However, Participants 1, 2, 3, and 6 made a major progress where the clocks began to tick down but at a wrong time. Participants 7 and 8 made similar patches to the actual fix. Participant 9 also made a similar patch, but s/he made sure the new code is only accessed when the game is under `EXAMINING_STATE`. So the last patch has less impact than the other patches.

3.8.2 Example Result

This section will explain the result for a change location in PdfSam as an example, see Table 3.6. The change location is at line 94-95 in the file `PdfUtility.java`.

From Table 3.6, the participant read only one file between the first time s/he read the change location and the first time s/he altered the change location (“FReT to FMT” in Table 3.6). This file is the file containing the change location. This indicates s/he did not do IA across different files before s/he made the change. Additionally, there is no Eclipse functionality used to navigate source code between FReT and FMT, which further indicates the programmer did not do much IA in this period within the file. For FMT to FRT and LMT to LRT, the numbers are same, which indicates the programmer did not do much IA before s/he ran the program.

3.8.3 Aggregate Result

From the 18 videos, we found 31 change locations. Table 3.7 listed the total time for each video, the number of visited files, and the number of the times that IDE functionalities were used. The average distances of the 31 changes in three time periods are presented in Table 3.8. We also presented the average results for each bug in Tables 3.9 and 3.10.

A big difference between Tables 3.9 and 3.10 is that the participants without the suggested fixes spent more time in FReT and FMT in PdfSam than in Raptor.
<table>
<thead>
<tr>
<th>Participant</th>
<th>PdfSam</th>
<th>Raptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant 1</td>
<td>fixed</td>
<td>not fixed with a major progress</td>
</tr>
<tr>
<td>Participant 2</td>
<td>fixed</td>
<td>not fixed with a major progress</td>
</tr>
<tr>
<td>Participant 3</td>
<td>fixed</td>
<td>not fixed with a major progress</td>
</tr>
<tr>
<td>Participant 4</td>
<td>fixed</td>
<td>not fixed</td>
</tr>
<tr>
<td>Participant 5</td>
<td>fixed</td>
<td>not fixed</td>
</tr>
<tr>
<td>Participant 6</td>
<td>fixed</td>
<td>not fixed with a major progress</td>
</tr>
<tr>
<td>Participant 7</td>
<td>fixed</td>
<td>fixed</td>
</tr>
<tr>
<td>Participant 8</td>
<td>fixed + refactored</td>
<td>fixed</td>
</tr>
<tr>
<td>Participant 9</td>
<td>fixed</td>
<td>fixed</td>
</tr>
</tbody>
</table>
TABLE 3.6

THE MEASURED DISTANCES FOR ONE CHANGE LOCATION

<table>
<thead>
<tr>
<th></th>
<th>$FReT$ to $FMT$</th>
<th>$FMT$ to $FRT$</th>
<th>$LMT$ to $LRT$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in seconds</td>
<td>4</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td># of Files</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># of Dep. Navs.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* This change location is in PdfSam. See Section 3.8.2 for the description of this example.
However, in the same period of time, the number of files visited and the number of uses of functionalities are similar in PdfSam and Raptor. This indicates that the participants without the suggested fixes read similar amount of code before they made the changes, but they spent more time in reading the code in PdfSam.

For Figures 3.8, 3.9 and 3.10, each figure corresponds to a metric we measured, i.e., time length in seconds, the number of files visited, and the number of IDE functionality uses. There are three boxplots in each figure. Each boxplot represent a time period, that is \( FRcT \) to \( FMT \), \( FMT \) to \( FRT \) and \( LMT \) to \( LRT \). In general, we can see the programmers put the most effort before they make any modification for a change location, and they put the least effort after they make changes and before they run the programs.
<table>
<thead>
<tr>
<th>Participant Id</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdfsam Time in Sec.</td>
<td>7800</td>
<td>3051</td>
<td>3314</td>
<td>453</td>
<td>3544</td>
<td>1555</td>
<td>4295</td>
<td>13367</td>
<td>6168</td>
</tr>
<tr>
<td># of Files</td>
<td>33</td>
<td>15</td>
<td>6</td>
<td>2</td>
<td>13</td>
<td>9</td>
<td>12</td>
<td>34</td>
<td>24</td>
</tr>
<tr>
<td># of Func.</td>
<td>15</td>
<td>21</td>
<td>4</td>
<td>0</td>
<td>36</td>
<td>21</td>
<td>38</td>
<td>46</td>
<td>37</td>
</tr>
<tr>
<td>Raptor Time in Sec.</td>
<td>3119</td>
<td>4537</td>
<td>3352</td>
<td>3543</td>
<td>4941</td>
<td>4081</td>
<td>2809</td>
<td>6546</td>
<td>5343</td>
</tr>
<tr>
<td># of Files</td>
<td>15</td>
<td>24</td>
<td>6</td>
<td>6</td>
<td>23</td>
<td>9</td>
<td>15</td>
<td>21</td>
<td>24</td>
</tr>
<tr>
<td># of Func.</td>
<td>18</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>60</td>
<td>23</td>
<td>34</td>
<td>33</td>
</tr>
</tbody>
</table>
### TABLE 3.8

THE AVERAGE DISTANCES FOR THE 31 CHANGES

<table>
<thead>
<tr>
<th>Time in Sec.</th>
<th>FRcT to FMT</th>
<th>FMT to FRT</th>
<th>LMT to LRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Suggested Fix</td>
<td>472 (137)*</td>
<td>119 (48)</td>
<td>29 (17)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>1729 (1247)</td>
<td>53 (46)</td>
<td>19 (19)</td>
</tr>
<tr>
<td>All</td>
<td>877 (416)</td>
<td>97 (46)</td>
<td>27 (17)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Files</th>
<th>FRcT to FMT</th>
<th>FMT to FRT</th>
<th>LMT to LRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Suggested Fix</td>
<td>4 (3)</td>
<td>2 (2)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>10 (9)</td>
<td>1 (1)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>All</td>
<td>6 (4)</td>
<td>2 (1)</td>
<td>2 (2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of Func.</th>
<th>FRcT to FMT</th>
<th>FMT to FRT</th>
<th>LMT to LRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Suggested Fix</td>
<td>4 (0)</td>
<td>2 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>11 (9)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>All</td>
<td>6 (1)</td>
<td>1 (0)</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

* The median distances are in parentheses.
<table>
<thead>
<tr>
<th></th>
<th>FReT to FMT</th>
<th>FMT to FRT</th>
<th>LMT to LRT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time in Sec.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Suggested Fix</td>
<td>316 (226)*</td>
<td>88 (56)</td>
<td>40 (20)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>1979 (1633)</td>
<td>39 (40)</td>
<td>19 (19)</td>
</tr>
<tr>
<td>All</td>
<td>903 (416)</td>
<td>71 (40)</td>
<td>31 (20)</td>
</tr>
<tr>
<td><strong># of Files</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Suggested Fix</td>
<td>4 (4)</td>
<td>2 (2)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>10 (7)</td>
<td>1 (1)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>All</td>
<td>6 (4)</td>
<td>2 (1)</td>
<td>2 (2)</td>
</tr>
<tr>
<td><strong># of Func.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Suggested Fix</td>
<td>2 (1)</td>
<td>1 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>11 (7)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>All</td>
<td>5 (1)</td>
<td>1 (0)</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

* The median distances are in parentheses.
### TABLE 3.10

**THE AVERAGE DISTANCES FOR THE 14 CHANGES FOR RAPTOR**

<table>
<thead>
<tr>
<th>Time in Sec.</th>
<th>( FR_{eT} ) to ( FMT )</th>
<th>( FMT ) to ( FRT )</th>
<th>( LMT ) to ( LRT )</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Suggested Fix</td>
<td>643 (133)*</td>
<td>156 (40)</td>
<td>21 (9)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>1354 (1234)</td>
<td>74 (67)</td>
<td>n/a</td>
</tr>
<tr>
<td>All</td>
<td>846 (275)</td>
<td>131 (64)</td>
<td>21 (9)</td>
</tr>
<tr>
<td>With Suggested Fix</td>
<td>5 (1)</td>
<td>2 (1)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>9 (9)</td>
<td>1 (1)</td>
<td>n/a</td>
</tr>
<tr>
<td>All</td>
<td>6 (3)</td>
<td>2 (1)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>With Suggested Fix</td>
<td>7 (0)</td>
<td>2 (0)</td>
<td>0 (0)</td>
</tr>
<tr>
<td>No Suggested Fix</td>
<td>10 (10)</td>
<td>0 (0)</td>
<td>n/a</td>
</tr>
<tr>
<td>All</td>
<td>8 (1)</td>
<td>2 (0)</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

*The median distances are in parentheses.*
3.8.4 Difference between the study with the suggested fixes and the study without
the suggested fixes

In Table 3.5, Participants 1 to 6 fixed the PdfSam’s bug by using our suggested fix. But for Raptor, all the participants fail to fix the bug. This indicates that the quality of the suggested fixes affects whether the participants could fix the bugs. If the suggested fix is similar to the actual fix, the participants can easily fix the bug without extra effort to refactor the program for better code quality. If the suggested fix is not in the code where the actual fix is, the participants cannot fix the bug in a limited time.

Participants 7, 8 and 9 fixed both the bugs in PdfSam and Raptor, because they were asked to fix the bugs to finish the task without a time limit. Note that only Participant 8 refactored PdfSam’s code like the actual fix in the repository.

In Tables 3.8, 3.9, and 3.10, the biggest difference between Participants 1-6 and Participants 7-9 is during FReT to FMT. Participants 7-9 spent more time, visited more files and used Eclipse functionality more times before they modified locations. The difference is understandable because the participants have less information (no suggested fixes) than Participants 1 to 6.

3.8.5 RQ1: Do programmers use any IA tools?

No programmer used any IA tools except code navigation functionalities in Eclipse. Note that we explicitly told participants that they can install any plugin. Our interpretation is that the programmers tend not to use IA tools. One possible reason is that for fine-grained IA, such as statement-level IA, programmers often do it by comprehending code themselves. For coarse-level IA, such as method-level IA, the code navigation functionalities, such as “open call hierarchy”, are enough. Not using IA tools indicate that programmers are not familiar with the techniques proposed for IA tasks.
Figure 3.8. The distribution of the time lengths

Figure 3.9. The distribution of the number of visited files
3.8.6 RQ₂: Do programmers navigate to the dependencies or the dependents of the first section of code they read?

In the 11 out of 18 debugging sessions, the programmers in our study did not navigate to dependents and dependencies of the first section of code that they read. Therefore, we think the programmers do not prioritize IA at the beginning of the debugging process, which should be fault localization. In 11 of the 18 videos, the programmers jumped from the first file they read to the second file.

In the remaining seven videos, we found evidence that the programmers do IA. Most of them do dynamic IA by following the execution. They ran the program with a debugger step by step. Two of them did static IA by using “open declaration of” a method call in the first file.
3.8.7 RQ₃: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of the first read and the first modification?

Between the first read (FReT) and the first modification (FMT) of a change location, the programmers had more code navigations than they did in the other periods, see Table 3.8. We think this indicates programmers doing IA in a static way. The main task in this period is to decide whether or how the change location should be changed. To do this task, the programmers need to understand what the change location affects.

When we did not provide the suggested fixes, the programmers visited an average of ten files. However, in the videos that we provided the suggested fixes, the number of visited files in this period is small. On average, the programmers only navigated less than four files. By comparing these two results, our interpretation is that the programmers tend to try changing the code as soon as possible with the minimum amount of program comprehension.

In Figure 3.9, most of the programmers read fewer than twelve files. Our interpretation is that reading twelve files should be sufficient for programmers figure out an initial change for a location. The outliers at FReT to FMT in Figures 3.8, 3.9 and 3.10 correspond to four change locations.

For two change locations, the programmers altered other change locations first after they read the change location. For another outlier, the programmer comprehended the entire program before she made any modification. The activities between the first read and the first modification include the tasks to comprehend the overall structure of the program. These two outliers are similar to the cases of inattentioanal blindness reported by Robillard et al. [142], which is the situation where programmers do not intend to change a location when they read the location at the first time.

For the fourth outlier, we believe this is the case where the programmer did
extensive IA (18 files visited, 22 times of using IDE functionalities) after s/he locate this line and before s/he make a change to the location.

3.8.8 RQ4: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of modification and the first run of the altered code?

The programmers in our study almost always ran the programs immediately after they made the last modification of a change location. In 75% of the cases, the programmers navigated to no more than two files (including the files that contain the change locations) in Figure ?? during both LMT to LRT and FMT to FRT.

The numbers of uses of the Eclipse navigational functionalities are similar in the two periods, too. FMT to FRT on average have one navigational functionality used. And in LMT to LRT, there is almost no functionality used. Our interpretation is that the programmers tend to run programs immediately once they made the changes.

The average time between FMT and FRT is overall 97 seconds. Because the time we recorded for modifications is the beginning time of the modifications, the period between FMT and FRT includes the time that the programmers spent on actually changing code.

The outliers at FMT to FRT in Figures 3.8 and 3.9 correspond to two change locations. For one change location, the first edit of the location is not intended to change the program, but to help the programmer to “open declaration of” a method. The call of the method was commented out, in order to use Eclipse navigational functionality, the programmer uncommented this code, so that s/he could click to the declaration of the method. For the other outlier, the programmer navigated to other places and inserted some log function calls, so that when the program runs, it would output more useful information. This case shows that the programmer did IA
after s/he made the change.

3.8.9 Qualitative Results

For the last three participants that we hired to fix the bugs, we also did interviews after they finished their jobs. The interviews were conducted in an online chatting tool, which is provided by the online hiring platform we used.

3.8.9.1 knowledge of IA

One participant reported that he knew the term, “but have not seen anything related to that”. In his understanding, he thought IA is the question about “what would be the impact of some change that I want to make”. The other two participants do not know IA, but said they did IA after we introduced IA to them. IA is introduced as “Change-Impact Analysis is a task of finding the source code which is affected by the source-code change that you are going to make.”

When the programmers were asked about IA in their daily work, they often refers to post-change IA. “I always try to understand how I can influence the code. If I’m uncertain about my changes I can make a list of influenced part and give it to our QA[Quality Assurance] engineers. They are checking all cases.” (Participant 8) “I have to make sure that my change will not cause bugs or other problems for other parts of the project or system’s components ... ” (Participant 9) “... I get that a lot in my work - as the systems quite often use global variables, that are a mess to track.” (Participant 7)

3.8.9.2 practice of IA

From what the participants said, the need of doing IA is different for different fixes. “In Raptor project - not much, because the fix seemed to be fairly isolated. In PDF project - yeah - as I still had some doubts about what’s the lifecycle of PdfDictionary
and what uses it” (Participant 7) “in pdfsam I understood that changing Pdutils
class can lead to bugs in other ways of manipulating pdfs. .. And I’ve tried to fix
the bug not introducing any changes in other components ... In raptor it seems clear
that the input params are not processed correctly. I’ve checked places where this
state of game is used and it seems that my changes can’t influence the other code.”
(Participant 8)

The participants also mentioned to do IA by exploring the methods being called.
“Example: I change a method of a class. After my changes I have to find direct
and indirect calls of this method and make sure that system will be ok after my
changes.” (Participant 9) Note that the programmers also mentioned to rely on
quality assurance team to do IA. “If I'm uncertain about my changes I can make a
list of influenced part and give it to our QA engineers. They are checking all cases.”
(Participant 8)

3.8.10 Comparison to Other Results of Code Changing Tasks

Many research projects were conducted on the topic of code changing [86, 91, 155].
Sillito et al. [155] studies three research questions: 1) What knowledge programmers
need when they change code? 2) How programmers get the relevant information? 3)
How well the existing tools help programmers get the knowledge? They observed 27
sessions of programmers doing code changing tasks. They identified four categories
of questions that programmers asked when they did the tasks. In the four categories,
there are two categories that are related to our results.

