ABSTRACT

GE, XI. Improving Tool Support for Software Developers through Refactoring Detection. (Under the direction of Emerson Murphy-Hill.)

With changing software requirements, developers need to maintain the source code of software systems to ensure their continuous usefulness. Maintaining software involves both the functional and nonfunctional improvements of the codebase. The functional improvements, such as adding new features and fixing defects, benefits external clients by enhancing or correcting the behavior of software systems. On the other hand, the nonfunctional improvements enhance the internal quality of the codebase, therefore increasing the longevity of the software under maintenance.

The nonfunctional improvements of codebase are also called refactoring. Existing studies show that refactoring improves the maintainability, testability and reusability of software systems. Therefore, refactoring becomes an essential and frequent activity of software developers. Developers can refactor their codebase manually. However, refactoring manually is both error-prone and time-consuming. To improve the correctness and efficiency of manual refactoring, researchers and developers implemented multiple refactoring tools. In addition, almost every mainstream Integrated Development Environment (IDE) integrates refactoring tools, exposing these tools to a large population of developers working in various programming languages.

However, refactoring tools are underused. Existing studies suggest that about 90% of refactorings were performed manually even though refactoring tools are available. To investigate the reason for this underuse problem, I conducted a formative study on 12 developers and found out that a developer sometimes starts manual refactoring without realizing so, thus missing the chance of invoking refactoring tools. To solve this late awareness problem, I integrate refactoring detection algorithms into the conventional refactoring engines, proposing a novel refactoring tool called BeneFactor that detects the start of a manual refactoring, reminds the developer that automated refactoring is available and finally finishes the started refactoring automatically.

Another reason for the underuse problem of refactoring tools is developers’ lack of trust on the changes that refactoring tools apply on their codebases. To solve this problem, I combined the refactoring detection technique with a static analysis technique to delegate refactoring support to the error messages in IDEs. The proof-of-concept tool, called GhostFactor, detects manual refactorings, checks conditions retrospectively and notifies the developer if it finds an error
in the performed refactoring. A controlled study with eight developers shows that GhostFactor improves the correctness of manual refactorings significantly.

Beyond helping developers refactor correctly and efficiently, the refactoring detection technique can also improve change comprehension. Code review is a commonly adopted activity among developers working on the same project. By examining each others’ code changes, code review can detect defects, correct bad programming practices and transfer project-specific knowledge among team members. However, during code review, developers have the different concerns about the refactoring and non-refactoring changes. To understand such a difference, I conducted a survey study among Gerrit developers, finding that refactoring changes slow down code review and make correct review harder. To solve this problem, I designed a refactoring-aware code review tool that allows developers to review the refactoring and non-refactoring part in a given change separately.

In summary, this dissertation presents the application of the refactoring detection technique on several existing tools for software developers. I show that refactoring detection techniques can improve the usability of refactoring tools, enhance IDE error messages, and help code reviewers understand code changes.
Improving Tool Support for Software Developers through Refactoring Detection

by Xi Ge

is licensed under a Creative Commons Attribution 4.0 International License

Jan 2015
Improving Tool Support for Software Developers through Refactoring Detection

by

Xi Ge

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

Mladen Vouk

Laurie Williams

William Enck

Emerson Murphy-Hill
Chair of Advisory Committee
DEDICATION

To my wife and parents.
XI GE

Xi Ge is a member of the Developer Liberation Front (DLF) research group in the department of Computer Science, North Carolina State University. Before joining NCSU, he completed his Bachelor’s degree in Software Engineering, Nankai University, China in the year of 2009. Xi’s research passion lies in software refactoring, programming languages and human factors in software development. During his PhD study, Xi contributed to several open source projects, including BeneFactor, GhostFactor, ReviewFactor and Sando, with a common theme of improving the productivity of software developers. Research associated with these tools has been published in the International Conference on Software Engineering (ICSE), the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), and other conferences. During his PhD study, Xi interned at the corporate research center, ABB Inc, in the summers of 2013 and 2014; he also interned at Google Inc., in the summer of 2012. The paper associated with Sando won the Best Long Paper Award at VL/HCC14.
ACKNOWLEDGEMENTS

Many people helped me through the journey of getting my research done.

Firstly, I would like to thank my adviser, Dr. Emerson Murphy-Hill, for his guidance and patience. He introduced me to the world of software refactoring, shared with me brilliant ideas, and helped me through the most difficult time during my studies. Also, I would like to thank my committee members, Dr. Mladen Vouk, Dr. Laurie Williams, and Dr. William Enck, for their kindly help and insightful comments on my research. I also would like to thank Dr. Tao Xie for his guidance during the initial stage of my PhD studies.

I also feel grateful to all my external collaborators. Dr. David Shepherd from ABB research provided me a great opportunity to push a fancy research idea into an actual product serving developers in the wild. For their comments and help, I thank my ABB colleagues Mithun Acharya, Vinay Augustine, Andy Cordes, Kosta Damevski, Patrick Francis, Nick Kraft, Xiao Qu, and Will Snipes. During my internship at Google, Dr. Atul Adya and Dr. Gregory Cooper shared with me the state-of-the-practice programming techniques which benefit the implementation of GhostFactor and ReivewFactor.

I am thankful to my colleagues at the Developer Liberation Front (DLF), NC State University: Quinton DuBose helped conduct the formative study associated with the BeneFactor tool; Jim Witschey helped edit several paper submissions and the design of the code review study; Saurabh Sarkar helped the implementation of the ReviewFactor tool. For their insightful feedback and constant help, I also thank other DLF members including Titus Barik, Michael Bazik, Samuel Christie, Daniel Franks, Brittany Johnson, Kevin Lubick, John Majikes, Justin Smith, Feifei Wang, and Shundan Xiao.

Also, I would like to thank all my dear friends and colleagues at North Carolina State University. Special thanks to Anthony Elliott, Denae Ford, Jeehyun Hwang, KyungWha Hong, Ye Jin, Donghoon Kim, Da Young Lee, Madhuri Marri, Fei Meng, Patrick Morrison, Vivek Nair, Rahul Pandita, Kiran Shakya, John Slankas, Will Snipes, Gustavo Soares, Kunal Taneja, Chris Theisen, Suresh Thummalapenta, Ruowen Wang, Xusheng Xiao, Zhe Zhang, and many others. In addition, I would like to thank all participants in my user studies.

I thank my parents for having faith in me and for having patience for all these years. Last but not the least I would like to thank my wife, Guangchao Yuan, for believing in me and for providing me every kind of support I needed for writing this dissertation.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................. viii

LIST OF FIGURES ............................................................... ix

Chapter 1  Introduction ....................................................... 1
  1.1 Motivation ................................................................. 1
  1.2 Problems ................................................................. 5

Chapter 2  BeneFactor ........................................................ 8
  2.1 Introduction ............................................................. 8
  2.2 Motivation ............................................................... 10
  2.3 Formative Study ......................................................... 11
    2.3.1 Participants and Refactorings ................................... 12
    2.3.2 Data Collected .................................................... 13
      2.3.2.1 Pre-study Questionnaires ................................... 14
      2.3.2.2 Observing Manual Refactoring ............................... 14
      2.3.2.3 Post-study Questionnaire .................................. 15
    2.3.3 Results .................................................................. 16
      2.3.3.1 RQ1. Refactoring Correctness .............................. 16
      2.3.3.2 RQ2. Late Awareness ........................................ 18
      2.3.3.3 RQ3. Refactoring Workflow Patterns .................... 19
    2.3.4 Threats to Validity ................................................ 27
  2.4 BeneFactor ................................................................. 28
    2.4.1 Refactoring detection ............................................. 28
    2.4.2 Code modification ................................................ 30
      2.4.2.1 Configuration Information Collection .................... 30
      2.4.2.2 Code Recovery ............................................... 31
      2.4.2.3 Change Creation ............................................. 33
    2.4.3 Example .............................................................. 33
    2.4.4 Technique Challenges ............................................ 34
    2.4.5 Future Work ......................................................... 37

Chapter 3  GhostFactor ........................................................ 38
  3.1 Introduction ............................................................. 38
  3.2 Motivation ............................................................... 39
  3.3 Approach ................................................................. 42
  3.4 Implementation ......................................................... 43
    3.4.1 Supported refactoring types .................................. 43
    3.4.2 Refactoring Detectors ............................................ 45
5.2 Refactoring Opportunities ........................................... 117
5.3 Source Differencing Techniques .................................. 122
5.4 Refactoring Detection .................................................. 124
5.5 Refactoring Studies ...................................................... 126
5.6 Refactoring Tools ......................................................... 128
5.7 Refactoring Conditions .................................................. 133
5.8 Code Review .............................................................. 136

Chapter 6 Conclusion ......................................................... 140

BIBLIOGRAPHY .............................................................. 144
LIST OF TABLES

Table 2.1 Questionnaire results. ................................. 18
Table 3.1 Pre-study questionnaire results for both treatment (T) and control (C) group. 61
Table 4.1 Numbers of respondents who agree with each statement. ............................... 73
Table 4.2 Respondents’ frequency for activities. ...................................................... 74
Table 4.3 Similarity test examples. ...................................................... 86
Table 4.4 Hidden code change for different refactoring types. ................................. 90
Table 4.5 Data for ReviewFactor’s refactoring detection algorithms. ...................... 98
LIST OF FIGURES