The first category is “finding focus points.” In this category, Sillito et al. noted
five questions, like “Which type represents this domain concept or this UI element or
action?” [155] In our in-depth study specifically, the participants asked the question:
“where is the code that represents this action?” Particularly for PdfSam, all the
three participants in the interview said they wanted to locate the code that rotating
the pdf file. “The first step is select project modules which may contain the bug. E.g. for PdfSam project I looked for the rotation related modules.” (Participant 7)
“I knew from description that rotation failed and I was finding places where rotation is using.” (Participant 8) “... I tried to find the code that handles rotation first” (Participant 9)

The second category is “expanding focus points.” This category has 15 questions, such as “Where is this method called or type referenced?” and “What does the declaration or definition of this look like?” In the videos, we recorded the uses of Eclipse navigational functionalities. To compare our results to Sillito et al.’s report, we generated a snapshot of the uses of Eclipse navigation functionalities. We listed the navigational functionalities in Table 3.4. In the Tables 3.11 and 3.12 we listed the number of the uses of each Eclipse navigational functionality. Note that we count only the action of a functionality. For example, Participant 1 opened type hierarchy once, but s/he might check the subtypes/supertypes multiple times as long as s/he does not close the result window. Among the six functionalities, opening declaration and searching are two most often used functionalities.

Ko et al. performed a study where the participants were asked to do two debugging tasks and three enhancement tasks. The study found three activities. The first activity is searching. We also found searching activities in the participants. The number of searches is reported in Table 3.11. Ko et al. also found that the participants lost track of relevant code because the navigational tools were used for various purposes. We found one case that is consistent with this finding. Participant 9 opened the declaration of getState seven times, and isInState three times in a debugging session. This shows that although the participant had visited the declaration of getState, s/he still had the need to find the information again.

LaToza et al. studied both novices and experts and observed their behaviors

---

5This data is preliminary and not included when we answer our research questions.
when they were asked to improve the design of the programs [94]. LaToza et al. reported that the participants made “path choice decisions”, which is choosing to explore the locations that may have useful information. We have similar notes from our participants. “Firstly I found an exception in pdfsam and it misleads me. I spend time to fix this exception.” (Participant 8) “there seems to be a.swt gui thread, the code hits this thread and pauses on a lock it is like it is waiting for a change of state so the thread can be unlocked ... investigating Game.IS_CLOCK_TICKING_STATE with the debugger” (Participant 3, Raptor, 33:06, in the notepad shown in the video)

We also found the evidence for the participants confirming or disconfirming hypotheses, which is also reported in Latoza et al.’s study. “For the Raptor project I initially had a hypothesis that there should be some event handler that does not get activated once you unpause, ... and tried to confirm/deny that first.” (Participant 7) This is also consistent with the debugging model we introduced in Section 3.4

---

6We did not require the participants to take notes. This is what we observe from the video.
TABLE 3.11

IDE ACTIVITIES IN DEBUGGING *PDFSAM*

<table>
<thead>
<tr>
<th>Participant Id</th>
<th>Call Hierarchy</th>
<th>Type Hierarchy</th>
<th>Declaration</th>
<th>Implementation</th>
<th>References</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>21</td>
<td>1</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>28</td>
</tr>
</tbody>
</table>
TABLE 3.12

IDE ACTIVITIES IN DEBUGGING *RAPTOR*

<table>
<thead>
<tr>
<th>Participant Id</th>
<th>Call Hierarchy</th>
<th>Type Hierarchy</th>
<th>Declaration</th>
<th>Implementation</th>
<th>References</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>0</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>55</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>5</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>31</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
3.9 Breadth Study Design

In this section, we will describe our breadth study, in which 35 professional programmers filled out our online survey. We will cover our research questions, online survey format, methodology, survey software, participants, and threats to validity.

3.9.1 Research Questions

In this section, we designed five research questions to study the measurable factors related to the research objectives.

RQ5. Do programmers know the term of IA?

RQ6. Do programmers self-report doing IA?

RQ7. Do programmers self-report using IA tools?

RQ8. After programmers apply a possible fix to a bug, do they run the program first or check the impacts of the fix first?

RQ9. After programmers apply a possible fix to a bug, what functionality of the IDE do they use?

The rationale behind RQ5 is that programmers may not know the term of “change impact analysis”, because it is a term heavily used in academia but perhaps not in industry. If the programmers do not know the term, they may not do IA intentionally. The rationale behind RQ6 is to determine the self-reported level of IA. The rationale behind RQ7 is that if programmers use IA tools, we know that programmers do IA. If programmers do not use the tools, it is necessary to further investigate what prevents IA techniques from practice.

RQ5 to RQ7 are self-reported questions. In addition to these self-reported questions, we have RQ8 and RQ9, which focus on the behavior of programmers. The rationale of RQ8 is that if programmers run a program immediately after they apply a possible fix, they prioritize dynamic IA after they make changes. RQ9’s rationale is that the different functionalities of the IDE that programmers use is an evidence
of whether or how they do IA. For example, if programmers click “call hierarchy” of a method, it indicates that the programmers do IA by inspecting call graphs.

3.9.2 Methodology

We used a survey study methodology, and followed the steps done by LaToza and Myers [92]. Our procedure was as follows:

1. **Formulate research questions.** We based our research questions on related literature on IA, such as the study done by Rovegård et al. [145].

2. **Design survey questions.** We aim to rely as little on self-reported level as we can. Therefore, we put $RQ_8$ and $RQ_9$ first. For these two questions, we provide a real situation to the participants: a real project and a real bug report. Then, we provide a suggested fix, so that the participants have an initial change location. Under this scenario, we ask the participants what they will do next (see Section 3.9.3 Item 3). This question corresponds to $RQ_8$. Next, we give an image of an Eclipse IDE, and we ask the participants to point out where they will go next after they make a change in source code (see Section 3.9.3 Item 4). This question corresponds to $RQ_9$. After these questions, we ask participants about IA, including whether they know about it ($RQ_5$), whether they do it ($RQ_6$), and whether they use IA tools ($RQ_7$). With this order of the questions, we mitigated the biases that may occur in the previous questions.

3. **Distribute the survey.** We recruited participants by *convenience sampling* [145]. We selected participants based on their programming experience and their availability. We targeted professional programmers in industry, especially in large software companies, see Section 3.9.4. We obtained more than 30 participants in order to have statistically significant results.

4. **Collect and analyze the results.** We used an automated process of collecting responses provided by Qualtrics Survey Software. In this way, we ensured maximum accuracy and maximum response rate. We have one hypothesis based on our results in Sections 3.10.2. For this hypothesis, we used Pearson’s Chi-square test and Fisher exact test [117]. We used Fisher exact test because some numbers in our data are smaller than five. In such cases, Fisher exact is more accurate than Pearson’s Chi-square test.
3.9.3 Online Survey Format

We built our online survey by Qualtrics Survey Software[7]. This online survey has six web pages. A deactivated survey is available at the online appendix in Section 3.9.6. The format of this survey is as follows:

1. **The first page** asks about programmers’ professional experience.

2. **The second page** shows a bug report, which is the bug report of Raptor described in Section 3.6.3.

3. **The third page** suggests a fix to that bug, and asks programmers whether they will run the fix or check the impacts of the fix first \( (RQ_8) \).

4. **The fourth page** shows an image of an Eclipse IDE. In the image, there is an IDE, where the suggested fix is applied, see Figure 3.11. There are 34 clickable areas in the image and each area corresponds to a functionality of the IDE. We asked programmers to click on the area that they will go next after they apply the fix \( (RQ_9) \).

5. **The fifth page** asks programmers whether they know about IA \( (RQ_5) \), whether they do IA \( (RQ_6) \), and whether they use any automated tools for IA \( (RQ_7) \). If the programmers do not know about IA, we used the following exact words for defining IA: “Change-Impact Analysis is a task of finding the source code which is affected by the source-code change that you are going to make.”

3.9.4 Participants

There are 35 participants. All of them self-reported that they program in Java comfortably. Seventeen of them have more than four years of professional experience in industry. Fourteen of them have one to four years of professional experience. Four of them have less than one year of professional experience. Twenty-four of them are working or worked in industry. Ten of them are graduate students. One of them programs as a hobby. In ten graduate students, six have intern experience as a programmer. The professional programmers are from the Computing Research Center (CRC) at the University of Notre Dame, and other various large IT companies.

Figure 3.11. Page 4 of the survey for our breadth study in Section 3.9.3. This page shows an image of an Eclipse IDE. The red boxes are the areas that participants can click. The areas include the window of Package Explorer, the button of Open Declaration in the context menu, and other menu options and Eclipse windows. We asked each participant to click on the area that she will go next after she makes a fix. This page is to answer RQ_9.
3.9.5 Threats to Validity

As with any study, our work has threats to validity. The project and the bug we used in the survey may not represent a general bug scenario. Also, the participants may not understand the project or the bug, so their responses may not represent their usual behavior in their own projects. However, in the survey, we do not ask the participants to actually fix the bug. Therefore, the participants do not need to consider the details of the project and the bug. Additionally, we used an image of Eclipse, which may not be the usual IDE that the participants use. We mitigate this threat by choosing Eclipse, one of the most often used IDE for Java programs.

Additionally, the population of the participants may not be representative. We mitigate this bias by having a sample of 35 programmers who have various programming experience.

3.9.6 Reproducibility

A deactivated survey, the result, and the scripts for data analysis are available in the online appendix: [http://nd.edu/~sjiang1/IA-study.htm](http://nd.edu/~sjiang1/IA-study.htm)

3.10 Breadth Study Results

In general, our breadth study shows that programmers do not use IA tools but do dynamic IA after they make changes. In the rest of the sections, we will explain how we came to this conclusion. The overall results are shown in Figures 3.12 and 3.13

3.10.1 RQ5: Knowledge of Term of Change Impact Analysis

We found evidence that the majority of the programmers do not know what IA is. In our 35 participants, nine of them reported that they knew the term of IA. Twenty-six (77%) of the programmers did not know the term IA before this study.
Our interpretation is that the academic theory of IA has not penetrated education such that the programmers leave school knowing what IA is.

3.10.2 RQ₆: Self-reported Level of Change Impact Analysis

According to our results, the majority of the programmers reported they did IA. Twenty-eight (80%) of the 35 participants reported that they did IA. We think the high self-reported level indicates that the programmers think IA is an important and a necessary process in programming.

However, we found evidence showing that programmers who know IA are less likely to think they do IA. In the nine programmers who knew IA, four (44%) programmers reported that they did IA. In the 26 programmers who did not know IA, 24 (92%) programmers reported that they did IA. We think whether programmers report they do IA is related to whether they know the IA concept. We tested the hypothesis in Table 3.13. The null hypothesis is that reporting doing IA and having heard of IA are independent. With a critical alpha level of 0.05, the null hypothesis is disproved using Pearson’s chi-square test and Fisher exact test. Therefore, whether the programmers report they do IA or not is dependent on whether they knew the concept of IA before the study. Our interpretation is that the theory of IA may be different from the industry practice. When programmers know the academic term of IA, programmers may refer IA to the IA process in the literature that is described in Section 3.3. They do not report they do IA because they do not follow the IA process. For those who do not know the academic term, they report that they do IA because they consider IA as a general concept.

3.10.3 RQ₇: Use and Knowledge of IA Tools

Our results show that the majority of the programmers do not use IA tools, which matches our finding in Section 3.8.7. Of the 28 programmers who reported that
TABLE 3.13

THE COUNTS OF THE PROGRAMMERS

<table>
<thead>
<tr>
<th></th>
<th>Heard of IA</th>
<th>Not heard of IA</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do IA</td>
<td>4</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>Do not do IA</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>All</td>
<td>9</td>
<td>26</td>
<td>35</td>
</tr>
</tbody>
</table>

* The null hypothesis: “Do/Do not do IA” and “Heard/Not heard of IA” are independent. Pearson’s chi-square statistic = 9.573, p-value = 0.002. Fisher exact test, p-value = 0.006.

they did IA, 23 programmers (82%) reported that they did not use IA tools. Three programmers (11%) reported that they were not sure whether they used tools or not. One possible explanation is that the three programmers use IDE functionalities to do IA, such as “open call hierarchy of” a method, and they are not sure whether these functionalities are IA tools or not. Overall, the result shows that most of the programmers do not use IA tools.

3.10.4 RQ8: First Step after Having a Possible Fix

In this research question, we asked what the programmers do directly after they apply a possible fix. Of the 35 participants, 17 (49%) programmers chose to run the program to see whether the fix works; 17 (49%) programmers chose to check the effects of the change; One programmer chose neither.

An explanation for running the program first is that programmers can find out 
whether the fixes work or not by running the program. If the fixes do not work, the programmers may undo the fixes. In this case, the programmers may not need to further investigate these changes.

3.10.5 RQ9: First IDE Functionality Used after a Possible Fix

To process the result, we divided the functionalities listed in the Figure 3.13 into three categories. The first category is “running” the program. There are 23% programmers who chose to run the program. By running the program, programmers can observe the impacts of a change on the outputs. So running the program is one of the simplest dynamic IA methods.

The second category is “debugging” the program, which is running the program with a debugger. We believe debugging is a type of dynamic IA because programmers can follow dependencies from the change location in execution. Figure 3.13 shows that 40% of the programmers run in debug mode directly after they make a change. The first and second categories cover 63% of the participants, which indicates that most programmers use dynamic approaches to do IA after they make changes.

The third category includes the functionalities that may help the programmers check the effects of the change for multiple places. These functionalities are opening “call hierarchy”, navigating “workspace”, “navigating outline”, “searching”, and “opening type hierarchy”. Our results show that there are 31% of the participants do IA by these static methods.

In summary, most programmers do IA after they make a change. Among the programmers who do IA, more than half of them do IA dynamically.
Figure 3.12. The results for $RQ_5$ to $RQ_8$ in our breadth study in Section 3.9. $RQ_5$: do programmers know the term of change impact analysis? $RQ_6$: do programmers think they do change impact analysis? $RQ_7$: do programmers use change impact analysis tools? $RQ_8$: after programmers make a possible fix, do they run the program first or check the impacts of the fix first?

Figure 3.13. The result of $RQ_9$ in our breadth study in Section 3.9. $RQ_9$: after programmers make a possible fix, what IDE functionality do they use?
3.10.6 Qualitative Results

We received nine comments related to IA and software engineering in our breadth study. Three comments expressed the needs for IA tools. For example, one programmer commented “Repeating the specific bug in a large project is time consuming. I would appreciate if Eclipse community can offer a tool to help me figure out which variables will be changed before tracing the code in debug mode.” Three programmers described their IA practice in the comments. One practice is single-step debugging. Another practice is “in a manual trial and error manner”. The third comment said “I would compile the fix just to see what happens first (see if it addresses the bug). If so, I would then do change impact analysis to determine if this produces new bugs.”

3.11 Discussion

In this section, we will discuss our three research objectives based on our results of the research questions in the two studies.

3.11.1 Objective1 What knowledge do programmers have of research activities in IA?

Programmers do not know the term Change Impact Analysis. In the survey, 26 out of 35 programmers do not know the term. In the interviews in the in-depth study, two of the three programmers do not know the term. 23 out of 35 programmers in the survey do not use any IA tools, and the nine programmers in the in-depth study did not use any IA tools in the video. In the three interviews, the programmers all reported that they do not use any IA tools.