Figure 2.1 Rename local variable refactoring example. . . . . . . . . . . . . . . . . . . . . 11
Figure 2.2 Extract method refactoring example. . . . . . . . . . . . . . . . . . . . . . . 12
Figure 2.3 Code illustrating the complex refactoring 2. . . . . . . . . . . . . . . . . . . . . 15
Figure 2.4 Code illustrating the complex refactoring 3. . . . . . . . . . . . . . . . . . . . . 17
Figure 2.5 Refactoring patterns for rename field. . . . . . . . . . . . . . . . . . . . . . . 20
Figure 2.6 Refactoring patterns for extract method. . . . . . . . . . . . . . . . . . . . . . 20
Figure 2.7 Workflow patterns for extract constant. . . . . . . . . . . . . . . . . . . . . . 22
Figure 2.8 Workflow patterns for extract local variable. . . . . . . . . . . . . . . . . . . . 24
Figure 2.9 Workflow patterns for inline local variable. . . . . . . . . . . . . . . . . . . . . 25
Figure 2.10 Workflow patterns for change method signature. . . . . . . . . . . . . . . . . . 26
Figure 2.11 Workflow patterns for pull up field. . . . . . . . . . . . . . . . . . . . . . . . 27
Figure 2.12 Code before extracting method. . . . . . . . . . . . . . . . . . . . . . . . . . . 34
Figure 2.13 Code after cutting the statements to extract. . . . . . . . . . . . . . . . . . . . 34
Figure 2.14 Code after making a non-refactoring change. . . . . . . . . . . . . . . . . . . 34
Figure 2.15 Code after adding a new method declaration. . . . . . . . . . . . . . . . . . . 35
Figure 2.16 Code after invoking BeneFactor. . . . . . . . . . . . . . . . . . . . . . . . . . 35

Figure 3.1 Refactoring with compiler warnings. . . . . . . . . . . . . . . . . . . . . . . . 40
Figure 3.2 GhostFactor components. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figure 3.3 Snapshots compared in a snapshot list. . . . . . . . . . . . . . . . . . . . . . . 41
Figure 3.4 Examples for the refactoring types supported by GhostFactor. . . . . . . . . . 44
Figure 3.5 Code before extract method refactoring. . . . . . . . . . . . . . . . . . . . . . 49
Figure 3.6 Missing return value error. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
Figure 3.7 Extract method errors fixed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Figure 3.8 Missing parameter error. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Figure 3.9 Code before inline method refactoring. . . . . . . . . . . . . . . . . . . . . . . 52
Figure 3.10 Inline method errors. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Figure 3.11 Inline method errors fixed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Figure 3.12 Original code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
Figure 3.13 Code after manually extracting method. . . . . . . . . . . . . . . . . . . . . . 57
Figure 3.14 GhostFactor error message. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
Figure 3.15 GhostFactor quick fix. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
Figure 3.16 Code after GhostFactor fixes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
Figure 3.17 Refactoring Correctness. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Figure 3.18 Refactoring Time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

Figure 4.1 Motivating example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
Figure 4.2 ReviewFactor components. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
Figure 4.3 Examples of mapping software entities. . . . . . . . . . . . . . . . . . . . . . . 81
| Figure 4.4 | Mapping tree examples. | 84 |
| Figure 4.5 | Beyond-file refactoring hiding. | 87 |
| Figure 4.6 | In-file refactoring hiding. | 88 |
| Figure 4.7 | Git diff example. | 92 |
| Figure 4.8 | Open-source projects study results. | 93 |
| Figure 4.9 | Html page example. | 97 |
Chapter 1

Introduction

**Thesis Statement:** Refactoring detection techniques can improve developers’ productivity by prompting developers to use refactoring tools, checking the correctness of manual refactorings, and expediting code review by separating the refactoring and non-refactoring changes.

1.1 Motivation

Developers have to change software to accommodate requirement changes as well as maintaining existing features. The cost of changing software increases with the phase at which the change is performed. Considering fixing defects as an example, the cost of fixing a defect at the operation stage can be one hundred times as much as fixing the same defect at the design stage [Boe81; Eic01]. The reason for the increase is the propagation effect of software change; any changes performed in the later phase of the development process require the update to the artifacts delivered previously. To minimize changes in the later phase, software developers have to design software carefully before implementing it. However, no matter how savvy the developer is at predicting possible changes in the future, no designs can ensure the absolute stability of the software after delivery [McC93]. As an illustration of this point, agile
development methodologies, which focus on embracing change, prevails over the conventional sequential development processes such as the waterfall model [SS08].

The disciplined modification of software after delivery is also known as software maintenance. Sustaining a software system through its life cycle, software maintenance can be either functional or nonfunctional. Functional maintenance modifies the behavior of the software system to satisfy functional requirements, including activities such as fixing defects, adding new features, and adapting the system to a modified environment. On the other hand, nonfunctional maintenance improves the health of software systems without altering their external behaviors, aiming at improving nonfunctional attributes of the software under maintenance such as reliability, testability, availability, cohesion, and portability. The nonfunctional maintenance also involves polishing software artifacts other than source code, such as design documentation and specification.

The nonfunctional maintenance is also known as refactoring. According to the book written by Martin Fowler that systematically introduces the refactoring technique to developers, refactoring is “the process of restructuring existing computer code, changing the factoring, without changing its external behavior” [Fow]. According to the first PhD dissertation on this topic written by William Opdyke, refactoring is defined as “reorganization plans that support change at an intermediate level” [Opd92]. Developers refactor their codebase typically in two circumstance: removing code smells or preparing for the emerging functional changes [Fow]. Code smells indicate design weakness of source code, whose continued existence may lead to real problems. For instance, multiple instances of cloned code snippets potentially increase the effort of fixing defects contained in the snippet [Kam02]. To remove code clones, developers can refactor code to facilitate reusability.

Another reason for refactoring is to prepare for the functional changes. No software design can successfully predict every functional requirement change in the future. Thus, when an existing design does not accommodate an emerging new requirement, developers have to refactor the code to a new design. Compared to removing code smells, refactoring to the new design performs more radical changes in the codebase. For instance, Kim and colleagues studied the Microsoft Windows codebase; the surveyed developers reported that one of the refactoring outcomes is the ability to “release a new product at a much faster cadence” [Kim12].

Refactoring is not only a useful activity but also a frequent one. As evidence, more than half of the global software population is engaged in modifying existing software instead of writing new applications [Pig96]. Refactoring, as a method to improve the quality of the exist-
ing software and prepare for the additional features, certainly becomes an integral part of these developers’ daily tasks. As other evidence, Cherubini and colleagues conducted a structured survey among industrial developers; the surveyed developers rated the importance of refactoring as equal or greater than that of understanding code and producing documentation [Che07]. As further evidence, Murphy-Hill and colleagues studied the commits of both ordinary developers and the toolsmiths who developed programming tools; they found that toolsmiths perform more than forty refactorings each week, while over forty percent of the programming sessions performed by other developers contain at least one refactoring [MH09b]. Xing and Stroulia studied the evolution history of Eclipse, concluding that about 70% of the structural changes are due to refactoring [XS06b]. Similarly, Dig and colleagues studied the API evolution of libraries and found that refactoring is attributable to 80% of the client-breaking API changes [DJ05].

By definition, any code changes that preserve the external behavior are refactorings. The scale of refactoring can range from adding an extra layer of abstraction to the software system to removing a line of dead code. To facilitate communication, developers and researchers generalized a set of frequent and similar refactorings as refactoring types. For instance, updating the identifier names of program elements, such as classes or methods, to more informative names is called the rename refactoring. As another example, reducing a long method body by extracting several statements into a new method and invoking the new method is called the extract method refactoring. These refactoring types are the building blocks for larger refactoring tasks. Also due to the clear definition of these types, automatic refactoring support can be configured by the refactoring type a developer intends to implement.

Automated techniques can support refactoring from various perspectives. Before implementing refactorings, tool support is available to assist developers in identifying the refactoring opportunities; these tools are more commonly known as code smell detectors. For most code smell types, a metrics-based tool can help the developer identify them. For instance, Murphy-Hill and Black proposed an ambient view installed in Eclipse to visually notify developers the detected code smells [MHB08c; MHB10]; Kanemitsu and colleagues visualized the dependency graph to assist developers in identifying extract method opportunities [Kan11]. For the smell types that are more difficult to detect, such as code clones, researcher proposed multiple techniques to help developers by comparing the different versions of the codebase in various representations. For instance, Kamiya and colleagues proposed CCFinder to detect cloned snippets based on similar tokens [Kam02]; Jiang and colleagues proposed DECKARD that is
based on tree differencing [Jia07]; to better the toleration of syntactic difference, Gabel and colleagues proposed a semantic-based clone detector [Gab08].

After identifying a refactoring opportunity, tool support is also available to help the developer execute the refactoring. These tools are more commonly known as refactoring tools. Since Roberts and colleagues proposed the first refactoring tool on smalltalk, mainstream IDEs including Eclipse, Visual Studio, XCode, and IntelliJ, have incorporated refactoring tools as an integral part of the programming environments [Rob97; Ecl; Xco; Int; Net; Vs]. In addition to these refactoring tools integrated to IDEs, standalone refactoring tools are both available as proprietary and open source software. For instance, ReSharper includes a commercial refactoring tool for Visual Studio developers [Res]; building as an extension to LLVM, Clang also provides a refactoring tool library for the C++ developers [Llv; Cla]. Although refactorings can be manually performed, refactoring by tools is inherently less error-prone and more efficient than manual refactorings [MH09b].

In addition to implementing refactoring, automated support can extend further to the post-refactoring stages. Although refactorings potentially improve the software quality, some side effects may also be introduced by the code change. For instance, after renaming a program element, other developers who are familiar with the original name may have difficulty in finding the element; the refactoring on the API level of libraries may break the client code [DJ05]. To solve these problems, developers implemented multiple techniques. For instance, the refactoring tools in Eclipse provide a refactoring script generator allowing developers to replay the refactorings elsewhere [Ecl]. The Git version control system detects the renamed source files and tracks their content change with the files before the renaming [Git]. In the research community, Dig and colleagues proposed a configuration management system that merges commit conflicts in a refactoring-aware fashion [Dig07]. Taneja and colleagues proposed the RefacLib tool that detects the client-breaking refactorings in the libraries’ codebase [Tan07].