However, programmers often recognized IA once the term was introduced. Most programmers in the survey reported that they do IA. In the survey, 28 out of 35 programmers reported they do IA. Three programmers in the interviews
all recognized IA and reported that they did IA. This implies that IA exists in the
industry practice and is important to the programmers. However, the IA tools are
not known to the programmers. This shows that the industry has not been benefited
from the research work of IA.

In the interviews, we found IA often is recognized as a process after programmers
have a possible fix. The importance of IA is mentioned because programmers do not
want their patches causing new bugs.

“I always try to understand how I can influence the code. If I'm uncertain about my
changes I can make a list of influenced part and give it to our QA[Quality Assurance]
engineers. They are checking all cases.” (Participant 8)

“I have to make sure that my change will not cause bugs or other problems for
other parts of the project or systems components ... ” (Participant 9)

“... change impact analysis to determine if this produces new bugs.” (a comment
in the breadth study)

Three possible reasons can explain why we do not have evidence for much post-
change impact analysis. First, in the study, the quality of the code is not a priority,
because the participants know that their changes are for research purposes and the
code will not be used by others. Second, they may not see post-change impact analysis
as a part of the debugging process. It is possible that programmers think that the bug
is fixed once the program outputs the correct output. As Participant 8 mentioned,
IA tasks can be assigned to quality assurance engineers. Quality assurance engineers
are those programmers who ensure that software meets specific quality standards.
For example, one of the most common practice of quality assurance is testing. Note
that in this chapter, we did not study the practice of quality assurance engineers.
However, IA tools may be used in such practice.

Third, in our study, we did not count the activities after the participants ran
the program, which is after FRT and LRT (in Table 3.3). The participants might
do post-change impact analysis after FRT and LRT, but we cannot distinguish the post-change IA from the pre-change IA for the next change.

Interestingly, we found evidence of programmers doing IA where IA is not recognized, which is in the middle of debugging processes. In the in-depth study, programmers ran programs and used debuggers very often after they made changes. This indicates that IA tools may be useful during debugging process, especially after programmers try a change which does not work.

3.11.2 Objective 2: What technologies do programmers use to do IA if they do any?

The plainest static method of IA is reading code by hand. Programmers use IDE navigational features to navigate through the dependencies. In the in-depth study, on average each programmer visited 16 files and used 28 times of IDE navigational functionalities during debugging.

The plainest dynamic method of IA is running program and checking the outputs. Instead of running program directly, programmers often use debuggers so that they can check the values in the middle of an execution. In the breadth study, 14 of the 35 programmers chose to use debuggers and 8 of the 35 programmers chose to run programs after they make a change. In the survey, one programmer commented “I understand change impacts mostly via single-step debugging.”

In the interviews, the programmers expressed their satisfaction about the current toolset for debugging. “VM and modern IDEs have enough tools count for app’s debugging” (Participant 9) “the default debugging functionality was sufficient for me” (Participant 7)

In the survey, however, the programmers commented about the need for IA tools. “There is still a lack of automatic supports of impact analysis.” “Repeating the specific bug in a large project is time consuming. I would appreciate if Eclipse community can offer a tool to help me figure out which variables will be changed
before tracing the code in debug mode.”

These results show IA may be helpful for debugging. For example, programmers read at most 26 files before they made changes in our study. However, neither navigational features nor debuggers keep track of the dependencies discovered among the files. If IA tools can store the dependencies of the visited files, programmers may navigate the code more efficiently.

3.11.3 Objective At what phases of debugging, do programmers do IA if they do any?

Programmers do static IA before they make a change. In the videos of the in-depth study, before the programmers made changes, they used an average of six times of IDE navigational functionalities. For the programmers without suggested fixes, they used IDE navigational functionalities for an average of eleven times before they made changes. In the interviews, the programmers also reported doing static IA in the end of debugging. “After my changes I have to find direct and indirect calls of this method and make sure that system will be ok after my changes.” (Participant 9) However, whether and how much programmers actually do IA in the end of debugging needs to be further studied.

Programmers do dynamic IA almost immediately after programmers make changes. In the in-depth study, the programmers on average visited two files and used one time of navigational functionality before they run the programs (with or without debuggers). In more than the half of the changes, the programmers visited only one file and did not use any navigational functionality. Especially, for the last modification of a change location, the programmers spent an average of 27 seconds before they ran the programs.

In the debugging process in literature (described in Section 3.4), programmers create and test hypotheses. In testing hypotheses, programmers are doing some form
of IA. From the breadth study, one programmer commented “I hadn’t heard of the term Change-Impact Analysis but I do some form of it, usually in a manual trial and error manner.” Also, we have similar comments from the interviews in the in-depth study. “first locate the issue and then confirm/deny hypothesis what is wrong” (Participant 7)

These findings imply that IA processes in practice differ in the two phases: pre-change and post-change. Likewise, IA tools may need different types of techniques in the two phases. For pre-change, static IA techniques may be preferred because they have lower cost than dynamic techniques.

For the post-change phase, dynamic IA techniques may be used because programmers often run programs and check dynamic information after they make changes. In this case, the cost of dynamic IA techniques can be much lowered. First, dynamic IA techniques are applied on only one execution, which is prepared by programmers already. Second, dynamic IA techniques are applied on one specific change, which can be identified automatically by comparing the current code and the code ran in the previous execution.

3.12 Conclusion

In this chapter, we did two studies to find out whether programmers do change impact analysis (IA). In our in-depth study, we hired nine professional programmers repairing two real bugs in two open source systems. We recorded and analyzed the videos of their debugging processes. In our breadth study, we hired 35 professional programmers to fill out our online survey. In the online survey, we asked them about what they know about IA and what they do to fix a bug.

From our two studies, we discovered the evidence of programmers doing IA. In the in-depth study, we found the evidence indicating programmers doing IA, and in the breadth study, most programmers reported that they did IA. Second, we found the
evidence that the practice of IA is different from the process of IA described in the
literature. No programmer in our in-depth study used any IA tools, even though they
did IA to fix the bugs. In the breadth study, most of the programmers reported they
did not use IA tools and most of the programmers did not know the term “change
impact analysis”. The purposes of IA can be various. We found programmers tend
to think IA as a process that should be done after debugging. However, during
debugging, programmers also do IA to figure out how to fix the bugs.
CHAPTER 4

AUTOMATICALLY GENERATING COMMIT MESSAGES FROM DIFFS USING NEURAL MACHINE TRANSLATION

This chapter represents work published in the Proceeding of the 32nd IEEE/ACM International Conference on Automated Software Engineering 2017 [76].

4.1 Introduction

Commit messages are natural language descriptions of changes in source code. When a programmer updates code, a typical procedure is to upload the change to a version control system with a short commit message to describe the purpose of the change, e.g., “adds support for 9 inch tablet screen size.” The repository stores the message alongside a diff that represents the difference between the current and previous version of the affected files. The practice is extremely common: for this chapter alone, we obtained over 2M diffs and messages from just 1K projects.

Commit messages are useful because they help programmers to understand the high level rationale for a change without reading the low level implementation details. They serve a valuable purpose in comprehension of software evolution, and act as a record of feature additions and bug repairs [29]. Unfortunately, programmers sometimes neglect commit messages [47, 109], likely due to the same time and market pressures that have been reported to affect many types of documentation [51, 81, 144]. In short, programmers use commit messages but often avoid writing them themselves.

Automated generation of commit messages has been proposed as an alternative to manual efforts by programmers. For example, Buse et al. [29] describe DeltaDoc,
a tool that summarizes what changed in the control flow of a program between code versions. Likewise, Cortes-Coy et al. [103] built ChangeScribe, which summarizes changes such as method additions. These and other existing techniques (see Section 2.4) have been shown to be effective in answering questions about what changed and where from one code version to another.

What is missing from existing approaches is a short, high level description of the purpose behind commits. Current approaches are effective at summarizing what changed and where, but do not answer the question why [29]. Questions of why traditionally require human insight since they involve synthesis of different, complex data sources and context. However, as Mockus et al. [122] observed, many commit messages are similar and can be broadly categorized as related to bug repair, feature additions, etc. Plus, they follow similar grammatical patterns such as verb-direct object structure (e.g. “adds support for...”) [75]. This observation leads us to believe that the text of commit messages can be learned and predicted if there is sufficient data. Our view is in line with the hypothesis of “naturalness” of software [70], that software artifacts follow patterns that can be learned from sufficiently large datasets.

In this chapter, I adapt a neural machine translation (NMT) algorithm to the problem of commit message generation. Several NMT algorithms have been designed to translate between natural languages by training a neural network on pairs of sentences that humans have already translated. The datasets required are enormous by typical software engineering research standards, involving up to tens of millions of pairs of sentences [107, 151]. We trained an NMT algorithm using pairs of diffs and commit messages from 1K popular projects on GitHub. While we were able to obtain quite large datasets (over 2M commits), we encountered many commit messages that were gibberish or very low quality (a problem others have observed [47, 109]), which if left in the training data could be reflected in the NMT’s output. Therefore, we designed a filter to ensure that we only trained the algorithm using messages with a
verb-direct object pattern.

We investigate and report the effectiveness of the predictions from the process. We found promising results as well as key constraints on the accuracy of the predictions. In short, the NMT process performed quite well under select conditions, but poorly in others. We report these results and promising and poor conditions as a guide to other researchers and platform for advancement in this research area. To further promote advancement of the area, we make our implementation and data freely available in an online replication package.

Our approach has two key advantages that make it a supplement to, rather than a competitor of, existing automatic commit message generation techniques. First, we produce short summary messages rather than exhaustive descriptions of code changes. And second, our approach produces messages for changes to many types of software artifact in the repository, not solely source code.

4.1.1 The Problem

In this chapter, we target the problem of automatically generating commit messages. Commit messages are useful in the long term for program comprehension and maintainability, but cost significant time and effort in the short term. These short term pressures lead programmers to neglect writing commit messages, like other types of documentation [47, 51, 81, 109, 144]. Buse et al. [29] point out that programmers use commit messages for two reasons: 1) to summarize what changed, and 2) to briefly explain why the change was necessary. To date, research into commit message generation has exclusively focused on the question what. In this chapter, we seek to begin answering why.

Existing commit message generation techniques produce relatively long messages that include details such as the methods that were added or the number of files changes (what information). While useful, these techniques are a complement to,
rather than a replacement for, high level *why* information that humans write such as “adds support for 9 inch tablet screens.” Normally, this high level information requires human judgment. But we hypothesize that there are patterns of commits, and that these patterns can be detected and used to generate messages for similar commits later. Given a large number of pairs of *diffs* and messages, we believe we can train an algorithm to write new messages for new commits, based on the new commits’ similarity to older ones.

**Please note** that we do not claim to generate new insights for completely new types of commits — that task is likely to remain in the hands of human experts. However, we do aim to write messages that reflect knowledge that can be learned from records of previous commits. In the long run, we hope that this technology will help reduce manual effort by programmers in reading and understanding code changes in repositories.

### 4.1.2 Chapter Overview

Figure [4.1](#) depicts an overview of this chapter. We have divided the work into three segments: In Part A (Section [4.3](#)), we present our approach to filtering for verb/direct-object (V-DO) commit message patterns and training an NMT algorithm to produce messages. The V-DO filter was introduced because a large percentage of the messages in the repositories we downloaded were very low quality, and we needed to ensure that we trained the NMT algorithm only with examples matching an acceptable pattern. We then trained an NMT algorithm on the pairs of *diffs* and commit messages where the messages followed the V-DO pattern.

In Part B (Sections [4.4](#) and [4.5](#)), we evaluate the quality of the commit messages produced by the algorithm with an automated method and a human study with 2 Ph.D. students and 18 professional programmers. We observe that while there are a significant number of positive results, there are also a significant number of negative
results. Therefore, in Part C (Sections 4.6), we design a quality assurance (QA) filter to detect cases in which the NMT algorithm is likely to produce a negative result. We then modify our approach to produce a warning message instead of a commit message in those cases, and update our evaluation to show the effects of our modification. In short, we reduce the number of poor predicted messages by 44% at a cost of also mistakenly reducing high quality predictions by 11%.

4.2 Background

We split the background section into three subsections. The first subsection is about the empirical studies on commit messages, which motivate us to generate short descriptions of commits. The second subsection describes \textit{RNN Encoder-Decoder}, a popular Neural Network Translation model, which is an important background for the third subsection. The third subsection describes \textit{attentional RNN Encoder-Decoder}, which is used in our work.
4.2.1 Commit Messages

Empirical studies of what commit messages are like has been emerging [6, 29, 66, 69, 113, 123]. The work in this chapter is motivated by the findings of the studies by Buse et al. [29] and by our previous study [75]. The results of the two studies indicate three things. First, commit messages are pervasive and desired. Buse et al. examined 1K commits from five mature software projects and found that 99.1% of the commits have non-empty messages. We collected over 2M commit messages from 1K projects.

Second, human-written commit messages are short. In Buse et al.’s study, the average size of the 991 non-empty commit messages is 1.1 lines. Similarly, our study shows that 82% of the commit messages (excluding merges and rollbacks) have only one sentence.

Third, commit messages contain various types of information not solely summaries of code changes. Buse et al. manually analyzed 375 commit messages and found that the messages are not only about what the changes are but also about why the changes are made. Supported by the three findings, our technique aims to generate one-sentence commit messages which mimic the human-written commit messages.

4.2.2 RNN Encoder-Decoder Model

Neural Machine Translation (NMT) is neural networks that model the translation process from a source language sequence $x = (x_1, ..., x_n)$ to a target language sequence $y = (y_1, ..., y_n)$ with the conditional probability $p(y|x)$ [8, 108]. Cho et al. introduced RNN Encoder-Decoder as an NMT model [38], which is commonly used and can produce state of the art translation performance [107, 151]. As a promising deep learning model, RNN Encoder-Decoder has been used in addressing other software engineering tasks [8, 59].

RNN Encoder-Decoder has two recurrent neural networks (RNNs). One RNN is
used to transform source language sequences into vector representations. This RNN is called the encoder. The other RNN is used to transform the vector representations to the target language sequences, which is called the decoder.

4.2.2.1 Encoder

The input of the encoder is a variable-length sequence \( x = (x_1, ..., x_T) \). The encoder takes one symbol at a time as shown in Figure 4.2. As an RNN, the encoder has a hidden state \( h \), which is a fixed-length vector. At a time step \( t \), the encoder computes the hidden state \( h_t \) by:

\[
h_t = f(h_{t-1}, x_t)
\]  

(4.1)

where \( f \) is a non-linear function. Two common options for \( f \) are long short-term memory (LSTM) \([71]\) and the gated recurrent unit (GRU) \([38]\) (due to space limit, we do not describe these two unit types in detail here). For example, Bahdanau et al. use GRU \([12]\) and Sutskever et al. use LSTM \([162]\). The last symbol of \( x \) should be an end-of-sequence (\(<\text{eos}>\)) symbol which notifies the encoder to stop and output the final hidden state \( h_T \), which is used as a vector representation of \( x \).