Although tool support is available before refactoring, during refactoring and after refactoring, these tools remain underused for various reasons. Considering refactoring tools as an example, these tools are the most accessible refactoring support for developers because of their close integration into every mainstream IDEs. However, according to multiple studies, developers prefer manual refactoring to automated refactoring, even though the former is both more error-prone and more tedious than the latter [MH09b; Vak12]. For instance, Murphy-Hill and colleagues studied four data sets, three of which are the Eclipse programmers’ activity logs and one is the version history of the Eclipse and JUnit open source projects, concluding that
89% of the committed refactorings are manually performed [MH09b]. In addition, they also found that developers almost never mention refactorings in the commit messages [MH09b]. In another study conducted by Vakilian and colleagues, they confirmed that refactoring tools are underused, and further find the evidence showing that developers sometimes invoke refactoring tools in an undesirable way, that is, they use refactoring tools to implement behavior-altering code changes [Vak12].

The underuse of the refactoring support inadvertently effects software development. Firstly, the underuse unnecessarily wastes developers’ time. Refactoring is sometimes tedious to finish. For example, manually renaming a class that is referred in over one hundred source files would take the developer nontrivial amount of time to finish; in contrast, refactoring tools can rename the class almost instantly. Secondly, the underuse potentially introduces more defects into the software. Manual refactoring is as error-prone as adding new features or fixing defects in the codebase [WD06a]; in contrast, refactoring tools update codebase with careful static analysis to ensure behavior-preservation. Therefore, refactoring tools are more correct and reliable in implementing refactorings. Thirdly, the underuse leads to less informative activity logs in IDEs. Logging the refactoring activities can serve multiple purposes including: (1) it helps the version control systems and the developers keep track of the behavior-preserving commits [Ger; Dig07], (2) it helps replay the consistent refactorings across the different systems [Ecl], and (3) it enables researchers to investigate the developers’ workflow patterns [Vak11; MH09b]. Without invoking refactoring tools, these informative logs are missing as well.

This dissertation emphasizes the application of the refactoring detection techniques on improving the tool support for software developers. Because of the blurred boundary between refactoring and non-refactoring changes from the developers’ point of view, existing tools should not rely on developers to identify refactorings, rather, they should be able to detect refactorings and properly respond to them.

1.2 Problems

Refactoring is both a frequent and useful practice to improve software quality. However, the tool support for refactoring faces the underuse problem. I argue that the automatic detection of refactoring is key to improving the usability of refactoring tools due to the blurred boundary between refactoring and non-refactoring changes during code development. As an illustration of this point, Murphy-Hill and colleagues studied the commits of software developers and found
out that refactoring and non-refactoring changes are frequently interleaved, meaning a single commit may contain both types of changes [MHB08c]. They further categorized refactoring as the floss refactoring and the root-canal refactoring, where the former refers to refactoring happens when a developer dedicates her effort into improving the code quality while the latter refers to refactoring happens as a side effect of other behavior-altering changes, such as fixing defects and adding new features [MHB08a]. Floss refactoring is both more beneficial and frequent than root-canal refactoring. As another example, Kim and colleagues interviewed a number of software developers, concluding that developers’ definition of refactoring is not entirely identical to that of researchers [Kim12].

Based on this observation, I use refactoring detection algorithms to improve the usability of refactoring tools, IDE error messages and code review tools. The application of refactoring detection algorithms can help developers to use refactoring tools more often, fix refactoring-introduced defects earlier, and comprehend code changes better. More specifically, the problems I try to solve are:

• Refactoring tools are underused because of the late awareness problem [GMH11; Ge12]. To investigate why developers avoid refactoring tools, I conducted a formative study with twelve developers. I found that developers sometimes unintentionally avoid refactoring tools, meaning that they want to invoke tools but miss the chance because they start manual refactoring without knowing they are refactoring. After realizing they are refactoring, the developers have to choose either to manually finish the refactoring or roll back the partial manual refactoring and reinvoking tools. Both choices are labor-intensive and time-consuming. I refer to this problem as the late awareness problem. To solve the obstacles brought about by the late awareness problem, I implemented a novel refactoring tool called BeneFactor. BeneFactor detects the start of a manual refactoring, reminds developers to use tools to refactor, and finishes the started refactoring automatically. Chapter 2 details the formative study and the design of the BeneFactor tool.

• Another reason for developers’ avoiding refactoring tools is their lack of trust in the automatic code changes performed by refactoring tools [GMH14]. Refactoring tools sometimes apply large-scale code changes, such as renaming an interface that is referred to across the entire project. Although conventional refactoring tools provide a preview to communicate with the developers about the changes they are about to make, few developers pay attention to the preview [MH09b]. Also, recent studies found that refactoring
tools are sometimes buggy, implying that their automatic code changes potentially introduce defects to the developer’s codebase [Dan07; Gli13; Soa13]. To allow developers benefit from refactoring tools without applying automatic code changes, I enhanced the error messages in the Visual Studio IDE with the capability of identifying refactoring-introduced defects. I developed a tool called GhostFactor that detects the manually finished refactorings and checks the refactoring conditions retrospectively; after finding any violations, GhostFactor issues warning messages to help the developer correct her manual refactoring. Chapter 3 details the design of the GhostFactor tool and the human study evaluating the effectiveness of GhostFactor.

- The mixture of the refactoring and non-refactoring changes makes code review challenging. To improve the code quality and communication among developers, many development teams adopt code review as an integral part of the process. By allowing colleagues read and critique each others’ commits, code review can identify defects, transfer knowledge, and incubate alternative designs [BB13]. During code review, a developer may have the varied concerns towards the different types of code changes, such as refactoring and non-refactoring. However, according to existing studies, these types of changes are frequently interleaved with each other in a single commit, placing developers an extra layer of complexity to identify the change types [MH09b]. To help code reviewers more effectively and efficiently evaluate the functional impact of a given commit, I developed a refactoring-aware code review tool called ReviewFactor that allows code reviewers to review the refactoring and non-refactoring part in the change separately. In addition to the tool implementation, I also conducted a survey study about how code review tools can better support refactoring activities. Chapter 4 details the formative study of code reviewers’ refactoring concerns, the design of the ReviewFactor tool, and the evaluation of the current implementation of ReviewFactor.
Chapter 2

BeneFactor

Although useful and widely available, refactoring tools are underused. One cause of this underuse is that a developer sometimes fails to recognize that she is going to refactor before she begins manually refactoring. To address this issue, I conducted a formative study of developers manual refactoring process, suggesting that developers reliance on “chasing error message” when manually refactoring is an error-prone manual refactoring strategy. Additionally, my study distilled a set of manual refactoring workflow patterns. Using these patterns, I designed a novel refactoring tool called BeneFactor. BeneFactor detects a developer’s manual refactoring, reminds her that automatic refactoring is available, and can complete her refactoring automatically. By alleviating the burden of recognizing manual refactoring, BeneFactor is designed to help solve the refactoring tool underuse problem. In this chapter, I described both the formative study collecting developers’ refactoring tool usage patterns as well as the implementation of the BeneFactor tool.

2.1 Introduction

Refactoring can be finished either manually or automatically. Refactoring by hand has been shown to be error-prone [MH09b]. In order to help developers transform programs more efficiently and more correctly, various refactoring tools have been implemented in both academia and industry [Vs; Ecl; Rob97]. These tools promise to help developers refactor faster and with a trivial likelihood of introducing defects. Due to the importance of refactoring tools, all mainstream IDEs integrate them as a part of the environment, making them available in various programming languages to a large population of developers [Vs; Ecl; Xco; Net]. Despite the
wide availability, developers still prefer to refactoring manually. According to Murphy-Hill and colleagues’ study, 89% of all committed refactorings cannot correlate with any refactoring tools’ logs, implying that they are performed manually [MH09b]. More surprisingly, those developers who implemented refactoring tools use these tools no more frequent than ordinary developers do [MH09b].

Besides correlating committed changes with refactoring tools’ logs, other evidence also suggest the underuse problem of refactoring tools. For instance, in a survey of 16 students, only 2 reported having used refactoring tools, and even then only 20% and 60% of the time [MH09b]. In another survey of 112 agile enthusiasts, the developers reported refactoring with a tool a median of 68% of the time [MH09b]. These survey studies subjectively suggest the poor usability of refactoring tools. Recently, Vakilian and colleagues improved refactoring tools’ data collectors and analyzed the richer dataset of developers’ refactoring tool usage [Vak12]. They confirmed Murphy-Hill and colleagues’ finding and summarized multiple causes for the underuse problem.

To improve the usability of refactoring tools, researchers proposed multiple novel tools. For instance, Murphy-Hill and colleagues proposed three types of visual clues to help developers extract method correctly [MHB08b]; However, in spite of their better usability than conventional refactoring tools, these tools do not fundamentally alter the way of invoking refactoring tools, which requires a developer’s recognition of refactoring needs and configuring tools explicitly. However, as I will show in my formative study, many developers cannot recognize they are about to refactor before they manually start doing so. For instance, in the response to the question “why not use refactoring tools” in Murphy-Hill and colleagues’ study, a participant said:

I already know exactly how I want the code to look like. Because of that, my hands start doing copy- paste and the simple editing without my active control. After a few seconds, I realize that this would have been easier to do with a refactoring [tool]. But since I already started performing it manually, I just finish it and con-tinue.

This situation illustrates how refactoring tools do not support the developer when she does not realize she is refactoring until after she has already begun. Without that realization, a software developer will not use any the refactoring tool, no matter how usable, useful, or reliable that tool is. In this chapter, I investigate how to design a refactoring tool that is aware of a
developer’s refactoring, rather than relying on the developer’s recognition of it. I first describe a formative study about developers’ manual refactoring. Building on the study’s results, I designed a novel refactoring tool. I make the following major contributions in this chapter:

- A formative study of developers’ manual refactoring. To the best of my knowledge, I am the first to study developers’ manual refactoring process. My study suggests that reliance on compiler errors when manually refactoring is a common and error-prone technique.