4.2.2.2 Decoder

Figure 4.3 shows the RNN of the decoder. The output of the decoder is the target sequence \( y = (y_1, ..., y_T') \). One input of the decoder is a \(<\text{start}>\) symbol denoting the beginning of the target sequence. At a time step \( t \), the decoder computes the hidden state \( h'_t \) and the conditional distribution of the next symbol \( y_t \) by:

\[
h'_t = f(h'_{t-1}, y_{t-1}, h_T)
\]  

(4.2)
\[ p(y_t | y_{t-1}, \ldots, y_1, h_T) = g(h'_t, y_{t-1}, h_T) \]  

(4.3)

where \( h_T \) (generated by the encoder) is called the context vector; \( f \) and \( g \) are non-linear functions. Function \( f \) here and \( f \) in Equation 4.1 are often the same. Function \( g \) must produce valid probabilities. For example, softmax can be used as \( g \). The decoder finishes when it predicts an \(<eos>\) symbol.

4.2.2.3 Training Goal

The encoder and the decoder are jointly trained to maximize the conditional log-likelihood:

\[ \max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p(y_i | x_i; \theta) \]  

(4.4)

where \( \theta \) is the set of the model parameters; \( N \) is the size of the training set; and each \((x_i, y_i)\) is a pair of a source sequence and a target sequence in the training set.

4.2.3 Attentional RNN Encoder-Decoder and Nematus

Bahdanau et al. introduced the attentional RNN Encoder-Decoder, in which attention mechanism is introduced to deal with long source sequences [12]. We use this mechanism in our work because our source sequences, \textit{diffs}, are much longer than natural language sentences. The attention mechanism includes several modifications in both the encoder and the decoder, which we describe in the following subsections.

![Figure 4.2. The architecture of the encoder in RNN Encoder-Decoder](image-url)
4.2.3.1 Encoder

The encoder in the attentional model is a bidirectional RNN, which has two RNNs: forward and backward. The two RNNs have the same architecture. The forward RNN is the same as the RNN in the original RNN Encoder-Decoder model (Figure 4.2), which reads the source sequence \( x \) as it is ordered, from \( x_1 \) to \( x_T \). The forward RNN generates a sequence of the hidden states \((\overrightarrow{h_1}, ..., \overrightarrow{h_T})\). In contrast, the backward RNN reads \( x \) in the reversed order, and generates a sequence of the hidden states \((\overleftarrow{h_T}, ..., \overleftarrow{h_1})\).

In the end, for each symbol \( x_i \) in \( x \), the encoder outputs \( h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}] \), which is a concatenation of \( \overrightarrow{h_i} \) and \( \overleftarrow{h_i} \).

4.2.3.2 Decoder

The decoder computes the hidden state \( h_t' \) and the conditional distribution of the next symbol \( y_t \) by:

\[
h_t' = f(h_{t-1}', y_{t-1}, c_t) \quad (4.5)
\]

\[
p(y_t|y_{t-1}, ..., y_1, c_t) = g(h_t', y_{t-1}, c_t) \quad (4.6)
\]
where $f$ and $g$ are non-linear functions like $f$ and $g$ in Equations 4.2 and 4.3. $c_t$ is the distinct context vector for $y_t$, and can be computed by

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i$$  \hspace{1cm} (4.7)

where $T$ is the length of the input sequence; the weight $\alpha_{ti}$ can be trained jointly with the other components in the model, and $h_i$ is generated by the encoder. Since $c_t$ is designed to introduce the context’s impact to $y_t$, **attentional RNN Encoder-Decoder** works better on long source sequences. Therefore, we use this NMT model in this chapter rather than the original one.

4.3 Approach

This section describes our approach, including the data set preparation and the NMT training procedure. This section corresponds to Part A in the chapter overview Figure 4.1 and is detailed in Figure 4.4.

4.3.1 Preparing a Data Set for NMT

We used the commit data set obtained in our previous study [75], which contains 2M commits. The data set includes commits from top 1K Java projects (ordered by the number of stars) in Github. We describe how we prepared the data set for NMT algorithms as follows.

4.3.1.1 Preprocessing the Data Set

First, we extracted the first sentences from the commit messages by using a natural language processing tool, Stanford CoreNLP [111]. We used the first sentences as the target sequences because of two reasons. One reason is that 82% of the messages have only one sentence [75], so one-sentence messages are close to the existing
human-written messages. The other reason is that the first sentences often are the summaries of the entire commit messages. Similarly, Gu et al. used the first sentences of the API comments as their target sequences [59].

Then, we removed issue numbers from the extracted sentences and removed commit ids from the diffs, because these numbers are unique ids and increase the vocabularies of the source and the target languages dramatically, which in turn cause large memory use of NMT.

Third, we removed merge and rollback commits (the same practice done in our previous study [75]). Merges and rollbacks are removed because the diffs of merges and rollbacks are often more than thousands of lines, which NMT is not suitable to translate. For the same reason, we also removed any diff that is larger than 1MB.

Figure 4.4. The detailed process in Part A, Figure 4.1. There are three main steps: 1) filtering and preprocessing the data; 2) training a Neural Machine Translation model; 3) evaluating the model, which is Part B, Figure 4.1.
After the above steps, we have 1.8M commits remaining. Finally, we tokenized the extracted sentences and the diffs by white spaces and punctuations. We did not split CamelCase so that identifiers (e.g., class names or method names) are treated as individual words in this study.

4.3.1.2 Setting Maximum Sequence Lengths for NMT Training

A maximum sequence length for both source and target sequences need to be set for an RNN Encoder-Decoder. Since NMT is for translating natural language sentences, maximum sequence lengths for both source and target sequences are often set between 50 to 100. Because the lengths of our source and target sequences are very different, we set the maximum sequence lengths separately.

For our target sequences, we set the maximum length at 30 tokens (including words and punctuations), because the first sentences from the commit messages tend to be short. In our data set, 98% of the first sentences have less than 30 tokens. For our source sequences, we set the maximum length at 100 tokens because 100 is the largest maximum length used by NMT in natural language translation. Many configurations are possible, and optimizing the maximum diff length for generating commit messages is an area of future work. In pilot studies, a maximum length of 100 outperformed lengths of 50 and 200.

After applying the maximum lengths for source and target sequences (30 and 100), we have 75k commits remaining.

4.3.1.3 V-DO Filter

We introduced Verb-Direct Object (V-DO) filter because we found that the existing messages have different writing styles and some of the messages are poorly written, which may affect the performance of NMT.

To obtain a set of commit messages that are in a similar format, we filtered the
messages for verb/direct-object pattern. We chose this pattern because a previous study shows that 47% of commit messages follow this pattern [75]. To find the pattern, we used a Natural Language Processing (NLP) tool, Stanford CoreNLP [111], to annotate the sentences with grammar dependencies. Grammar dependencies are a set of dependencies between parts of a sentences. Considering a phrase, “program a game”, this phrase has a dependency, which is called “dobj” in Stanford CoreNLP, where the governor is “program” and the dependent is “game”. For V-DO filter, we look for “dobj” dependencies which represent the verb/direct-object pattern.

For each sentence, we checked whether the sentence is begun with a “dobj” dependency. If the sentence is begun with a “dobj”, we mark the sentence as a “dobj” sentence. In the end, we have 32k commit messages that are “dobj” sentences.

4.3.1.4 Generating Training/Validation/Test Sets

We randomly selected 3k commits for testing, 3k commits for validation, and the rest 26k commits for training.

4.3.1.5 Selecting Vocabularies

NMT needs predefined vocabularies for commit messages and diffs. In the training set, the commit messages have 16K distinct tokens (words or punctuations) and the diffs have 65K distinct tokens. We selected all the 16K tokens in the commit messages to be the vocabulary of commit messages. We used the most frequent 50K tokens in the diffs to be the vocabulary of diffs. All the tokens that are not in the diff vocabulary only occur once in the training set. Additionally, the vocabulary size of 50K is often used by other NMT models [107].
4.3.2 NMT Training and Testing

In this section, we describe how we trained and tested an NMT model for generating commit messages.

4.3.2.1 Model

We used Nematus \cite{152} in our work because it is robust, easy to use, and produced best constrained systems for seven translation directions (e.g., English to German, etc.) in WMT 2016 shared news translation task \cite{151}. Nematus is based on Theano \cite{165}, and implements the attentional RNN encoder-decoder (see Section 4.2.3) with several implementation differences \cite{152}.

4.3.2.2 Training Setting

We borrowed the training setting that Sennrich \textit{et al.} used to produce the best translation systems in WMT 2016 \cite{151}. The training goal is cross-entropy minimization \cite{146}. The learning algorithm is stochastic gradient descent (SGD) with Adadelta \cite{180}, which automatically adapts the learning rate. The size of minibatches is 80; the size of word embeddings is 512; the size of hidden layers is 1024. For each epoch, the training set is reshuffled. The model is validated every 10K minibatches by BLEU \cite{134}, which is a commonly used similarity metric for machine translation. The maximum number of epochs is 5K; the maximum number of minibatches is 10M; and early stopping is used \cite{152}. During the training, the model is saved every 30K minibatches. So after the training, a list of models are saved and the ensemble results of the last four models are used for evaluation.

One key difference between our and Sennrich \textit{et al.}’s training processes is that Sennrich \textit{et al.} used maximum sentence length of 50 for all the languages; we used 30 for commit messages and 100 for \texttt{diffs} as explained in Section 4.3.1.2.
4.3.2.3 Training Details

We trained on the training set of 26K pairs of commit messages and diffs, with a validation set of 3K pairs. We conducted the training on an Nvidia GeForce GTX 1070 with 8GB memory. The learning algorithm stopped at 210K minibatches. Because a model is saved every 30K minibatches, seven models are saved from this training. The training process took 38 hours.

4.3.2.4 Testing Details

While we describe our evaluation in the next section, certain technical details are relevant here. We ran Nematus with the last four saved models on the testing set and we obtained the ensemble result. We used the same GPU as we used in training. The testing process took 4.5 minutes. We note that we followed the standard evaluation procedure for NMT and used a test set of 3K [38, 107, 150].

4.4 Evaluation Using An Automatic Metric

In this section, we evaluate the generated messages from our approach that we described in the last section. Our objective is to assess the similarity between the generated messages and the reference messages in the test set. This section corresponds to Part B in the chapter overview Figure 4.1. Note that this evaluation is distinct from the experiment with human evaluators that we describe in Section 4.5, which is also a component of “Part B.” In this section we ask:

RQ1 Compared to the messages generated by a baseline, are the messages generated by the NMT model more or less similar to the reference messages?

RQ2 Are the messages generated by the NMT model more or less similar to the reference messages when V-DO filter is enabled or disabled?

We ask RQ1 to evaluate the NMT model compared to a baseline, which we describe in the following subsection. We ask RQ2 in order to evaluate the impact of
V-DO filter. In the following subsections, we first introduce the baseline for RQ1. Then, we introduce the metric for measuring similarity between two messages. Finally, we report our results for the research questions.

4.4.1 Baseline: MOSES

We used MOSES [88] as the baseline in RQ1. MOSES is a popular statistical machine translation software, which is often used as a baseline in evaluating machine translation systems [30, 87]. For example, Iyer et al. used MOSES as a baseline when they evaluated Code-NN [74]. To run MOSES for translating diffs to commit messages, we trained a 3-gram language model using KenLM [67, 68], which is the same procedure in the study of Iyer et al. [74]. We did not use Code-NN as a baseline, because, in our pilot study of running Code-NN [74] to generate commit messages, Code-NN did not generate comparable results. A possible reason is that Code-NN needs parsing source sequences and diffs are not suitable for parsing.

4.4.2 Similarity Metric: BLEU

BLEU [134] is widely used to measure the similarity between two sentences in evaluation of machine translation systems [85, 104, 107]. Additionally, BLEU is recommended for assessing an entire test set instead of a sentence [134]. The calculation of BLEU needs the modified n-gram precisions. For any $n$, the modified $n$-gram precision is calculated by:

$$p_n = \frac{\sum_{(gen, ref) \in test} \sum_{ngram \in gen} Cnt_{clip}(ngram)}{\sum_{(gen, ref) \in test} \sum_{ngram \in gen} Cnt_{gen}(ngram)}$$  \hspace{1cm} (4.8)
\[ Cnt_{\text{clip}}(\text{ngram}) = \min(Cnt_{\text{gen}}(\text{ngram}), Cnt_{\text{ref}}(\text{ngram})) \]

where test is the set of pairs of the generated and the reference messages in the test set; gen is the set of distinct n-grams in a generated message; Cnt_{\text{clip}} is defined in Equation (4.9); Cnt_{\text{gen}} is the number of occurrences of an n-gram in a generated message; similarly, Cnt_{\text{ref}} is the number of the occurrences of an n-gram in a reference message. Then, BLEU is:

\[ \text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log(p_n)\right) \]

\[ BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \]

where \( N \) is the maximum number of grams; \( p_n \) is defined in Equation (4.8); \( BP \) is defined in Equation (4.11); \( r \) is the sum of the lengths of all the reference messages; \( c \) is the sum of the lengths of the generated messages. BLEU scores range from 0 to 100 (in percent). The default value of \( N \) is 4, which is used in our evaluation and is commonly used in other evaluations [59, 74, 85, 104, 107, 150].

4.4.3 RQ1: Compared to the Baseline

The first two rows in Table 4.1 list the BLEU scores of MOSES and the NMT model we trained in Section 4.3.2, which we refer to as NMT1. The BLEU score of our model is 31.92 while the BLEU score of MOSES is 3.63, so according to the BLEU metric, the messages generated by the NMT model are more similar to the reference messages than the messages generated by the baseline. One key reason that
the attentional NMT model outperforms MOSES is that MOSES does not handle well very long source sequences with short target sequences. Particularly, MOSES depends on Giza++ [129] for word alignments between source and target sequences, and Giza++ becomes very inefficient when a source sequence is 9 times longer than the target sequence or vice versa [4]. Table 4.1 shows that the total length of the generated messages ($\text{Len}_{\text{Gen}}$ in Table 4.1) of MOSES is much longer than the total length of the reference messages, which may cause the modified n-gram precisions ($p_1$, $p_2$, $p_3$, and $p_4$), of MOSES to be small.

To further examine the messages generated by our model, we split the test set by the lengths of the diffs into four groups and calculated BLEU scores separately for each group. Figure 4.5 shows the distribution of the lengths of diffs in the test set and Table 4.2 shows the BLEU scores for the diffs. This table shows that the diffs that have more than 75 tokens have the highest BLEU score. One possible reason is that there are many more diffs that have more than 75 tokens than the other smaller diffs. Figure 4.6 shows the distribution of the diff lengths in the training set. This figure shows that the training set is populated by larger diffs, which may cause the model to fit the larger diffs better.
TABLE 4.1

BLEU SCORES (%) OF MOSES AND OUR MODELS

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>Len&lt;sub&gt;Gen&lt;/sub&gt;</th>
<th>Len&lt;sub&gt;Ref&lt;/sub&gt;</th>
<th>p&lt;sub&gt;1&lt;/sub&gt;</th>
<th>p&lt;sub&gt;2&lt;/sub&gt;</th>
<th>p&lt;sub&gt;3&lt;/sub&gt;</th>
<th>p&lt;sub&gt;4&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSES&lt;sup&gt;a&lt;/sup&gt;</td>
<td>3.63</td>
<td>129889</td>
<td>22872</td>
<td>8.3</td>
<td>3.6</td>
<td>2.7</td>
<td>2.1</td>
</tr>
<tr>
<td>NMT1</td>
<td>31.92</td>
<td>24344</td>
<td>22872</td>
<td>38.1</td>
<td>31.1</td>
<td>29.5</td>
<td>29.7</td>
</tr>
<tr>
<td>NMT2</td>
<td>32.81</td>
<td>21287</td>
<td>22872</td>
<td>40.1</td>
<td>34.0</td>
<td>33.4</td>
<td>34.3</td>
</tr>
<tr>
<td></td>
<td>23.10&lt;sup&gt;b&lt;/sup&gt;</td>
<td>20303</td>
<td>18658</td>
<td>30.2</td>
<td>23.3</td>
<td>20.7</td>
<td>19.6</td>
</tr>
</tbody>
</table>

<sup>a</sup> MOSES is the baseline model. NMT1 is the NMT model with V-DO filter described in Section 4.3.2. NMT2 is a model trained without V-DO filter described in Section 4.4.4. Len<sub>Gen</sub> is the total length of the generated messages (c in Equation (4.11)). Len<sub>Ref</sub> is the total length of the reference messages (r in Equation (4.11)). The modified n-gram precision p<sub>n</sub>, where n = 1, 2, 3, 4, is defined in Equation (4.8).

<sup>b</sup> This BLEU score is calculated on a test set that is not V-DO filtered described in Section 4.4.4. The other BLEU scores are tested on a V-DO filtered test set described in Section 4.3.1.4.
## Table 4.2

**BLEU Scores (%) on Diffs of Different Lengths**

<table>
<thead>
<tr>
<th>Diff Length</th>
<th>BLEU</th>
<th>Len_{Gen}</th>
<th>Len_{Ref}</th>
<th>p₁</th>
<th>p₂</th>
<th>p₃</th>
<th>p₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ 25</td>
<td>6.46</td>
<td>870</td>
<td>655</td>
<td>18.6</td>
<td>6.9</td>
<td>4.3</td>
<td>3.1</td>
</tr>
<tr>
<td>&gt; 25, ≤ 50</td>
<td>9.31</td>
<td>3627</td>
<td>3371</td>
<td>23.1</td>
<td>10.8</td>
<td>6.6</td>
<td>4.5</td>
</tr>
<tr>
<td>&gt; 50, ≤ 75</td>
<td>12.67</td>
<td>4779</td>
<td>4418</td>
<td>24.8</td>
<td>14.1</td>
<td>9.8</td>
<td>7.6</td>
</tr>
<tr>
<td>&gt; 75</td>
<td>43.33</td>
<td>15068</td>
<td>14428</td>
<td>47.1</td>
<td>42.3</td>
<td>41.7</td>
<td>42.3</td>
</tr>
</tbody>
</table>

* See Table 4.1 for explanation of each column name. The BLEU scores are calculated based on the test results generated by Model1, the NMT model with V-DO filter trained in Section 4.3.2.

![Figure 4.6](image.png)

**Figure 4.6.** The distribution of the lengths of diffs in the training set
In Table 4.2, the modified 4-gram precision, $p_4$, is 7.6 when diff lengths are between 25 and 50, and becomes 42.3 when diff lengths are larger than 75. This increase of $p_4$ means that the number of the 4-grams that are shared by the generated and reference messages increase dramatically when the lengths of diffs increase to more than 75 tokens. In contrast, $p_4$ changes much less (3.1 to 4.5, 4.5 to 7.6) in other cases.

4.4.4 RQ2: Impact of V-DO Filter

Besides NMT1 (the NMT model trained with V-DO filter in Section 4.3), we trained another model without V-DO filter, which we refer to as NMT2. In this subsection, we compare NMT1 and NMT2 to see the impact of V-DO filter.

Then, we randomly selected 3K for validation and used the rest 66K commits for training. We note that the training set of NMT1 has only 26K commits, so NMT2 has 2.5 times more training data than NMT1. The training set includes 45K distinct tokens in commit messages and 110K distinct tokens in diffs. Similar to the vocabulary setting we used in Section 4.3.1.4, we used all the 45K tokens to be the vocabulary of commit messages. We used the most frequent 100K tokens in diffs to be the vocabulary of diffs. All the tokens that are not included in the vocabulary only occur once in the training set. We followed the same process described in Section 4.3.2. The training process took 41 hours. The testing process for Test1 took 21.5 minutes and Test2 took 20 minutes.

4.4.4.1 Results

The third and fourth rows in Table 4.1 show the BLEU scores of NMT2 on Test1 and Test2, which are 32.81 and 23.10 respectively. Comparing the BLEU scores of NMT1 and Test1, the result shows that the messages generated by NMT2 are more similar to the reference messages in Test1. This finding indicates that although the
training set without V-DO filter has low-quality messages, there are valuable commits that do not follow the V-DO pattern but help the NMT model improve over Test1 which follow the V-DO pattern.

However, the BLEU score of Test2 is about 10 percent lower than the BLEU score of Test1, which means that NMT2 does not perform well over the commits that do not follow the V-DO pattern. For example, a reference message in Test2 is “7807cb6 ca7a229”, which should be version numbers. For such reference messages in Test2, the NMT model cannot generate the same version numbers and is not meant to generate such numbers. However, similar messages in the training set cause the NMT model to try to generate such numbers for commit messages. For example, a generated message in Test2 is “Dd38b1cc2 92007d1d7” while the reference message is “Run only on jdk7 for the moment”.

4.5 Human Evaluation

In this section, we ask human experts to evaluate the generated messages by the NMT model we described in Section 4.3. In Section 4.4, we evaluated our model by the automatic metric, BLEU. Our human study complements the evaluation that uses BLEU in two ways. First, although BLEU is a widely used metric that enables us to compare our model with others and to deliver reproducibility, BLEU is not recommended for evaluating individual sentences [134]. Our human study can show how our model perform on individual messages. Second, BLEU calculates the textual similarity between the generated and the reference messages, while the human study can evaluate the semantic similarity.

In this study, we hired 20 participants for 30 minutes each to evaluate the similarity in a survey study. Two participants are computer science Ph.D. students and 18 participants are professional programmers with 2 to 14 years experience. In the rest of this subsection, we describe our survey design, the process of conducting the
Example 1 of 3

*message 1:* "Added X to readme"
*message 2:* "edit readme"

**Recommended score:** 6

**Explanation:** The two messages have only one shared word, "readme". But the two messages are very similar in the meaning, because "Added" is a type of "edit".

Figure 4.7. An scoring example we gave to the participants in the survey study.

survey, and the survey results.

4.5.1 Survey Design

We introduce our survey in the first page as: “This survey will ask you to compare two commit messages by their meaning. You will be able to select a score between 0 to 7, where 0 means there is no similarity and 7 means that two messages are identical.” We permitted the participants to search the internet for unfamiliar concepts. Then, we gave three scoring examples with recommended scores of 6, 3, and 1. Due to space limit, we present only the first example in Figure 4.7 (all the other examples are available in our online appendix, Section 4.10). Then, in the remaining pages of the survey, each page has one pair of the messages, and we asked the participants to score the similarity by meaning. Note that the participants do not know who/what generated the messages. The order of the messages in every page is randomly decided. In the end of the page, there is an optional text box for the participants to enter their justifications. A formal qualitative study about the participants’ comments will need to be performed in the future but is beyond the scope of this study. Figure 4.8 shows one page of the survey.
Below are two commit messages,

*Message 1*: Added Android SDK Platform with API level 16 to Travis build file
*Message 2*: Remove redundant commands in travis config.

How **similar** are the two messages (in terms of the **meaning**)?

0  1  2  3  4  5  6  7  
no similarity whatsoever  
identical

Figure 4.8. One of the pages that we ask the participants to score the similarity. There is an optional text box for the participants to write their justifications in the end of the page. This text box is omitted due to space limit.

4.5.2 Survey Procedure

First, the pairs of generated/reference messages are randomly ordered in a list. Then, for each participant, a survey is generated with the messages in the list from a given starting point. For example, for the first three participants, the surveys are generated with the messages starting from the first pair in the list. In 30 minutes, the first participant was able to score 107 pairs; the second participant was able to score 61 pairs; the third participant was able to score 99 pairs. So the first 61 pairs of messages were evaluated by three participants. For the fourth participant, we generated a survey starting from the 62th pair and the participant stopped at 99th pair in 30 minutes. So after the first four participants, we have 99 pairs scored by three participants. Although it would be ideal if we obtain three scores for every pair, we did not enforce all the pairs being scored by three participants because we want to have more pairs scored with the limited number of participants. In the end, 226 pairs were scored by three participants, 522 pairs were scored by two participants, and 235 pairs were scored by one participant.
Figure 4.9. The distribution of the median scores obtained in the human study. There are 983 scores in the figure. Each score is the median score of the scores made by one to three human experts for a generated message. The scores range from 0 to 7, where 0 denotes the generated message is not similar to the reference message at all, and 7 denotes the generated message is identical to the reference message. The most frequent scores are 0 and 7. There are 248 messages scored 0 and 234 messages scored 7. For the rest of the scores, the number of messages ranges from 68 to 100.

4.5.3 Results

Figure 4.9 shows the distribution of the median scores of the semantic similarity of the generated/reference messages. To be conservative, we round down the median scores. For example, if a generated message has two scores, 1 and 2, and the median score is 1.5, we round down the median score to 1. In total, 983 generated commit messages have scores made by the participants. Zero and seven are the two most frequent scores. There are 248 messages scored 0 and 234 messages scored 7, which shows that the performance of our model tends to be either good or bad.

4.6 Quality Assurance Filter

Based on the results from our study with human evaluators (Section 4.5), we propose a quality assurance filter (QA filter) to automatically detect the diffs for
which the NMT model does not generate good commit messages. By building this filter, we investigate whether it is possible to automatically learn the cases where our NMT model does not perform well. In this section, we describe the method of our filter, how we evaluate the filter, and the performance of the filter. This section corresponds to Part C in the chapter overview Figure 4.1.

4.6.1 QA Filter

Our method of QA filter has three steps. First, we prepared the gold set. We used the evaluated messages and the corresponding diffs in the human study as our gold set. For each diff and the corresponding generated message, there is a score we obtained in the human study (Figure 4.9) that indicates whether the generated message for the diff is similar to the reference message (i.e., the actual human-written message). To be conservative, we labeled the diffs that have scores of zero or one as “bad” and all the other diffs as not “bad”.

Second, we extracted the features of the diffs. We used term frequency/inverse document frequency (tf/idf) for every word in a diff as the features. Tf/idf is widely used in machine learning for text processing [63], which is computed based on the frequency of a word in a diff and whether the word is common in the other diffs.

Finally, we used the data set of diffs and their labels to train a linear SVM using stochastic gradient descent (SGD) as the learning algorithm. After we trained the SVM, to predict whether the NMT model will generate a “bad” commit message for a diff, we extract tf/idfs from the diff and run the trained SVM with the tf/idfs.

4.6.2 Cross-Validation Evaluation

Figure 4.10 illustrates our 10-fold cross-validation process. We shuffled the gold set first, and split the gold set into 10 folds. For each fold, we trained a SVM model on the other 9 folds, and tested the SVM model on the one fold. In the end, we
obtained the test results for every fold. Figure 4.11 shows the predicts of all the folds. In terms of detecting diffs for which the NMT model will generate “bad” messages, QA filter has 44.9% precision and 43.8% recall. Furthermore, if we label the messages with scores of 6 or 7 as “good”, in this evaluation, QA filter reduced 44% of the “bad” messages at a cost of 11% of the “good” messages.

4.7 Example Results

4.7.1 A Representative Result

Table 4.3 shows a representative example of a generated message that was rated highly by the human experts. It includes the generated and reference messages, three scores made by three participants, and the corresponding diff. In this example, the reference message refers to a replacement of a call to a function called deactivate() with a call to a function close(). To a human reader, that is evident from the diff: a call to deactivate() is removed and a call to close() is added. The NMT algorithm also picked up on this change, generating text “Close instead of mCursor.Deactivate.”
<table>
<thead>
<tr>
<th>TABLE 4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE RESULT</td>
</tr>
</tbody>
</table>

Diff:
```diff
--- a/core/.../CursorToBulkCursorAdaptor.java
+++ b/core/.../CursorToBulkCursorAdaptor.java
@@ -143,8 +143,7 @@ public final class
     CursorToBulkCursorAdaptor ...
     public void close() {
         maybeUnregisterObserverProxy();
-        mCursor.deactivate();
-        
+        mCursor.close();
     }
     public int requery(IContentObserver observer, ...
```

Generated Message:
```
“CursorToBulkCursorAdapter . Close must call mCursor . Close instead of mCursor . Deactivate . ”
```

Reference Message:
```
“Call close ( ) instead of deactivate ( ) in CursorToBulkCursorAdaptor . close ( ) ”
```

Scores: 7, 6, 7
Figure 4.11. The predict results of the cross evaluation of QA filter. QA filter reduced 108 messages that are scored 0, 45 messages that are scored 1, 42 messages that are scored 2, 32 messages that are scored 3, 32 messages that are scored 4, 21 messages that are scored 5, 18 messages that are scored 6, and 16 messages that are scored 7. We note that although we trained the QA filter with binary labels, “bad” and “not bad”, the evaluation result shows that QA filter is able to reduce more messages for lower scores.

4.7.2 More Examples

In this section, we will discuss in which cases our NMT model performs well, and in which cases it performs badly. First, we examine the good cases where the generated commit messages satisfy both the following conditions:

- The scores in the human-evaluation study are no lower than 5.
- The BLEU scores are no lower than 50.

There are 175 generated commit messages that satisfy the above conditions. Some examples of the changes of these commits are shown in the Table 4.4. We found most changes are short. All the changes in Table 4.4 involve only one or two lines. Not all the changes are in the source code. For example, Change 5 is a binary file change and the diff file lists only the changed file name. Change 6 is removing a tag of a Mercurial repository.

1Mercurial: https://www.mercurial-scm.org/
### TABLE 4.4

**THE CHANGES OF THE RESULTS THAT HAVE HIGH SCORES**

<table>
<thead>
<tr>
<th>change id</th>
<th>change</th>
</tr>
</thead>
</table>
| 1         | - import groovy.lang.Closure;  
|           | - import groovy.lang.GroovyObject; |
| 2         | - if (isTaken()) {  
|           | + if (robotTaking() != null) { |
| 3         | - versionCode = 15043  
|           | - versionName = '18.0.0'  
|           | + versionCode = 15044  
|           | + versionName = '18.0.1' |
| 4         | + def removeErrorListener(self, listener):  
|           | + self._listeners.remove(listener) |
| 5         | Binary files a/lib/native/linux-x86.jar and b/lib/native/linux-x86.jar differ |
| 6         | + 2b581c93 flyway-1.6  
|           | + 00000000 flyway-1.6 |
Then, we check the cases where the generated commit messages have low scores in both BLEU metric and the human-evaluation study. In Table 4.5, we show four changes for which the generated commit messages have scores of zero or one in the human evaluation study and have BLEU scores of zeros. In some cases, the generated commit messages may be correct but are different from human-written messages. For example, the generated message for Change 7 is "updated changelog". This is correct because this change is in "CHANGELOG.md". The scores for the generated message are low in BLEU and in the human evaluation study because the human-written message is more specific about the update of the changelog: ‘fix snapshot version’. The other three examples in Table 4.5 show that some diff files contain changes in two files, which may contribute to the low performance of the NMT model on these cases.

4.8 Threats to Validity

One threat to validity is that our approach is experimented on only Java projects in Git repositories, so they may not be representative of all the commits. However, Java is a popular programming language [1–3], which is used in a large number of projects. In the future, we will extend our approach to other programming languages.