- A proof-of-concept refactoring tool called BeneFactor. In addition to relieving the developer from the burden of recognizing that she is going to refactor, BeneFactor also allows implicit refactoring configuration and the interleaving of refactoring and non-refactoring changes.

### 2.2 Motivation

To use a refactoring tool, a developer must recognize that she is refactoring and select the appropriate refactoring with a menu or a hotkey. If she is unaware that she is refactoring and begins to refactor manually, she may become aware that she is refactoring part way through the manual refactoring. In this case, the developer faces the *late awareness* dilemma: She must either undo all the code changes that she has already made and then invoke the refactoring tool, or keep refactoring manually until the refactoring is complete.

I next illustrate this dilemma using two examples. Suppose Grace is a developer who works on the Apache Tomcat open source project [Apa]. To improve understandability, she wants to change the name of a local variable `descs` to `descriptors`. She starts doing this task manually from the declaration of `descs` towards the tenth and last reference to `descs`. After changing six names (as shown in Figure 2.1) she realizes that she is performing a rename refactoring. Grace decides it would be easier to just finish the refactoring manually, even at the risk of introducing a bug.

Late awareness of refactoring occurs for other refactoring types as well. Suppose that Glen is a developer who works on the Vuze project [Vuz]. He notices that the `getRequestedPieceNumbers()` method shown in Figure 2.2 contains three for loops that are nearly identical. To enhance maintainability, he intends to extract the loop as a new helper method. Glen starts by cutting the code in the third loop, and then declares the helper method `getRequestedPieceNumbersHelper()`. Glen then realizes that he is performing the extract method refactoring, and that he would like to have a refactoring tool figure out
Figure 2.1 Rename local variable refactoring example.

what variables to pass in to the method. To do so, he undoes his changes and invokes the extract method tool, even though he has to configure the refactoring tool with the name of the helper method, which he already specified once while manually refactoring.

2.3 Formative Study

In order to resolve the late awareness dilemma, I intended to build a novel refactoring tool that can complete manual refactorings. However, three important research questions must be answered first:

• RQ1. *How correctly do developers refactor manually?* If developers mistakenly modify program behavior when refactoring manually, then refactoring tools can potentially improve the refactoring process.

• RQ2. *How significant is the late awareness problem?* If many developers do not recognize that they are refactoring before they begin, then the late awareness is a contributor to refactoring tool underuse.

• RQ3. *What are developers’ manual refactoring workflows?* To create a refactoring tool that can complete refactorings automatically, the tool must be able to recognize when
a developer is refactoring. Models of manual refactoring workflows will help my tool recognize when a developer is refactoring.

I conducted a formative study to answer these questions\textsuperscript{1}.

### 2.3.1 Participants and Refactorings

I recruited 12 developers to participate in my study. Six were graduate students and five were commercial software developers. I also recruited one refactoring researcher (not one of the

\textsuperscript{1}Collaborative Work with Quinton L. DuBose
authors) who has current development experience of more than ten years. Although I did not collect all of the participants’ demographic information, at least 10 of the 12 participants had professional programming experience of more than four years.

I asked participants to perform refactorings in the source code of the Vuze [Vuz] project. I chose this project is for its large size and maturity.

I manually inspected the source code of Vuze, then located fourteen locations where refactoring could reasonably be performed. At each of the fourteen locations, I asked participants to perform one refactoring without the assistance of any refactoring tools. Among the fourteen refactorings, I chose three that were especially complex: A rename local variable refactoring, an extract method refactoring and a change method signature refactoring. When performing these complex refactorings, participants needed to carefully consider how to avoid changing program behavior. I describe the complex refactorings in detail in Section 2.3.3.1.

The fourteen refactorings spanned 8 refactoring types: Rename (2), extract constant (1), extract local variable (2), inline local variable (2), change method signature (2), extract method (2), introduce parameter (2), and pull up method (1). I chose these types from the types listed in Fowler’s catalog [Fow] according to three criteria. The first is their frequency in real world. According to Murphy-Hill and colleagues’ study, using automatic tools to perform rename, extract local variable, inline, extract method, and change method signature accounts for over 85% of all refactoring tool usage [MH09b]. This high tool usage suggests that these refactoring types occur frequently. Simplicity is the second criterion because simple refactorings would be easier for participants to complete in a short study session. The third criterion is wide coverage of a variety of software entities, such as constants, temporary variables, local variables, fields, and methods. We also chose pull up method because it involved class inheritance.

2.3.2 Data Collected

I collected data using a pre-study questionnaire, videos of participants’ manual refactorings, and a post-study questionnaire. To view the participants’ refactoring videos and their answers to the questionnaires, the reader can refer to my study material website.²

²https://sites.google.com/site/refactoringstudy/
2.3.2.1 Pre-study Questionnaires

I recruited developers through email. If they were willing, I sent them a consent form and pre-study questionnaire. One subject consented, but did not fill out the remainder of the questionnaire. I asked the following questions in the questionnaire:

- **Coding experience:** About what percentage of your job involves writing code?
- **Java proficiency:** How proficient are you with the Java programming language, rating from 1 to 5 (1 for not at all and 5 for expert)? Participants produced a slightly skewed distribution of Java experience (median=3.5), with all participants reporting at least some experience and two participants considering themselves experts.
- **Refactoring familiarity:** How familiar are you with the practice of refactoring, rating from 1 to 5 (1 for “not at all” and 5 for “I refactor every time I program”)? Participants produced a fairly normal distribution of refactoring experience (median=3).

To perform the study, participants connected to a remote computer running the Eclipse IDE and containing the code to be refactored. Using the online screen sharing service join.me[109], participants could easily view and operate Eclipse. At the same time, I established a Skype[110] conversation with the participant to give them directions.

2.3.2.2 Observing Manual Refactoring

After setting up screen sharing, I opened the first code location where I wanted the participant to perform a refactoring. I told the participant which refactoring I wanted her to perform and that no automatic refactoring tools were allowed. If the participant was unfamiliar with the refactoring, I gave her an explanation of what the resulting code should look like. I were careful to avoid telling the participant how to perform each refactoring. Once I had answered all of the participants’ questions about the task, I asked her to start.

I repeated this process with all 14 refactorings. While the participant was performing these refactorings, I used screen capture software to record the entire process, which I analyzed after the study session ended.

To analyze each refactoring, I tagged it as either “correct,” “incorrect,” or “unknown.” “Correct” meant that the participant’s refactoring resulted in the code structure that met with Fowler’s definition[111] and did not modify the software’s behavior. “Incorrect” indicated that the software’s behavior was modified, but the participant’s refactoring resulted in a code
structure that met with Fowler’s definition. Refactorings tagged as “unknown” included those that the participant skipped; those that were finished only with my detailed guidance (I guided some participants to spare them the embarrassment of not being able to complete the task); those that were finished by invoking refactoring tools; and those that I were not able to tell whether the new code structure met with Fowler’s definition. Only refactorings tagged as “correct” and “incorrect” were used in my analysis of refactoring workflows.

2.3.2.3 Post-study Questionnaire

After the study, I presented each participant with a questionnaire that asked the following questions:

- Q1. How often does this situation occur: I get part way through a code change when I realize there is a refactoring tool that can help me do the job. Choose from one of the following words: Never, rarely, sometimes, often, and always.

- Q2. What would you do after the situation in Q1 happens? Options: (1) I finish the change without a refactoring tool; (2) Back out of the change and redo the change using a refactoring tool; or (3) Other.
2.3.3 Results

2.3.3.1 RQ1. Refactoring Correctness

Murphy-Hill and colleagues’ previous research suggests that developers may rely on compilation errors to locate the related code to update [MH09b]. For example, in Figure 2.1, Grace is using the compilation errors to determine what parts of the code needs to be updated. However, this strategy is sometimes error prone because compilation errors do not indicate every location that needs to be updated for certain complex refactorings. I had participants perform three of these complex refactorings to determine to what extent they used this strategy. Overall, participants inadvertently changed behavior in eleven of the fourteen refactorings, and very few finished participants completed complex refactorings them correctly.

2.3.3.1.1 Complex Refactoring 1

I asked participants to perform the complex extract method refactoring shown in Figure 2.2. If a developer extracts the last \texttt{for} loop into a new method, the new method should return the value of \texttt{pos} because the extracted \texttt{for} loop modifies the value of \texttt{pos} and the code in the original method later reads the value. However, compilation errors do not result if \texttt{pos} is not returned, so relying on error messages for this part of the refactoring is not sufficient for a correct refactoring.

Only one participant correctly performed the complex extract method refactoring, while seven performed it incorrectly because of failing to return the value.

2.3.3.1.2 Complex Refactoring 2

I asked participants to perform a rename local variable refactoring on code that is summarized in Figure 2.3. I asked the participants to rename \texttt{life_hours} to \texttt{original_republish_interval}, which is also the name of a field of the containing class. If a developer renames \texttt{life_hours} to \texttt{original_republish_interval} in the declaration, names in \texttt{checkCacheExpiration(...) that originally bound to the field \texttt{original_republish_interval now bind to the local variable}. No compilation error is generated when the rebinding occurs.

No participant correctly performed complex rename local variable refactoring, while nine performed it incorrectly.
2.3.3.1.3 Complex Refactoring 3

I asked participants to perform a change a method signature refactoring by swapping the order of two parameters. More specifically, I asked them to refactor `setUserData(Object key, Object value)` to `setUserData(Object value, Object key)`, as illustrated in Figure 2.4. If a developer swaps these parameters in the method declaration, no compiler errors are shown because all invocations of this method still had arguments of correct types.

One participant performed this complex refactoring correctly, while two performed it incorrectly.

2.3.3.1.4 Summary

Across the three complex refactorings, 90% were performed incorrectly. It appeared that developers relied on compiler errors for refactoring, even in situations where that reliance was misplaced. Somewhat surprisingly, participants also made mistakes in the non-complex refactorings. In total, participants completed 96 non-complex refactorings, and 21 (22%) of them were incorrect. Most of these incorrect refactorings resulted from participants’ failing to address all compiler errors.

In addition to these results, I also observed another problematic manual refactoring technique. When performing the rename field refactoring, four of the eleven participants invoked

```java
public interface DownloadManager{
    //...
    public void setUserData(Object key, Object value);
    //...
}

protected void announceAll(boolean force){
    //...
    DownloadManager dm = (DownloadManager)
        downloads.get(i);
    dm.setUserData(DM_ANNOUNCE_KEY, new Long( now ));
    //...
}
```

**Figure 2.4** Code illustrating the complex refactoring 3.
Table 2.1 Questionnaire results.

<table>
<thead>
<tr>
<th>Coding experience</th>
<th>Java proficiency</th>
<th>Refactoring familiarity</th>
<th>Late awareness</th>
<th>Handle late awareness</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>4</td>
<td>4</td>
<td>sometimes</td>
<td>100% 0% 0%</td>
</tr>
<tr>
<td>90%</td>
<td>4</td>
<td>3</td>
<td>sometimes</td>
<td>100% 0% 0%</td>
</tr>
<tr>
<td>40%</td>
<td>3</td>
<td>2</td>
<td>often</td>
<td>80% 20% 0%</td>
</tr>
<tr>
<td>75%</td>
<td>5</td>
<td>3</td>
<td>sometimes</td>
<td>65% 35% 0%</td>
</tr>
<tr>
<td>100%</td>
<td>4</td>
<td>4</td>
<td>rarely</td>
<td>80% 20% 0%</td>
</tr>
<tr>
<td>70%</td>
<td>5</td>
<td>5</td>
<td>often</td>
<td>40% 30% 30%</td>
</tr>
</tbody>
</table>

the “find and replace all” tool, replacing all of the occurrences of the original field name in the file. Although this technique did not happen to modify behavior in the given code, this technique is incorrect because the original name may also occur in other places not referring to the field, such as in a method name.

2.3.3.2 RQ2. Late Awareness

How significant is the late awareness problem? I answer this question by using participants’ post-study questionnaires.

Among the 12 participants recruited in my formative study, six returned their post-study questionnaires. Table 2.1 shows the answers of these participants to each question I asked. Three of the six participants indicated that late awareness of refactoring tools happens to them at least “sometimes.” Two of these five participants indicated that late awareness of refactoring happens to them “often”. This suggests that the late awareness problem may happen to a variety of programmers.

Participants reported that when late awareness occurs, a median of 80% of the time they finish the refactoring manually. One participant indicated that she handles late awareness through other means, but did not elaborate.

---

3The first row shows the refactoring researcher’s answers.
2.3.3.3 RQ3. Refactoring Workflow Patterns

What are the developers’ manual refactoring workflows? To investigate this question, I studied the videos of participants performing the refactorings. We distilled a set of widely adopted refactoring workflows for each refactoring type, which I refer to as refactoring workflow patterns. I modeled these patterns using finite-state machines (FSMs) over a set of parameterized elementary operations to facilitate representation and interpretation.

I use operations on abstract syntax tree (AST) nodes to model refactoring workflow patterns. ASTs are tree representations of the syntactic structure of source code written in a programming language. Nodes inside an AST represent software entities at various levels, such as variables, fields, statements, methods, and classes. I use the following operations on the AST nodes to model my refactoring workflow patterns:

- $COP(x)$; copy node $x$’s source code to the clipboard.
- $CUT(x)$; cut node $x$’s source code and keep it in the clipboard.
- $INS(x)$; insert node $x$ into the AST, possibly via a Paste command.
- $UPD(x)$; update the value of node $x$.

Inspired by regular expressions, I also define the following quantifiers:

- $OP(x^*)$ indicates performing operation $OP$ on zero or more nodes $x$ simultaneously.
- $OP(x)^*$ indicates performing $OP$ on zero or more nodes $x$ sequentially.
- $OP(x^+)$ indicates performing operation $OP$ on one or more nodes $x$ simultaneously.
- $OP(x)^+$ indicates performing $OP$ on one or more nodes $x$ sequentially.

2.3.3.3.1 Rename Field

Participants had similar workflows when manually performing the rename field refactoring. Eleven rename field refactorings that were considered usable (that is, correct and incorrect). Each participant used one of two patterns:
Figure 2.5 Refactoring patterns for rename field.

Figure 2.6 Refactoring patterns for extract method.

• Seven participants first updated the name of the field in its declaration, and then iteratively updated the names of all the references to this field. I modeled this workflow by the upper transitions (red) in Figure 2.5. In the Figure, the number on a transition indicates the number of participants following that transition, while the square brackets indicate a conditional transition.

• Four participants invoked the “find and replace” tool to automatically replace all of the occurrences of the field’s name. I modeled this workflow by the lower transition (blue) in the Figure 2.5.
2.3.3.3.2 Extract Method

I collected ten usable extract method refactorings. Participants adopted several workflows when performing extract method. Three workflows were common to multiple participants:

- Two participants first copied the statements to extract, made a new method declaration, pasted the statements into the new method body, added parameters to the new method declaration, and finally replaced the statements to extract with the new method invocation. I modeled this workflow by the upper transitions (red) in Figure 2.6.

- Two participants first cut the statements to extract, made a new method declaration, pasted the statements into the new method body, added parameters to the new method declaration, and finally inserted the new method invocation to the place where the extracted statements were. I modeled this workflow by the middle transitions (blue) in Figure 2.6.

- Two participants first added a new method invocation near the statements to extract, cut the statements to extract, made the new method’s declaration, pasted the statement into the new method body, and finally added parameters to the new method declaration. I modeled this workflow by the lower transitions (green) in Figure 2.6.

I also observed the other four patterns (not shown in Figure 2.6) adopted by one participant each:

- One participant first added a new method declaration after the method of the statements to extract, copied the statements to extract, pasted these statements into the new method body, replaced these statements with the new method’s invocation, and finally added parameters to the new method declaration.

- One participant first cut the statements to extract, added a new method invocation at these statements’ original place, added a method declaration after the statements’ original method, pasted the statements to extract into the new method body, and finally add parameters to the new method declaration.

- One participant first cut the statements to extract, pasted these statements after their original method, surrounded these statements by a new method name and brackets, added parameters to the new method declaration, and finally inserted the new method invocation at the extracted statements’ original place.
Figure 2.7 Workflow patterns for extract constant.

- One participant first copied the statements to extract, added a method declaration below these statements (inside their containing method), pasted the statements into the new method declaration, cut the new method declaration, pasted the new method declaration after the method containing the statements to extract, added parameters to the new method declaration, and finally replaced the statements to extract with the invocation of the new method.

These patterns suggest that developers manually refactor using only a few different basic actions, but interleave those actions in a variety of ways.

2.3.3.3 Extract Constant

In total, I collected eleven extract constant refactorings. These refactorings fell into seven different refactoring workflow patterns. These following three patterns were adopted by more than one participant.

- Three refactorings (8, 10 and 14) first copied the constant string to extract, created a new local variable declaration, pasted the string to the declaration (assigned it to the local variable) and finally replaced the constant string to extract with a reference to the local variable. This pattern is illustrated by the red transitions in the finite state machine in Figure 2.7.

- Two refactorings (4 and 15) first created a new local variable declaration, copied the constant string to extract, pasted the string to the declaration (assigned it to the local variable) and finally replaced the constant string to extract with a reference to the local variable. This pattern is illustrated by the blue transitions in the finite state machine in Figure 2.7.
• Two refactorings (7 and 13) first cut the constant string to extract, created a new local variable declaration, pasted the string to the declaration (assigned it to the local variable) and finally added a reference to the local variable at the location where the constant string originally was. This pattern is illustrated by the green transitions in the finite state machine in Figure 2.7.

In addition to above patterns, I collected the following four workflow patterns that were adopted by less participant:

• One refactoring (5) first copied the constant string to extract, replaced the string with a reference, created a local variable declaration of the reference, and finally pasted the string to the declaration (assigned to the variable).

• One refactoring (6) first created a local variable declaration, cut the constant string to extract, pasted the string to the declaration (assigned to the variable), and finally added a reference to the variable at the location where the string to extract originally was.

• One refactoring (11) first cut the constant string to extract, added a reference at the location where the string originally was, pasted the string nearby the reference, and finally added a local variable declaration before the pasted string (assigned the string to the variable).

• One refactoring (12) first cut the constant string to extract, pasted the string nearby, created a local variable declaration before the pasted string (assigned the string to the variable), and finally added a reference to the local variable at the location where the extracted string originally was.

### 2.3.3.3.4 Extract Local Variable

In total, I collected fourteen extract local variable refactoring. These refactorings fell into six different refactoring workflow patterns. These following two patterns were adopted by more than one participant.

• Two refactorings (10 and 15) first created a new local variable declaration, copied the expression to extract, pasted the expression to the declaration, and finally replaced the expression to extract with a reference to the local variable. This pattern is illustrated by the red transitions in the finite state machine shown in Figure 2.8.
Figure 2.8 Workflow patterns for extract local variable.

- Three refactorings (2, 14, and 11) first cut the expression to extract, pasted the expression nearby, created a new local variable declaration before the pasted expression, and finally added reference at the location where the expression originally was. This pattern is illustrated by the blue transitions in the finite state machine in Figure 2.8.