Another threat to validity is the quality of the commit messages. We collected actual human-written commit messages from Github, and used V-DO filter to obtain a set of relatively good-quality commit messages. But the human-written messages may not contain all the useful information that should be in a commit message. And programmers are often lazy and write poor messages [39]. However, our objective in this chapter is to generate commit messages that can be learned from the history of the repositories. Further improvement on human-written messages falls outside the scope of this chapter.
<table>
<thead>
<tr>
<th>change id</th>
<th>change</th>
</tr>
</thead>
</table>
| 7         | - #2.2.0 (16/07/2015)-SNAPSHOT  
           | + #2.1.1 (29/02/2016)-SNAPSHOT  |
| 8         | deleted file: AndroidClock.ttf  
           | deleted file: AndroidClock2.ttf |
| 9         | In .gitignore:  
           |     + reference/.metadata/  
           |     + reference/antlr4.net/  
           | In antlr4:  
           |     - Subproject commit 8865...b6519  
           |     + Subproject commit 878b...ce61 |
| 10        | In classObject.kt:  
           |     - namespace foo  
           |     + package foo  
           | In innerClass.kt:  
           |     - namespace foo  
           |     + package foo |
Another threat to validity is about the human study because of the limited number of the participants. We cannot guarantee that every final score for a generated commit message is fair. We tried to mitigate this threat by hiring as many professional programmers as we can, and having 23% of the evaluated messages scored by three participants and 53% of the evaluated messages scored by two participants.

Additionally, our use of Stanford CoreNLP \cite{111} may not yield the best performance of Stanford CoreNLP. Because Stanford CoreNLP is trained on natural languages, CoreNLP may not perform well on the commit messages, which contain special identifiers, such as variable names, and software programming jargon. Several studies in sentiment analysis for software engineering \cite{79,128} have discussed the problem of using NLP tools without customization for software engineering. In our future work, we may work on this problem so that we can provide NMT with better training data.

4.9 Discussion and Conclusion

The key advancement that this chapter makes to the state-of-the-art is a technique to generate short commit messages that summarize the high-level rationale for a change to software. As we note in Section 4.1.1, we do not claim to be able to provide new insights for completely novel changes to software—that task is likely to remain in human hands for the foreseeable future. Instead, we learn from knowledge stored in a repository of changes that have already been described in commit messages. Several authors in the related literature have observed that many code changes follow similar patterns, and have a similar high-level rationale (e.g., \cite{75,122}). Traditionally programmers still need to manually write commit messages from scratch, even in cases where a commit has a rationale that has been described before. What this chapter does is automate writing commit messages based on knowledge in a repository of past changes.
Our strategy was, in a nutshell, to 1) collect a large repository of commits from large projects, 2) filter the commits to ensure relatively high-quality commit messages, and 3) train a Neural Machine Translation algorithm to “translate” from diffs to commit messages using the filtered repository. We then evaluated the generated commit messages in two ways. First we conducted an automated evaluation using accepted metrics and procedures from the relevant NMT literature (Section 4.4). Second, as a verification and for deeper analysis, we also conducted an experiment with human evaluators (Section 4.5).

What we discovered is that the NMT algorithm succeeded in identifying cases where the commit had a similar rationale to others in the repository. The evidence for this is the large bar for item 7 in Figure 4.9—it means that the human evaluators rated a large number of the generated messages as very closely matching the reference messages. However, the algorithm also generated substantial noise in the form of low quality messages (note the large bar for item 0). A likely explanation is that these include the cases that involve new insights which the NMT algorithm is unable to provide. While creating these new insights from the data is currently beyond the power of existing neural network-based machine learning (a problem observed across application domains [56]), at a minimum we would like to return a warning message to the programmer to indicate that we are unable to generate a message, rather than return a low quality message. Therefore we created a Quality Assurance filter in Section 4.6. This filter helped reduce the number of low quality predictions, as evident in the reduced bar for item 0 in Figure 4.11.

While we do view our work as meaningfully advancing the state-of-the-art, we by no means claim this work is definitive or completed. We release our complete data set and implementation via an online appendix (Section 4.10). Our hope is that other researchers will use this data set and implementation for further research efforts. Generally speaking, future improvements are likely to lie in targeted training for
certain types of commits, combined with detection of change types. It is probable that very high quality predictions are possible for some types of software changes, but not others. This chapter provides a foundation for those and other future developments.

4.10 Reproducibility

Our data sets, scripts, and results are accessible via: https://sjiang1.github.io/commitgen

4.11 Acknowledgments

This work was partially supported by the NSF CCF-1452959 and CNS-1510329 grants, and the Office of Naval Research grant N000141410037. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.
CHAPTER 5

TOWARDS AUTOMATIC GENERATION OF TOPIC MODEL LABELS USING NEURAL MACHINE TRANSLATION

5.1 Introduction

Topic model algorithms such as Latent Dirichlet Allocation (LDA) and Relational Topic Modeling (RTM) are heavily used across software engineering research [11, 18, 54, 149, 156, 157, 175, 177], in particular in program comprehension [14, 115, 156]. Generally speaking, they are used to create groups of conceptually-similar software artifacts (the “topics”) that are independent of the structure of the software program (such as the classes). For example, one classic application is to discover what features a program implements (each topic is assumed to represent a feature) [46, 115].

In the standard application of topic models in SE, a “topic” is really just a probability distribution over a set of text words extracted from the software. That is a powerful analysis tool, but it is very difficult to interpret how the topic connects to the high-level concepts behind the code. A topic might represent a particular feature in the software, but there is a large intellectual leap to be made from the probability distribution to the concept of the feature. Readers may see a topic, but are left to wonder “but what does the topic mean?”

The solution to this problem is to generate a “label” for each topic. A topic label is a short description of a topic generated by a topic model algorithm [95, 121]. The typical strategy for creating a label is to just use the top-$n$ words from the topic’s probability distribution. Sometimes this works well e.g. a programmer might
deduce that “database sql dbi connect” refers to the parts of the program that issue database queries. Just as often though, the topic labels are extremely cryptic – it is quite difficult to determine that “name jedit pre property color” actually refers to the part of JEdit that “returns jedit’s context menu configuration” without very extensive inside knowledge of the program. The above are actual examples found by McBurney and McMillan [115] in a study on topic model labeling. That study evaluated several existing approaches based on sentence selection and topic modeling, and determined that existing labeling strategies are inadequate for program comprehension and are very far away from being usable by practitioners.

In this chapter, I use Neural Machine Translation (NMT) to automatically create topic labels. In a traditional NMT application, the algorithm is trained on pairs of sentences in one natural language matched to equivalent sentences in another language. Our application is quite different. First, we used LDA to generate keyword lists for every class in a set of Java projects. Second, we extracted one sentence to describe each class from the JavaDocs for those classes. Then we trained the NMT algorithm to “translate” the word lists for the classes to the sentence descriptions of the classes. The result was an NMT model which can create sentence descriptions given word lists. In application, we use the top-$n$ keywords from the LDA topics as input to the NMT model we created. The output was English sentences that, we intended, would describe the topics and serve as topic labels. In this chapter, to examine if the NMT approach has different performance in different domains, we trained two NMT models for two domains of Java projects: Web applications and Android applications.

To evaluate the generated topic labels, we conducted a survey with human experts to follow best practice in evaluations in the Software Engineering domain. The survey results show that the labels generated by our NMT approach are more helpful for programmers to understand Java projects than the typical labels, which are lists of
words. Our results also show that the programmers tend to think the labels generated by our approach are different from the typical labels. We sampled and analyzed some examples and found that the typical labels tend to be general and the labels generated by our approach tend to be specific.

Naturally, our results were limited by an assumption we made that class descriptions are similar to topic labels for projects. In the future, we can improve our approach by 1) finding class descriptions that are more appropriate for labeling project topics, and 2) hiring human experts to manually create topic labels for projects.

5.1.1 The Problem

This chapter targets the problem of generating software feature descriptions by automatically labeling topics generated by topic modeling algorithms, such as Latent Dirichlet Allocation (LDA). Topics are abstract concepts that can be used as summaries or features of software artifacts. By reading the topics, programmers or users can quickly gain general knowledge about software artifacts without reading implementation details. The most common strategy of labeling topics is to use a list of keywords to represent a topic. However, it is often difficult for programmers to understand a list of words, because they need to make an effort to find the connections among the words.

In this chapter, our hypothesis is that the keywords (that are used to represent the topics for software projects) are often generic and the connections among the words can be learned. We also assume that class comments can be used to interpret these words. Therefore, given a large number of pairs of topic keywords and the corresponding class comments, a neural network can be trained to generate comment-like topic labels.
5.1.2 Roadmap

In the rest of this chapter, we will list the background knowledge about topic models, LDA, and NMT in Section 5.2. We will describe our approach, which includes the data preparation and the settings of LDA and NMT in Section 5.3. In Section 5.4, we will present the LDA topics that we obtained. Finally, we will describe our evaluation and our results in Sections 5.5.3 and 5.6.

5.2 Background: Topic Models and LDA

Topic modeling and neural machine translations are two main technologies that are background of this chapter. Neural machine translation is introduced in Chapter 4 Section 4.2.2. In the rest of this section, we will introduce topic models.

Topic models, such as LSA [44], pLSI [72], LDA [20], and their extensions, such as Relational Topic Modeling [33, 53], are exploratory tools for understanding large corpora by revealing their underlying topics. Because the number of topics in a topic model for a corpus is relatively small, users can quickly grasp the main themes of the corpus by reading the topics.

Taking LDA as an example, the input of the model is a corpus with $D$ documents and $V$ word types. LDA has three parameters that need users to specify: $K$, $\alpha$ and $\beta$. $K$ is the number of topics; $\alpha$, $\beta$ are hyper-parameters controlling how topics are learned [106, 163, 169].

The output of LDA is a set of topics $\{\phi_k\}_{k=1}^K$. Each topic is defined as a word distribution, $\phi_k = (\phi_{k,1}, \cdots, \phi_{k,V})$, where $\phi_{k,v}$ is a probability of a word $v$ in topic $k$. To label a topic $\phi_k$, the common strategy is to output a list of words with top probabilities in $\phi_k$ [20, 22, 58]. LDA also outputs a weight for each topic, which is determined by how many times the topic appears in the corpus. The topics can be ranked according to their weights.
For each document $d$ in a corpus, LDA can also output the document-level topic distribution $\theta_d = (\theta_{d,1}, \cdots, \theta_{d,K})$, where $\theta_{d,k}$ represents the probability of topic $\phi_k$ in the document $d$. Document $d$’s dominant topic is the topic that has the largest probability, i.e., $k^*_d = \arg\max_k \theta_{d,k}$. Bhatia et al. conducted an empirical study which points out that among the existing topic models, LDA can output the best document-level topics by human evaluation [17].

5.3 Approach

The objective of our approach is to create topic labels for topics generated for a software project. In a nutshell, our approach works in four steps: 1) download the software repositories. Then, 2) extract class descriptions from Java files, seen in Figure 5.1 area 1. Then, 3) run LDA on software projects and get keywords for each Java file, seen in Figure 5.1 areas 2-4. Finally, 4) train an NMT with the pairs of keywords and class descriptions, seen in Figure 5.1 area 5. The product of our approach is the trained NMT, which generates description labels for topic keywords.

Figure 5.1. An overview of our approach, which we describe in Section 5.3. The upper layer (areas 1, 4 and 5) is the training process. The lower layer (areas 2, 3 and 6) is the process of creating topic labels after training.
5.3.1 Collecting Project Repositories

To include as many projects as we can and at the same time, to have a quality control on the collected projects, we downloaded all the Java projects that have at least 100 stars in Github\(^1\) which results in 9K projects. Among the 9K projects, we excluded 3,692 projects that have no more than 20 Java files because LDA does not perform well on corpora with a small number of documents \([101, 127]\). In total, we ran LDA on 5,448 Java projects.

5.3.2 Extracting Class Descriptions

In this subsection, we describe how we extracted the class descriptions from a Java file, seen in Figure 5.1 area 1. First, we extracted the class comment from the Java file. We define a class comment as the comments that 1) are before the class declaration and 2) are not about licensing. From the class comments, we removed any JavaDoc tag that contains the information that is not related to class functionalities, such as the authors of the classes.

Then, we kept only the first sentence in each class comment, because the first sentence should be a short description of the class based on to the Javadoc guidance.\(^2\) This practice is similar to what is done by Gu et al. to extract summaries for methods.\(^3\) Finally, we performed CamelCase split\(^3\) and lemmatization\(^4\) on the class comments.

After we extracted the class descriptions, we have two files for each Java class file:

---

\(^1\) The number of the stars of a project shows the number of Github users who are interested in this project.

\(^2\) [http://www.oracle.com/technetwork/articles/java/index-137868.html](http://www.oracle.com/technetwork/articles/java/index-137868.html)

\(^3\) We used a Perl package for CamelCase split: [http://search.cpan.org/dist/String-CamelCase/lib/String/CamelCase.pm](http://search.cpan.org/dist/String-CamelCase/lib/String/CamelCase.pm)

\(^4\) We used Stanford CoreNLP \([111]\) to perform the lemmatization.
a class description file that contains the first sentence of the class comment, and a class file that have no class comment, which will be referred to as “a filtered class” in the rest of this section.

5.3.3 Getting Keywords for Classes

In this subsection, we describe how we obtained keywords for each Java file, which corresponds to Figure 5.1 areas 2-4. There are three steps to obtain keywords for Java class files. First, we cleaned the class files. Second, we ran LDA on the class files (Figure 5.1 area 2). Third, we extracted keywords based on the topic distributions on the class files (Figure 5.1 areas 3 and 4). We will describe these steps in the following subsections.

5.3.3.1 Data Preparation for LDA

For LDA, we used filtered class files (seen in the previous section, Section 5.3.2) instead of the original class files. In real applications, LDA should be computed over the original class files. However, in our project, we assume the worst-case scenario where the class comments are not in the files, so our results are not affected by the quality of class comments. Following the steps done by the previous work using LDA in Software Engineering [36, 112, 166], we performed CamelCase split and lemmatization on the filtered class files. Then, we removed stopwords from the filtered class files.

We removed four kinds of stopwords: 1) all the English stopwords listed in NLTK [19], 2) the words that are about licensing, such as “copyright”, 3) the Java and JavaDoc keywords, and 4) the words that have high-frequency but are too generic to contribute to a topic, such as “get” [112]. The list of all the stopwords are available in our online appendix in Section 5.8.
5.3.3.2 Configuration for LDA

To run LDA on software artifacts, we followed the practice done by Mahmoud et al. [110] and Maskeri et al. [112] that we treated each project as a corpus and each class file as a document. Because we need to run LDA on more than 5K projects and the LDA training process is often time consuming, we chose SparseLDA [178] to speed up the LDA training process.

LDA has three parameters: $K$ (the number of topics), $\alpha$, and $\beta$, which are described in Section 5.2. As these parameters largely affect the quality of the topics [115, 132], we followed the guideline provided by Tang et al. [163] and set $\alpha$ as 0.1 and set $\beta$ as 0.001. Although Panichella et al. has proposed LDA-GA to use genetic algorithm to automatically find near-optimal settings for software artifacts [132], we did not use this approach because the genetic algorithm costs too much time to run on every Java project.

To find a reasonable $K$ for each Java project, we had the following three steps. For a Java project, first, we ran LDA with three different $K$ values: 20, 50, and 100. As a result, we have three sets of topics for each Java project. Second, we computed normalized pointwise mutual information (NPMI) [21] on each set of topics. Third, for each project, we picked the highest NPMI score and use the corresponding topics as the topics of the project. We chose NPMI for evaluating the topics because NPMI is a widely-used metric that measures the quality of a topic distribution [7, 21, 60, 127, 133]. We tried Silhouette Coefficient [132] to evaluate our topics but according to our manual inspection, NPMI scores are more correlated to the quality of topics in our data set.