- Three refactorings (7, 12 and 13) first cut the expression to extract, added reference at the location where the expression originally was, created a new local variable declaration, and finally pasted the expression in the declaration. This pattern is illustrated by the green transitions in the finite state machine in Figure 2.8.

In addition to above patterns, I collected the following four workflow patterns that were adopted by less participant:

- Two refactorings first created a new local variable declaration, cut the expression to extract, pasted the expression to the local variable declaration, and finally added reference at the location where the extracted expression originally was.

- One refactoring (7) first cut the expression to extract, created a new local variable declaration, pasted the expression to the local variable declaration, add finally added reference to the local variable at the location where the extracted expression originally was.

- One refactoring (11) first created a new local variable declaration, cut the expression to extract, added a reference to the local variable at the location where the cut expression originally was, and finally pasted the expression to the local variable declaration.

- One refactoring (10) first copied the expression to extract, pasted it nearby, added a local variable declaration before the pasted expression, and finally replaced the expression with the local variable reference.
• One refactoring (14) first cut the expression to extract, added a reference at the location where the extracted expression originally was, created a new local variable declaration, and finally pasted the expression to the local variable declaration.

2.3.3.3.5 Inline Local Variable

In total, I collected seventeen extract constant refactorings. These refactorings fell into seven different refactoring workflow patterns. These following pattern was adopted by more than one participant: Five refactorings (4, 6, 6', 10 and 15) first copied the constant string at the declaration of the local variable, replaced the references to the local variable with the string, and finally deleted the declaration of the local variable, illustrated by the finite state machine in Figure 2.9.

In addition to the above pattern, I collected the following five workflow patterns that were adopted by less participant:

- Two refactorings (7 and 7') first cut the declaration of the local variable (left the constant string), cut the constant string in the declaration, and finally replaced the references to the local variable with the constant string.

- Two refactorings (11 and 11') first copied the constant string, deleted the local variable declaration, and finally replaced the references to the local variable with the constant string.

- One refactoring (12) first cut the constant string, replaced the references to the local variable with the string, and finally deleted the declaration of the local variable.

- One refactoring (13) first commented the declaration of the local variable, copied the constant string in the comment, and finally replaced the references to the local variable with the constant string.
• Two refactorings (14 and 14') first cut the constant string in the declaration of the local variable, deleted the declaration of the local variable, and finally replaced the references to the local variable with the constant string.

• One refactoring (4') first commented the declaration of the local variable, copied the constant string in the declaration, and finally replaced the references to the local variable with the constant string.

• Three refactorings (10', 12' and 13') first copied the constant string in the declaration of the local variable, and invoked the find and replace utility to replace all the references to the local variable with the constant string. (Problematic strategy)

2.3.3.3.6 Change Method Signature

I collected twelve change method signature refactorings falling into two workflow patterns. In this refactoring, I asked participants to switch the positions of two parameters. These two parameters did not share the same type. Therefore after the modification of the method declaration, a set of compiler errors can guide participants to locate the method calls for update. These two workflow patterns are:

• Eleven refactorings (4, 5, 6, 6', 7, 10, 11, 12', 14, 14', and 15) first cut one parameter in the signature of the method declaration, pasted it before/after the other parameter (added and deleted comma if needed), and finally updated the method calls to this method by performing the similar position switch. This workflow pattern is modeled by the finite state machine in Figure 2.10.

• One refactoring (8) first update the method calls by exchanging the positions of the two arguments, cut one parameter in the signature of the method declaration, and finally pasted the parameter before/after the other parameter (adjusted comma if needed).
2.3.3.3.7 Pull Up Field

I collected nine pull up field refactorings falling into two workflow patterns. In this refactoring, I asked participants to move a field from a class to its super class. This refactoring was comparatively straightforward since no name clash would happen after the moving. The two workflow patterns are:

- Eight refactorings (5, 6, 7, 8, 11, 12, 13, and 14) first cut the field declaration to pull up, navigated to the declaration of the super class, and finally pasted the field declaration to the super class declaration. This pattern is modeled by the finite state machine in Figure 2.11.

- One refactoring (10) first commented the field declaration to pull up, copied the field declaration from the comment, navigated the declaration of the super class, and finally pasted the field declaration to the super class declaration.

2.3.4 Threats to Validity

Although my formative study provides data on how developers manually refactor, there are several threats to validity that should be considered when interpreting my results.

The first threat is the criteria I used to choose refactoring types. I chose the refactoring types whose tools are used more frequently, yet these types of refactorings were not always the most underused refactoring tools. In the future, I would like to perform the study with refactoring types for which the corresponding tool is the most underused.

Another threat is the number of participants in my study, and that not all participants produced usable data for every refactoring. Thus, I cannot make strong claims about how my results generalize to other developers. For example, some additional refactoring patterns may exist that were not exhibited by my participants. However, I believe that the results provide a sufficient starting point to begin building a proof-of-concept refactoring tool, as described in
Section 2.4. Similarly, the selected refactorings may not be representative of all refactorings performed in the wild.

Another significant threat is that participants’ attitude toward refactoring third-party software may be different than their attitude towards refactoring their own software. Specifically, because participants have no investment in the third-party code, they may be less concerned about introducing behavior-modifying changes, as compared to their own code. This would cause my study to overestimate how often refactoring errors are made. It is also possible that participants exercised more diligence in refactoring correctly in the study, since they were not distracted by other coding tasks. This would cause my study to underestimate how often refactoring errors are made. In either case, a field study may be able to shed more light onto the frequency of errors made during manual refactoring.

2.4 BeneFactor

According to my formative study, manual refactoring can be an error-prone task (RQ1). Although conventional refactoring tools are available to assist the developers performing correct and efficient refactoring, they are significantly underused [MH09b]. This underuse problem partially results from the developers late awareness of refactoring (RQ2). In order to tackle this problem, I propose a novel refactoring tool called BeneFactor that is built on my distilled refactoring workflow patterns (RQ3). Unlike conventional refactoring tools, BeneFactor automatically detects an on-going manual refactoring, reminds the developers that automatic refactoring is available, and can finish the manual refactoring after the developer’s explicit invocation without requiring her to undo any code changes. Implemented as a plugin for the Eclipse IDE, a prototype of BeneFactor is publicly available. BeneFactor has two major components: Refactoring detection and code modification.

2.4.1 Refactoring detection

To solve the late awareness problem, the BeneFactor tool needs to recognize manual refactoring first. I designed to refactoring detection component to achieve this. In contrast to existing refactoring detection tools, such as RefacLib [Tan07] and REF-FINDER [Pre10], my refactoring detection component detects refactoring while the developer is programming, rather than in

\[\text{https://github.com/DeveloperLiberationFront/BeneFactor}\]
the version control system. Moreover, BeneFactor detects ongoing and unfinished refactorings, rather than completed ones. The refactoring detection component runs in the background of the IDE and continuously captures the developer’s operations on the code base. It collects two kinds of operations:

- Code-change based operations are detected by comparing subsequent snapshots of code base. These snapshots are captured after certain developer events, such as adding a statement, deleting a field declaration, and updating a variable name.

- Action-based operations are detected by monitoring the commands a developer executes, such as copying a statement and cutting a method declaration.

The refactoring detection component detects an ongoing refactoring by matching the operations performed by a developer against the transitions in the manual refactoring workflow patterns from my formative study. If a developer’s operations match with a prefix of refactoring type R’s workflow pattern, she may be performing a refactoring of type R. If her operations continue to match the same pattern, my confidence that she is actually refactoring increases; otherwise the confidence decreases. When the confidence exceeds a predefined threshold T, the refactoring detection component concludes that the developer is manually performing a refactoring of type R and offers a quick-fix to help her finish it.

I next illustrate my refactoring detection algorithm through a pseudo code example shown in Code 1. This example outlines detecting the rename field refactoring by using the refactoring workflow pattern illustrated in the upper transitions (red) in Figure 4. The value of confidence indicates the numbers of consecutive matches between a developer’s operations and the states of the workflow pattern. The algorithm detects an ongoing rename refactoring when the value of confidence is above a predefined Threshold (line 22). I currently use a threshold value of 1, resulting in the highest sensitivity of the detection algorithm.

If the refactoring detection component detects an ongoing refactoring, my Eclipse plugin adds a quick-fix button at the line of code where the developer made the latest change. The quick-fix button offers a user interface affordance to allow BeneFactor to finish the refactoring.
confidence = 0;
current_state = init;
while(true){
    operation = WaitforOperation();
    if(operation.isUpdate() && operation.Node.isName() && operation.Node.Parent.isFieldDeclaration()){
        confidence++;
        current_state = two;
        declaration = operation.Node;
    }
    else if (current_state == two && operation.isUpdate() && operation.Node.isName() && !operation.Node.Parent.isFieldDeclaration() && operation.bindsTo(declaration))
        confidence++;
    else if (confidence > 0)
        confidence --;
    if(confidence > Threshold)
        DetectedRefactorings.add(new RenameRefactoring(binding));
}

Code 1 Refactoring detection pseudo code example

2.4.2 Code modification

If the developer invokes BeneFactor to finish her manual refactoring, BeneFactor’s code modification component makes the requested change. At a high level, the code modification component captures any information the developer supplied while manually refactoring, recovers the code to its original state by rolling back the manual refactoring, and then re-applies the refactoring automatically. We discuss each of these steps below.

2.4.2.1 Configuration Information Collection

The configuration information collection step collects the necessary information to perform an automatic refactoring. Some examples of the configuration information include the visibility modifier of the extracted method and the position of the method in the containing class. The equivalent step in conventional refactoring tools is to ask for this information via a dialog box. But rather than asking for configuration information a second time using a dialog box, BeneFactor automatically collects this information from the code changes that the developer has already made.
While it is convenient to collect configuration information this way, it is not always possible to collect all information necessary to complete a refactoring. For example, if a developer invokes BeneFactor right after she cuts the statements to be extracted, the new method’s name is not yet known. In this situation, BeneFactor uses a default value to finish the refactoring first and then allows the developer to modify the default afterwards. However, sometimes even this is not enough. For example, when performing the move static field refactoring, the tool may not know the developer’s planned destination class. In those situations, BeneFactor does not offer to finish a refactoring until this critical information is supplied. An alternative approach would be to collect the missing information using a dialog box or wizard.

2.4.2.2 Code Recovery

The code recovery step recovers the code base to before the manual refactoring was applied. Undoing a manual refactoring is not a trivial task, because the developer’s manual refactoring effort may interleave with other kinds of code changes [MH09b]. For example, when a developer is performing a rename field refactoring, after renaming the field’s declaration, she may add new statements to the code. In this situation, arbitrarily undoing all code changes until the beginning of the manual refactoring causes the developer’s non-refactoring work to be lost. To tackle this problem, I designed an algorithm called selective undo. This algorithm simulates a developer’ rolling back of manual refactoring changes when she tries to invoke refactoring tools after encountering the late awareness problem.

I illustrate selective undo by using the pseudo code snippet shown in Code 2. Our selective undo algorithm takes two parameters as input. The first is the stack of operations performed by the developer from the starting of the Eclipse IDE to the moment of the algorithm’s execution, with the latest code change at the top. The second input is the refactoring to be completed.

In Code 2, the algorithm first declares a local variable undo_operation_stack to store all the operations that have been undone. After the declaration, performedEnoughUndos at line 5 checks whether it needs to undo more code changes. This method returns true if the existing automatic refactoring infrastructure can be applied to finish the refactoring, otherwise it returns false. The implementation of performedEnoughUndos depends on the type of refactoring. For example, for rename refactoring, performedEnoughUndos returns true when the name’s declaration has not been renamed; for extract method refactoring, it returns true when the statements to be extracted appear in their original method.
# Code 2  selective undo algorithm pseudo code example

When `performedEnoughUndos` returns false, the selective undo algorithm enters a loop. From line 6 to line 7, the algorithm pops the code change at the top of `operation_stack` and undoes it, regardless of whether or not the operation is a refactoring operation. For every operation that has been undone, the algorithm pushes it back to another stack `undo_operation_stack` at line 8. The selective undo algorithm continuously executes the code from line 6 to line 8 until `performedEnoughUndos` returns true.

After arbitrarily undoing all of these operations, the algorithm redoes only the non-refactoring operations among them. For each undone operation, the algorithm invokes `refactoring.isRefactoringOperation()` at line 13 to check whether or not it belongs to the manual refactoring workflow. `refactoring.isRefactoringOperation()` performs this check by comparing the input operation with the operations in my collected refactoring workflow patterns.

If `refactoring.isRefactoringOperation()` returns true, the algorithm skips the operation because BeneFactor will re-apply the refactoring later, as shown at line 14. If `refactoring.isRefactoringOperation()` returns false, the algorithm re-applies this operation. Directly applying this operation may be incorrect because the previously skipped operations may change the position where this operation should be applied to. Therefore, I use the patch technique to re-apply this operation. Widely used in source control systems,
patch allows one version to a file to be applied to the right location in the modified version of the same file [Dif]. The algorithm repeats this process from line 11 to 16 until no operation is left in undo_operation_stack.

2.4.2.3 Change Creation

After the code base has been recovered, I perform the last step: automatically finishing the refactoring. Eclipse ships with a set of automatic refactoring application programming interfaces (APIs) called the language toolkit (LTK) [Ltk]. Feeding the collected configuration information and the recovered code base as input, LTK APIs are now able to automatically perform the intended refactoring.

2.4.3 Example

I next illustrate how BeneFactor works by using a real-world code example. Suppose Grace is using Eclipse with BeneFactor installed. To enhance maintainability, she intends to extract the for loop in the code snippet of Figure 2.12 into a new method. Grace first cuts the statement of the for loop. BeneFactor captures this operation and detects that the developer is possibly refactoring, and therefore adds a marker at the line of code where Grace cuts the statement, proposing to help her finish this refactoring, as illustrated in Figure 2.13. Grace next intends to print a blank line before the printed information in the for loop, so she inserts System.out.println(), as illustrated by Figure 2.14.

Grace continues her extract method refactoring by adding a new method declaration. She specifies the visibility modifier, method name, and return type of the new method declaration, as shown in Figure 2.15. Although interrupted by a non-refactoring operation, Grace has performed a sequence of operations following the refactoring workflow pattern modeled by the middle transitions (blue) in Figure 2.6. Therefore, BeneFactor still detects that Grace is performing an extract method refactoring. Reminded by the BeneFactor icon, Grace realizes she can finish her manual refactoring by invoking the tool. Hence, she chooses the quick-fix item in Figure 2.15 to extract the method. Finally, BeneFactor finishes her refactoring, resulting in the code snippet shown in Figure 2.16. The parameter of the extracted method is inferred by the refactoring tool, and thus does not need to be explicitly specified by the developer.
2.4.4 Technique Challenges

While implementing BeneFactor, I faced several challenges. I believe that these challenges must be resolved before BeneFactor can be evaluated.

**False Positives.** False positives in refactoring detection refer to the situations when BeneFactor falsely assumes that the developer is refactoring manually, but she is actually not. Too
many false positives may annoy the developer with an overwhelming number of quick-fix items. The false positives may occur when (1) the developer is performing code changes that serve partially for restructuring purposes, but also are intended to modify the code behavior and (2) the default confidence value is too low, resulting in an overly sensitive refactoring detection algorithm.

**False Negatives.** False negatives in refactoring detection refer to the situations when BeneFactor fails to detect that a developer is manually refactoring. In this situation, the developer is
not able to invoke BeneFactor to finish her ongoing refactoring. The false negatives in refactoring detection may be caused by: (1) A manual refactoring workflow that greatly deviates from my collected manual refactoring workflow patterns; (2) A workflow that is dominated by non-refactoring code changes; (3) A default confidence value that is too high, resulting in an overly insensitive refactoring detection algorithm.

**Unresolvable Non-Refactoring Operations.** As I have mentioned, a developer’s manual refactoring operations may be interleaved with operations serving some other purpose, which I refer to as non-refactoring operations. These non-refactoring operations are preserved during the code recovery step. However, BeneFactor may have difficulty in resolving these non-refactoring operations when they are dependent on the interleaved refactoring.

One example of the unresolvable non-refactoring operation is illustrated by Grace’s insertion of `System.out.println();` in Figure 2.14. BeneFactor must figure out the order of the inserted statement and the invocation of the extracted method. Currently, BeneFactor assumes the invocation happens after all the inserted statements, resulting in the code snippet shown in Figure 2.16. The developer’s intention may have instead been to put the invocation before the inserted statements, but BeneFactor currently has no way of determining the developers’ intention.

**Removing Quick-Fix Items.** BeneFactor adds a quick-fix item after detecting a manual refactoring. However, it is not trivial to decide when to remove the item. If BeneFactor removes the item too early, the developer is not able to invoke automatic refactoring when she later wants to. If BeneFactor keeps the quick-fix item for too long, many quick fixes may be strewn around the developer’s IDE, long after code changes have been completed.

**Scalability.** I plan to enhance BeneFactor to support more types of refactorings, and as a developer uses BeneFactor, the developer’s operations may simultaneously match with workflow patterns of several different refactoring types. In this case, the developer will face multiple quick fix options to choose from.

**Other Causes for Underuse.** In the formative study in Section 2.3, I found several causes for developers’ not using refactoring tools. However, these causes may not be comprehensive, that is, other causes may also exist. For instance, developers prefer manual refactorings potentially for trust concerns. Or, developers would like to manually refactor a codebase to explore and understand the codebase better.
2.4.5 Future Work

Using the refactoring workflow patterns I collected in the formative study, I designed the BeneFactor tool to solve the late awareness problem. The most promising future work is to evaluate BeneFactor with actual developers working on real-world software. To conduct this study, I plan to deploy BeneFactor to a team of more than three professional developers who work on the same open source project. I plan to log the tool usage data of BeneFactor during their using the tool for several weeks. The collected data is supposed to answer the following questions.

Do developers invoke refactoring tools more often? My formative study shows that developers’ not using refactoring tools is sometimes due to the late awareness problem. With BeneFactor installed, if a developer’s manual refactoring falls into the supported workflow patterns, the developer can be reminded of the availability of the automated tool support for refactorings. Therefore, the developer is more likely to use refactoring tools. Using the collected data, I can test this hypothesis.

Do developers refactor more often? If developers use refactoring tools more often, refactorings in general are easier to implement than that when refactoring tools are underused. With the collected data, I plan to test the hypothesis that using BeneFactor allows developers to more often improve the internal quality of software.

How early should a manual refactoring being detected? The current implementation of BeneFactor detects a manual refactoring with the highest sensitivity, that is, when an editing step matches with the first transition of any supported refactoring workflow patterns, BeneFactor reminds the developer of the availability of tool support. However, one caveat of a higher sensitivity is more false positives, which may annoy developers in the longer term. On the other hand, a lower sensitivity potentially postpones manual refactorings being detected, thus lowering the usefulness of automated refactorings. In this study, I plan to investigate the optimum configuration of the refactoring detection.
Chapter 3

GhostFactor

Although BeneFactor has the potential of solving the late awareness problem. Developers may still manually refactor their codebase for other reasons, for instance, trust issues. Existing studies show that the conventional refactoring tools are buggy [Dan07; Soa13]; thus, refactoring by tools may sometimes introduce defects to the codebase. To address this problem, I propose a technique called GhostFactor separating transformation and correctness checking: I allow the developer to transform code manually, but check the correctness of her transformation automatically. I implemented my technique as a Visual Studio plugin, then evaluated it with a human study of eight software developers; GhostFactor improved the correctness of manual refactorings by 67%.