5.3.3.3 Keywords for Each Java File

As mentioned in section 5.2 for a corpus, i.e., a Java project, the output of LDA has two parts: a set of topics $\{\phi_k\}_{k=1}^K$, and a set of document-level topic distributions
\(\{\theta_d\}_{d=1}^{D}\). Each document \(d\) corresponds to a Java file.

We extracted the keywords for each Java file \(d\) in three steps. First, we get the dominant topic \(\phi_{k_d}\) according to the document-level topic distribution \(\theta_d\). Second, we obtain the top-10 words for the dominant topic \(\phi_{k_d}\) according to the word probabilities. Third, we use the obtained top-10 words as Java file \(d\)’s keywords.

5.3.3.4 Running Details

We used Mallet \cite{Mallet} in our training process of LDA with three threads on an Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz. Mallet implements SparseLDA \cite{SparseLDA}, which is a speed-up version of the original LDA. The average computation time for LDA on a Java project is 34 seconds. For 5,448 Java projects, we ran LDA on each project three times (each time with a different \(K\) value), so the total computation time is about 6.5 days.

5.3.4 Training NMT Models

In this section, we describe how we trained NMT models, which corresponds to Figure 5.1 area 5. We chose Nematus \cite{Nematus}, an implementation of the attentional RNN Encoder-Decoder algorithm, as our translation model, because Nematus produces stable results for eight translation directions (e.g., from English to Russian), and Nematus has been used for translating non-natural languages in Chapter 4 \cite{Chapter4}.

We followed a same training process for training two different NMT models. One NMT model was trained on Web projects and the other model was trained on Android projects. In the following subsections, first, we will describe how we classified a Java project as a Web or an Android project. Second, we will describe how we prepared data files for NMT training. Then, we will describe our configuration for NMT training and running details. Finally, we will discuss how NMT models were applied on our test sets, which corresponds to Figure 5.1 area 6.
5.3.4.1 Projects of Two Domains

We followed three steps to identify Java projects as Web or Android applications.

First, we defined a relevant keyword list: \{android, http, url, socket, json, servlet, web, html\}. Second, for each project, we counted the total occurrences of the relevant keywords in the topic keywords of the project. For example, we assume a project has 20 topics, and each topic has a list of top-10 words, which means the project has 200 topic keywords. If “android” occurs 10 times and “http” occurs 10 times in the topic keywords and there is no other relevant keyword, the total number of the occurrences of the relevant keywords is 20.

Third, we set a threshold of 20 for the occurrences of the relevant keywords. If the relevant keywords appear in the topic keywords of a project more than 20 times, the project is identified as either a Web or an Android project. Otherwise, the project is considered as neither a Web nor an Android project. Next, if the project passes the threshold, we check the occurrences of the word “android”. If “android” occurs more than 10 times, we identify the project as an Android project. Otherwise, the project is a Web project. We note that in this step, we treat a project as either Web or Android-based. In real world applications, there are projects that are both Web and Android-based. However, in this study, Web projects are defined as the projects that are Web-based but not Android-based.

We followed the above steps on the 5,448 Java projects (described in Section 5.3.1), and we found 578 Web projects and 1,135 Android projects. We randomly sampled and checked 30 projects from the Web projects and 30 projects from the Android projects. All the sampled projects were confirmed as correctly identified.

5.3.4.2 Data sets for NMT

To train an NMT, we need pairs of source sequences and target sequences. In our study, a source sequence is the list of keywords for a Java file (described in
Section 5.3.3.3). The corresponding target sequence is the class description of the same Java file (described in Section 5.3.2). Note that we did not include comments from Java test files because these comments typically are not suitable as topic labels.

The length limit for the source sequences is 10 (according to Section 5.3.3.3). For the target sequences, we also set 10 as the length limit, because most class descriptions contain less than 10 words.

For Web domain, we have 44,269 pairs of source and target sequences. For Android domain, we have 38,025 pairs of source and target sequences. For each domain, we randomly picked 3k pairs for testing, 3k pairs for validation. For Web domain, 38,269 pairs were used for training. The vocabulary size for source sequences is 7.7k and the vocabulary size for target sequences is 6.5k. For Android domain, 32,025 pairs remained for training and in the training set, the vocabulary size for source sequences is 8k and the vocabulary size for target sequences is 6k.

5.3.4.3 Configuration for NMT

We used the configuration that Sennrich et al. used in ACL 2016 First Conference on Machine Translation (WMT16) [151]. The number of hidden layers is set to 1024. The size of word embeddings is set to 512. Cross-entropy minimization [146] is used as the training goal. Stochastic gradient descent (SGD) with Adadelta [180] is used as the learning algorithm, and the size of minibatches is set to 80. The training will stop when the number of epochs exceeds 5k or the number of minibatches exceeds 10M or the early stopping on cross entropy is satisfied.

The model is saved every 30k minibatches. For example, if a training process stopped at 240K minibatches, 8 models will be saved. In application, we run the last four models and use the ensemble results of the four models.

We excluded all the files that have file names starting or ending with “test.”
5.3.4.4 Running Details

We trained the NMT model for Web domain on an GeForce GTX 1070 with 8GB memory. The training process took 25 hours and 2 minutes and terminated at 210K minibatches with 7 models saved. We trained the NMT model for Android domain on an Nvidia GeForce GTX 1080 with 8GB memory. The training process took 14 hours and 45 minutes and terminated at 210K minibatches with 7 models saved.

5.3.4.5 Application of NMT

We applied the NMT models on the corresponding test sets, which were created in Section 5.3.4.2. The test sets are two sets of source sequences, which are keywords for Java files. As described in Section 5.3.3.3, the keywords for a Java file are top-10 words of a topic. Given a list of keywords, the trained NMT models output a description for the keywords. For example, the topic mentioned in Section 2.6, “socket web websocket jetty eclipse factory upgrade api policy driver”, is one test input for our NMT model, and the output of the NMT model is “factory to create web socket connection object”.

Note that we did not compare our test output to the extracted class descriptions, because the class descriptions are not our gold set of topic labels. That is, if a generated description is not similar to the class description, this does not mean that the generated description is not a good topic label. On the other hand, if a generated description is similar to the class description, it does not indicate the description is a good topic label. Therefore, instead of comparing the generated labels with the class descriptions, we conducted a survey with human experts to evaluate the labels.
5.4 Intermediate Results: LDA Topics

In this section, we will describe the topics we generated by LDA for the Java projects, because as the inputs of NMT, the quality of LDA topics affects the effectiveness of our approach.

5.4.1 Evaluation Metric

Since LDA is run on more than 5K Java projects with three different settings, there are about 1 million topic keyword lists in total. It is impossible to check every output of LDA manually, so we used an automatic metric, NPMI [7], which has been verified to have a strong correlation with human ratings on evaluating topics [143]. The NPMI metric on topic keywords is an extension of the NPMI\_pair metric for two words [21], shown as Equation (5.1) and (5.2).

\[
NPMI\_pair(a, b) = \frac{\log \frac{P(a,b)+\epsilon}{P(a)P(b)}}{-\log(P(a,b) + \epsilon)} \tag{5.1}
\]

\[
NPMI\{(w_1, w_2, \cdots, w_n)\} = \frac{2}{n(n-1)} \sum_{i=2}^{n} \sum_{j=1}^{i-1} NPMI\_pair(w_i, w_j) \tag{5.2}
\]

In Equation (5.1), \(P(a)\) means the probability of word \(a\) that appears in a reference corpus. To avoid 0 in numerator, a small \(\epsilon = 10^{-3}\) is added. An NPMI score ranges from -1 to 1. And a large score indicates the high quality of a learned topic.

For each topic, we have a list of top-10 words, which were described in Section 5.3.3.3. And for each topic, we computed an NPMI score on the top-10 words. For each Java project, the NPMI of the project is the mean of NPMI scores on all the topics. Note that we used the local corpus (the Java project on which the topic model runs), as the reference corpus for NPMI, instead of all the Java projects. The
trade-off was taken based on the following two reasons: (1) considering 1 million topics to be evaluated, running NPMI on local corpora is more efficient, and (2) the NPMI metric computed on local corpora is correlated with the human ratings [143].

5.4.2 Overall Results

There are 3 settings of $K$: \{20, 50, 100\} for each Java project. We chose the best $K$ for each project in terms of NPMI. The overall results of topic quality is shown in Figure 5.2. The NPMI of all the projects, Web projects, and Android projects are $0.442 \pm 0.123$, $0.428 \pm 0.116$, and $0.481 \pm 0.108$ respectively. They share similar distributions as having very close standard deviations, while the mean of NPMI of Android projects is slightly higher than the mean of NPMI for all the projects.

We sampled 5 topics from 5 NPMI score groups in Table 5.1 in order to show the topic quality of different NPMI scores. Note that the listed NPMI scores are project-level NPMI scores, and we picked the dominant topic for each project.

5.5 Evaluation

This section describes our evaluation of the topic labels that we generated. We compare our NMT-based approach to the typical top-$n$ words labeling approach.

![Figure 5.2](image-url)  

Figure 5.2. The overall quality of learned topics of all projects, Web projects, and Android projects in terms of NPMI. They share similar distributions.
5.5.1 Research Questions

Our research objective is to determine whether our approach helps programmers understand what a topic means in the context of comprehending a software project, versus a popular baseline approach. Towards that end, we ask the following Research Questions (RQs):

*RQ*₁ How well does the approach, versus the baseline, help programmers understand what a software project does?

*RQ*₂ How much different are the labels generated by our approach from the baseline labels?

*RQ*₃ Does our approach perform differently in Web domain and in Android domain?

The rationale behind *RQ*₁ is that, by comparing our topic labeling approach with a baseline approach, we can learn in what cases our approach can be useful and under what circumstances our approach does not perform well and need improvement. We intend *RQ*₁ to measure the performance of our approach and the baseline approach, in the context of comprehension of a software project. A typical application of topic model labels is to assist programmers while they try to understand the behavior of a program, i.e. what the program is for and what it does. As mentioned in Section 5.1 the idea is that topic model labels should help programmers rapidly gain knowledge about software; it does not imply a deep reading of the code. Therefore, in *RQ*₁, we seek to compare our approach and the baseline in the context of rapid comprehension.

The labels generated by our approach are naturally different from the baseline labels because our approach aims to generate sentences and the baseline labels are lists of words. However, we ask *RQ*₂ because we want to assess and analyze whether the programmers can see the similarity between the labels from the baseline and our approach. There are many potential reasons that why the programmers do not see the similarity between the two types of labels. For example, maybe the two labels are actually about different topics. By analyzing the survey responses, we can understand
in what cases, the programmers think the two labels are similar and in what cases, the programmers think the two labels are different.

The rational behind RQ$_3$ is to determine if our approach perform differently in different domains. In our study, we applied our approach to two domains, Web and Android domains. There are potential differences between the two domains that may affect the performance of our approach.

5.5.2 Baseline

For a topic $\phi_k$, where $k$ is in $(1, ..., K)$ and $K$ is the number of topics for each Java project, the baseline approach we used is to pull the top-$10$ words with the highest probabilities in $\phi_k$. This approach is widely used in topic modeling community [20, 22, 32, 58] and software engineering community [11, 36, 115]. In this chapter, we refer to this approach as the top-$n$ words labeling. We did not use the sentence-extraction approach as our baseline because the existing techniques are found unhelpful for program comprehension [115]. Additionally, we did not use the re-ordering method [149, 154] for the top-$n$ words labeling, because this method did not perform well in our preliminary study.

5.5.3 Methodology

Our methodology to answer our RQs is to conduct a survey. In the survey, we asked human programmers to read topic labels in the context of the software project from which those labels were generated. And we asked the programmers to answer questions about those labels corresponding to RQ$_1$ and RQ$_2$. Figure $5.3$ shows a screenshot of the survey, which we implemented as a web interface. We will describe the process of the survey study as follows.

First, we sent every participant an invitation email with a brief introduction and detailed instructions with the survey link. The exact content of the invitation email
is at our online appendix in Section 5.8. For each participant, the web interface would choose a project for the participant to start. The participant was instructed to read the Github profile of the software project (the README and brief navigation of the code) to gain an idea of what the project does. The web interface also showed the participant five topic labels for that project. The web interface would decide whether to show labels from the baseline, or to show labels from our approach.

We asked the participant to rate the sentence “I believe this description helps me understand what the project does/is for” on a scale from 1-4, where 1 was Strongly Disagree, 2 was Disagree, 3 was Agree, and 4 was Strongly Agree. There is an extra option “cannot tell”, which could be chosen when the participant could not decide. After the survey, we used these ratings to answer RQ1. To be consistent with the objective in which topic modeling is often used, we did not specify a goal beyond obtaining a high-level concept of what tasks are implemented in the project.

Once the participant rated the five topic labels we showed and clicked submit, the web interface showed the participant the topic labels from both approaches. That is, if the interface had showed five labels from our approach, it would then show the same five labels from our approach, each label accompanied with a corresponding label from the baseline approach. For each pair of the labels, the participant then

![Figure 5.3. A screenshot of our survey page which asks participants whether the labels help them understand the projects.](image)

125
rated the similarity between the two labels on a scale from 1 to 4, where 1 was Completely Different, 2 was Not Similar, 3 was Similar, and 4 was Identical. Due to space limit, we put the example of this page at our online appendix in Section 5.8.

5.5.4 Survey Setup

5.5.4.1 Subject Applications

We selected the Java projects that have at least five topics in the test sets, which were described in Section 5.3.4.2. In total, we have 222 subject projects. There are 124 projects are from Web domain and 98 projects are from Android. We sorted the 222 subject projects in a random order, which is the input of the web interface. We note that in this survey, not all the subject projects got evaluated.

We selected only the projects that have at least five labels because we want the participants at least rate five labels for each project. For a project that have more than five topics in the test set, we selected the five topics that have the largest weights (described in Section 5.2).

5.5.4.2 Participants

We hired 30 participants, each of them is either a graduate student majoring computer science or has at least one year of professional programming experience. We paid them USD$30 each for 30 minutes.

5.5.4.3 Assigning Subject Projects to the Participants

In this evaluation, we have 222 subject projects, \((S_1, S_2, ..., S_{222})\). Each project \(S_i\) has five topics, and each topic has two types of labels: the baseline labels \(\text{keywords}(S_i)\) and the labels generated by our approach, \(\text{nmt}(S_i)\). To avoid potential bias of the participants, we did not let the participants rate both types of the labels for one project for RQ_1. So we designed two separate task lists for the participants. One
task list is $T_1 = (\text{keywords}(S_1), \text{nmt}(S_2), \text{keywords}(S_3), ...)$ and the other task list is $T_2 = (\text{nmt}(S_1), \text{keywords}(S_2), \text{nmt}(S_3), ...)$). We have 30 participants ($p_1, ..., p_{30}$), and we assigned $p_1, p_3, ..., p_{29}$ to do $T_1$ and $p_2, p_4, ..., p_{30}$ to do $T_2$. For example, if $p_1$ has evaluated $(\text{keywords}(S_1), \text{nmt}(S_2), \text{keywords}(S_3))$, $p_3$ would start from $\text{nmt}(S_4)$. If $p_2$ has evaluated $(\text{nmt}(S_1), \text{keywords}(S_2))$, then $p_4$ would start from $\text{nmt}(S_3)$. In this case, Project $S_1$ is evaluated by $p_1$ and $p_2$, and we compare $p_1$’s score on $\text{keywords}(S_1)$ with $p_2$’s score on $\text{nmt}(S_1)$. For Project $S_2$, we compare $p_2$’s score on $\text{keywords}(S_2)$ with $p_1$’s score on $\text{nmt}(S_2)$. We note that in this setup, for RQ$_2$, one pair of the labels would be evaluated by two participants.

5.5.5 Threats to Validity

The key threats to validity of our study are: 1) our selection of the subject applications, and 2) the participants in our study. While we attempted to mitigate the first risk by using a large dataset of popular software applications, it is still possible that the results of our experiments would change if we used different applications. Several human factors influence the behavior of our participants, including fatigue, prior experience, and education. We attempted to mitigate this risk by recruiting 30 participants with a diverse histories, but again, it is possible that our study results would change with different participants.

5.6 Results

In this section, we present our results of the survey study. In the following subsections, we will present the statistics of the collected responses, the results for RQ$_1$, RQ$_2$, and RQ$_3$. For convenience, the labels generated by our approach will be referred to as “our labels” in this section.
5.6.1 Collected Survey Responses

In this study, we collected responses from 30 participants. For 84 projects, both the baseline labels and our labels were evaluated. We excluded the subject projects with incomplete evaluation results. For example, if a project is evaluated by only one participant, which means that only one type of labels (the baseline labels or our labels) were evaluated, we exclude the project from our analysis. On average, each participant evaluated 10 projects in 30 minutes.

For the 420 topics, 37 baseline labels and 52 labels generated by our approach were marked as “cannot tell”, which means that the participants cannot determine if the labels can help them understand the subject projects. For three projects, the participants scored all our labels as “cannot tell”. For two projects, the participants scored all the baseline labels as “cannot tell”. In these five projects, one of the project’s readme file is in Chinese so the participant did not understand the project. So we removed the results for this project. In the rest of four projects (that have either all the baseline labels or all our labels marked as “cannot tell”), three projects were evaluated by the same participant. In the end, the results of 415 topics remained for our analysis.

5.6.2 RQ1: Whether the Labels Help Programmers Understand Subject Projects

From the survey results, we found that our labels are more helpful than the baseline labels for programmers to understand Java projects. Specifically, we examined the scores that the participants gave on whether they agree that the given labels are helpful for understanding the subject projects. The distributions of the scores for the two types of labels are shown in Figure 5.4a. The statistics on RQ1 are shown in Table 5.2.
Figure 5.4. Distributions of Scores on Helpfulness. Score 1 means the participants strongly disagree that the given label was helpful. 2 means “disagree”; 3 means “agree”; and 4 means “strongly agree”. In all the sub-figures, most baseline labels have scores of 2; most our labels have scores of 3.
TABLE 5.1

EXAMPLES OF TOPIC KEYWORDS

<table>
<thead>
<tr>
<th>NPMI</th>
<th>Example of topic keywords list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-0.2</td>
<td>(<a href="https://github.com/gaohannk/Leetcode.git">https://github.com/gaohannk/Leetcode.git</a>) tmp iterator palindrome leetcode partition zigzag iter size subset solution (0.182)</td>
</tr>
<tr>
<td>0.2-0.4</td>
<td>(<a href="https://github.com/thymeleaf/thymeleaf-spring.git">https://github.com/thymeleaf/thymeleaf-spring.git</a>) spring variable thymeleaf template expression request environment allow fail bind (0.312)</td>
</tr>
<tr>
<td>0.4-0.6</td>
<td>(<a href="https://github.com/ac-pm/Inspeckage.git">https://github.com/ac-pm/Inspeckage.git</a>) view setting webkit enable access plugin check javascript chrome allow (0.507)</td>
</tr>
<tr>
<td>0.6-0.8</td>
<td>(<a href="https://github.com/forJrking/FontZip.git">https://github.com/forJrking/FontZip.git</a>) language str latin anonymous serbian finland france malay russia format (0.607)</td>
</tr>
<tr>
<td>0.8-1.0</td>
<td>(<a href="https://github.com/bluejamesbond/TextJustify-Android.git">https://github.com/bluejamesbond/TextJustify-Android.git</a>) view text document bluejamesbond sample layout activity instance create hyphenator (0.901)</td>
</tr>
</tbody>
</table>

* For each example, we list the project link, the top-10 keywords for the dominant topic, and the NPMI score for the project.
## TABLE 5.2

### THE STATISTICS OF THE HELPFULNESS SCORES FOR RQ$_1$ AND RQ$_3$

<table>
<thead>
<tr>
<th>Group</th>
<th>Descriptive statistics</th>
<th>Paired t-test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>approach</td>
<td>n</td>
</tr>
<tr>
<td>RQ1</td>
<td>baseline</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>ours</td>
<td>334</td>
</tr>
<tr>
<td>RQ3</td>
<td>baseline</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>ours</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>Android</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>ours</td>
<td>131</td>
</tr>
</tbody>
</table>

If the statistic $t$ is larger than $t_{1-α/2}(n-1)$, the hypothesis that two approaches are not different should be rejected. $t_{1-α/2}(n-1)$ is calculated at the significance level $α = 0.05$ and $n$ is the number of samples.

$^a$ We removed all the topics that have at least one “cannot tell” for the labels.

$^b$ **, *, and ns denote $p \leq 0.01$, $p \leq 0.05$, and $p > 0.05$ respectively.
Figure 5.4a shows that the distribution of the scores of our labels is a right-skewed distribution while the distribution of the scores of the baseline labels is left-skewed. A right-skewed distribution has a bigger mode than its mean, which is the case of our labels but not the case of the baseline labels. There are only 41.2% samples agree that the baseline labels are helpful (with score 3 or 4) for understanding the subject projects. In contrast, our approach increased this proportion by 12.5% to 53.7%.

After removing the rows which contains the ”cannot tell” values, we got 334 samples. On these samples, we performed the paired t-test, and identified a significant difference in helpfulness between our approach and the baseline ($t=2.93 > t_{1-0.05/2}(333)$, $p = 0.0036$).

In the following paragraphs, we will list two typical examples where our labels were rated as helpful or unhelpful.

The first example shows one of our helpful labels. This topic is from Project dropwizard which is a framework for RESTful web services. Both of the labels have scores of 3, which means that the participants found the labels helpful. The baseline label is about file web service, and our label is more specific.

baseline: path resource asset file uri index bundle servlet example trim

our label: resource implementation for java . Io . File handle

The second example is from Project GraphAware Neo4j Framework which is a framework for Neo4j development. The baseline label was rated as 4, while our label was rated as 1.

baseline: node graphdb graphaware neo common non hate rest spec expander

our label: serializable detach node with custom node id

In the above case, the baseline label captured the main topic of the project:

\[\text{http://www.dropwizard.io}\]

\[\text{https://github.com/graphaware/neo4j-framework}\]
“graphaware” and “neo”. However, our label is about implementation details, so the participant rated the label as 1. The reason why some of our labels are too detailed to be topic labels (like the label in this example) is that NMT is trained to generate class descriptions, and it is difficult to distinguish class descriptions that are general enough to be topic labels from the class descriptions that are too detailed. For example, this subject project is a framework designed to have a large number of utility functionalities for different purposes. Under this situation, it is more likely for NMT to generate topic labels that are similar to utility functionalities.

5.6.3 RQ$_2$: Similarity between the Labels from the Baseline and Our Approach

For RQ$_2$, we asked the participants to score the similarity between the baseline labels and our labels. For each pair of the labels, there are two scores given by two participants. For each pair that was evaluated, we take the average of the two scores as the score for the pair. The distribution of the scores for all the pairs is shown in Figure 5.5. The average score is 2.3, which means that in general, the participants tend to think that the baseline labels and our labels are different. While there are 202 pairs that have scores of 1 to 2, 119 pairs have scores of 3 to 4. We will discuss some examples in the following paragraphs.

![Figure 5.5](image.png)

Figure 5.5. The distribution of the similarity scores of 415 pairs of the labels. Each pair of the labels corresponds to a same topic. Score 1 means “completely different”; 2 means “different”; 3 means “similar”; and 4 means “identical”. There are 106 pairs of the labels scored as 2. In total, 213 pairs have scores that are larger than 2.
5.6.3.1 Disagreement among the participants

Among 415 pairs, the participants disagreed on 82 pairs of labels, where the difference between the two scores is at least 2. One possible reason for disagreement is that baseline labels tend to be general but our labels tend to be specific. Some participants think they are different, while others think they refer to one topic. For example, the baseline of the following pair of the labels is about web service request. In contrast, our label is about testing security context.

**baseline**: *json service user servlet http query request inject opt admin*

**our label**: *test explicitly specify the security context*

For the above labels, one participant rated as “identical” and another participant rated as “not similar”. One can argue that “test specify security context” is possibly about http query and admin authorization, so they are about the same topic. On the other hand, another argument can be that JSON service and web query do not necessarily mean “security context”, so they are not similar.

5.6.3.2 An example of a pair of completely different labels

There are 15 pairs of labels that were identified by both participants as “completely different”. We present one example as follows.

**baseline**: *non cgeo geocach support annotation nullable model lang common*

**our label**: *connector capability of search online for a cache by geocode*

The above example topic is from an Android project [c:geo](http://www.cgeo.org/). The project is about geocaching, which is a real-world treasure hunting game using GPS-enabled devices, like “Pokemon Go”. Even with this background knowledge, it is still difficult to understand the baseline label, which is basically a list of general words, with project-specific words, e.g., “cgeo” or “geocach”. Based on the baseline label, we do not
know what feature or functionality that this topic refers to. This difficulty of understanding baseline labels may be the reason why the participants determined that the baseline label and our label are completely different.

5.6.3.3 Examples of Pairs of similar labels

Among the 416 pairs of labels, 94 pairs of labels were considered as similar or identical by both participants. We list one example as follows.

**baseline**: socket nio client response selector write connect reconnect send read

**our label**: niotcp socket tcp version of nio socket

This topic is from a Web project *CommonCrawl*. For the above labels, both participants agree that the labels are similar. The reason may be that two important words, “nio” and “socket” in our label, are also in the baseline label. Compared to the previous examples, it is easier for the participants to determine that the labels are similar.

Another example is from another Web project, *Netty*, an asynchronous network application framework.

**baseline**: channel ctx handler netty close write read flush msg cause

**our label**: handler implementation for the echo peer

Both participants determined that the above labels were similar to each other. The reason may be that both labels are about “handler”. And the words in the baseline label, especially the words “close”, “write”, “read” and “flush” are all actions that often appear in implementation details, which matches our label “handler implementation”. However, our label also contains “for echo peer”, which cannot be matched back to the baseline label, so the participants do not rate them as identical.

9[https://github.com/commoncrawl/commoncrawl-crawler](https://github.com/commoncrawl/commoncrawl-crawler)

10[http://netty.io/](http://netty.io/)
5.6.4 RQ3: Differences between Web Domain and Android Domain

Based on the survey results, we found that the distributions of helpfulness scores of our labels are overall similar in Web projects and in Android projects. Figures 5.4b and 5.4c show the distributions of the helpfulness scores on Web projects and on Android projects, where the distributions on our labels in the two subfigures are both right-skewed. Table 5.2 shows that our labels in Group Web and Group Android have the same mode value of 3, both bigger than their mean values. However, our labels in Group Web has a higher mean than Group Android, which indicates that our approach may perform better in Web projects. Figure 5.4c shows that in Android projects, our labels that were rated as 4 are only two more than the baseline labels that were rated as 4. This is the main reason why the mean value of our labels in Group Android is lower than the mean value of our labels in Group Web.

We performed the paired t-test on each group. The p-value for the group of Web projects is 0.011, which indicates a significant difference between our labels and the baseline labels under Web domain. However, we did not find significance under Android domain. One possible reason is that for Android domain, we have fewer samples.

5.7 Discussion and Conclusion

The key advancement we made in this chapter to the state-of-the-art is an approach that generates natural language sentences to describe topics in topic models. In this chapter, we aim to learn knowledge from the class descriptions, which often are written by programmers as class comments. We assume that the knowledge in the class descriptions can be used to generate topic labels of software projects. Traditionally, a topic is represented by a list of keywords. However, it tends to be difficult to understand a list of words. Several researchers in related field have tried to use
extracted sentences as topic labels [115], but it is not successful.

Our strategy is to use neural machine translation to translate topic keywords into sentences. We discovered that NMT generated labels can be helpful for the programmers for understanding projects. Because topic keywords are often too general, NMT-generated labels may be favored because NMT can generate labels with concrete information learned from class descriptions. For example, NMT-generated label “factory to create web socket connection object” may be favored over the traditional keywords label “socket web websocket jetty eclipse factory upgrade api policy driver”.

While we see our work as a meaningful advancement, our work is not completed or definitive. One important area for improvement is that NMT-generated labels can be too specific containing lower-level implementation details which are not suitable for topic labels.

5.8 Reproducibility

For reproducibility, we provide our data files and results at

https://goo.gl/JCTPrW
CHAPTER 6

CONCLUSION

I have presented three projects on program comprehension in this dissertation. My overarching research goal is to improve automatic software documentation to help programmers understand software projects. Specifically, I have focused on automatic documentation of software changes and software feature descriptions.

First, I did an extensive empirical study on how programmers understand software changes. The study indicates that to understand changes in source code, programmers do change impact analysis in debugging differently from the methodology that the existing literature has suggested. The main difference is that programmers do not want extra information as long as they have an idea of modifying the code, which is consistent with the concept of “opportunistic” programming where programmers focus on solving problems quickly instead of code quality.

Based on this finding, the second project aims to generate short descriptions of software changes, which grasp only the key information about the changes. In this project, I introduced neural machine translation instead of rule-based methods to generate short commit messages. The existing rule-based methods tend to be verbose because high-level summaries are difficult to be defined by rules. This project has shown that neural machine translation can learn rules from history data to generate summaries of software changes.

The third project also aims to generate short descriptions—software feature descriptions. By treating each feature as a topic in the source code files of a software, I used neural machine translation to generate a natural language sentence to describe
a topic. This is the first project in the current literature to generate natural language feature descriptions automatically.

These projects presented in this dissertation are not definitive or completed. Instead, the projects lay the foundation for future work on high-level documentation of software projects.
BIBLIOGRAPHY


89. J. Krinke. Identifying similar code with program dependence graphs. In *Pro-
ceedings Eighth Working Conference on Reverse Engineering*, pages 301–309,

90. A. Lakhotia. Understanding someone else’s code: analysis of experiences. *Jour-

deep learning with information retrieval to localize buggy files for bug reports
(n). In *2015 30th IEEE/ACM International Conference on Automated Software

92. T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Pro-
ceedings of the 32Nd ACM/IEEE International Conference on Software En-

of developer work habits. In *Proceedings of the 28th International Conference

94. T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program com-
prehension as fact finding. In *Proceedings of the the 6th Joint Meeting of the
European Software Engineering Conference and the ACM SIGSOFT Symposium
on The Foundations of Software Engineering*, ESEC-FSE ’07, pages 361–370,

topic models. In *Proceedings of the 49th Annual Meeting of the Association for
Computational Linguistics: Human Language Technologies - Volume 1*, HLT ’11,
pages 1536–1545, Stroudsburg, PA, USA, 2011. Association for Computational
Linguistics.


How programmers debug, revisited: An information foraging theory perspective.
10.1109/TSE.2010.111.

98. D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What’s in a name? a study of
identifiers. In *14th IEEE International Conference on Program Comprehen-


144. T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference*