3.1 Introduction

Although automated refactoring tools refactor more correctly than developers do, developers rarely use them. According to existing studies, only 11% of 145 refactorings in real-world open source systems were performed automatically [MH09b; XS06b].

To solve this underuse problem, researchers have proposed novel tools to encourage developers to refactor automatically. For instance, BeneFactor and WitchDoctor automatically finish refactorings after developers start refactoring manually [Fos12]. These tools significantly reduce, but do not completely remove, the barriers to using refactoring tools. For instance, developers must still explicitly invoke most refactoring tools. In addition, researchers found that developers do not trust automatic refactorings to be correct [MH07; Dan07]. Existing tools
remove neither of these barriers: developers are unlikely to change their behavior to use a tool they distrust [MH07].

To address these problems, in this chapter, I propose a novel static analysis technique. I make the following contributions:

- A technique called GhostFactor that can detect manually performed refactorings and check their correctness. GhostFactor is novel for combining light-weight static analysis with refactoring detection algorithms to quickly detect refactoring errors. Section 3.3 describes the design of GhostFactor.

- I implemented my technique in an open-source plug-in for the Visual Studio IDE [Vs]. This plugin, also called GhostFactor, instantly notifies developers when they refactor incorrectly and suggests ways to fix the error. Unlike previous refactoring tools, GhostFactor integrates into the IDE’s notification system, a familiar mode of interaction for developers. Section 3.4 describes the implementation.

- I evaluated GhostFactor by conducting a human study with eight developers. In this study, I compared how participants refactored with or without GhostFactor. GhostFactor improved the correctness of their manual refactorings by 67%. Section 3.5 presents the design and results of the study.

3.2 Motivation

One desirable property of software development tools, such as refactoring tools, is that they should accommodate the developer, rather than the developer having to accommodate the tools. In this section, I give two common patterns developers use when refactoring, show how existing refactoring tools do not accommodate this behavior, and describe how my proposed approach accommodates this behavior.

Developers’ Refactoring Patterns. My work in Chapter 2 found two common patterns in developers’ refactoring behavior. Murphy-Hill and colleagues found developers transformed their code manually nine out of ten times, rather than using a refactoring tool [MH09b]. Part of the reason for this is that developers do not trust the changes refactoring tools make [MH07].

As illustrated in Chapter 2, the second pattern I found in developers’ refactoring behavior was using compiler errors to guide and check the correctness of manual refactorings. For example, suppose a developer is going to change the signature of method m() in Figure 3.1a by
exchanging the positions of its first and second parameters. She might first change the method declaration, as illustrated in Figure 3.1a. The compiler raises errors at all the method’s call sites, because the types of their arguments no longer match the declared type signature. The developer can use these compiler errors to find all the call sites, then correct them to finish the refactoring, as illustrated in Figure 3.1b.

In some cases, such as the example in Figure 3.1, developers can use this pattern to finish refactorings completely and correctly. However, this is not always the case. For example, changing the parameters of void n(int a, int b) to void n(int b, int a) will not raise any compiler errors because the exchanged parameters share the same type. My study in Chapter 2 found developers who rely on such compiler errors make more refactoring errors.

**Existing Tools Do Not Accommodate These Patterns.** Existing refactoring tools do not accommodate these common refactoring patterns: they require developers to manually invoke them, so developers must change their workflow to use them. However, I found some developers do not know what refactoring tools are available. Even when they do, developers sometimes complete part of a refactoring before realizing they are refactoring. They often choose to simply complete the refactoring manually, rather than undoing previous changes and invoking a tool. Thus, as shown in Chapter 2, requiring explicit invocation is a barrier to the use of traditional refactoring tools.

The refactoring tools BeneFactor and WitchDoctor [Fos12] address these awareness problems, but do not accommodate developers’ existing refactoring patterns. Specifically, both tools
require the developer to trust the transformations the tools make. As mentioned previously, many developers do not trust tools to make correct changes [MH07].

**My New Approach Accommodates These Patterns.** GhostFactor fits into more developers’ workflows because it does not assume that developers know about refactoring tools or trust automatic code transformations. My approach separates automatic transformations from automatic correctness checks so developers can benefit from the second without adopting the first. GhostFactor uses familiar notifications, similar to compiler warnings, to inform the developer of errors made while manually refactoring.
3.3 Approach

GhostFactor detects developer-introduced defects to ensure the correctness of manual refactorings. In this section, I describe its design. GhostFactor has several independent components, each of which handles a different task. Figure 3.2 illustrates these components and how they interact with each other.

The first component is the **history saving** component. This component records the change history of different files that a developer has worked on. It registers a listener to content change events issued from the active editor. When an event occurs, the component takes a snapshot of the source file currently opened in the active editor and saves its content in memory. The change history of a specific source file is maintained as a snapshot list to facilitate sequential access, where the head of the list is the most recent snapshot. After saving the source file’s content and updating its snapshot list, the component feeds the list to the next component.

The **refactoring detection** component detects refactorings that the developer completes manually. This component takes the snapshot list for a source file as input, then dynamically loads available refactoring detectors and applies them to the snapshot list. This component uses each detector to compare the latest snapshot in the snapshot list with one of the previous snapshots. For instance, Figure 3.3 depicts a snapshot list where the latest node is Snapshot 4. I show the compared snapshots by drawing arcs between them. In this figure, I also label the distance between compared snapshots, where the distance is one more than the number of snapshots between the two. GhostFactor does not compare snapshots if the distance between them is larger than some constant maximum; the selection of the maximum is based on two criteria: (1) the value should be great enough to contain a developer’s typical steps to finish a manual refactoring, and (2) the value should not be overly great to ensure the time efficiency of the detection. Tentatively, I set the maximum value as 30.

All the refactoring detectors share a common interface that developers can use to implement their own detectors. The interface takes two abstract syntax trees (ASTs) of the same source file as inputs and generates outputs that indicate whether a refactoring is detected, the detected refactoring’s type, and the AST nodes affected by the refactoring. For example, the affected AST nodes of an extract method refactoring are the newly declared method and the invocations of that method.

If GhostFactor detects a manual refactoring, it feeds the refactoring to the **condition checking component**. This component dynamically loads available condition checkers for
the given refactoring’s type. These condition checkers can check for both pre-conditions and post-conditions for the given refactoring [Opd92]. All of the condition checkers for a refactoring type share a common interface with which developers can implement their own condition checkers to add to GhostFactor. The inputs of this interface include the detected refactoring and the ASTs in which the refactoring is detected. The output indicates whether the refactoring violates or passes the condition.

The last component, refactoring warning component, keeps track of detected condition violations. Taking a condition violation as input, the component first checks whether a warning has already been issued for this violation. If one has, this component discards the redundant violation; otherwise, the component saves it and issues a new refactoring warning. Taking a passing condition as input, the component will check whether this passing condition resolves an existing refactoring warning. If it does, the component dismisses this warning. For each refactoring warning, GhostFactor also provides quick fixes with which the violation can be automatically resolved.

To illustrate how the last component works, I next give an example. Suppose a developer is refactoring on a source file $F$, consecutively generating four different snapshots $F_1$, $F_2$, $F_3$, and $F_4$. $F_1$ is the original file; $F_2$ contains an erroneous refactoring $R$ that violates a condition $C$; $F_3$ also contains $R$ violating the condition $C$; $F_4$ contains the corrected $R$ that fixed its violation of $C$. In this example, the refactoring warning component will consecutively receive three different condition checking results, which are $R$’s violation of $C$, $R$’s violation of $C$ again and finally $R$’s pass of $C$. According to the mechanism of the refactoring warning component, GhostFactor will issue a refactoring warning at $F_2$ and $F_3$, while removing the warning at $F_4$.

### 3.4 Implementation

Built on the Microsoft Roslyn project [Ros], I implemented GhostFactor as a plug-in for the Visual Studio IDE to help refactor C# software; the source code of this plug-in is open to the public\(^1\). In this section, I describe this implementation.

#### 3.4.1 Supported refactoring types

The GhostFactor plug-in currently supports the following three refactoring types.

\(^1\)https://github.com/DeveloperLiberationFront/GhostFactor2
• **Extract method**: According to Fowler’s refactoring catalog, extract method is a type of refactoring that “turns a set of statements into a new method whose name clearly explains that purpose of the method” [Fow], as illustrated in Figure 3.4a. Fowler recommends Extract method to eliminate code smells such as long method and duplicated code [Ker04].

• **Change method signature**: Change method signature is a refactoring type that changes the parameter list of a given method without modifying the method’s functionality, as illustrated in Figure 3.4c. According to Fowler’s list, change method signature can be further divided into subcategories such as add parameter, remove parameter, and introduce parameter object [Fow]. Change method signature is mainly for eliminating code smells such as long method signature, data clumps, and long method [Ker04].

• **Inline method**: The opposite of extract method, inline method is a refactoring type that “puts a method body into its caller and remove the method” [Fow], as illustrated in Figure 3.4b.
Figure 3.4b. The refactoring is intended to eliminate methods that are short and unnecessary [Ker04].

I selected these refactoring types for two reasons: (1) according to Murphy-Hill and colleagues’ previous study of refactoring tool usage, these refactoring types are among the most frequently performed ones [MH09b]; and (2) in my study in Chapter 2, developers introduced the most defects when manually performing extract method and change method signature. For inline method, although I do not have evidence, I speculate that it is as error-prone as extract method because of both refactorings’ similar complexity.

3.4.2 Refactoring Detectors

To check the correctness of refactorings, GhostFactor needs to detect them first. Taking two versions of a source file as an input, each detector outputs whether a refactoring is performed and describes the detected refactoring for later analysis. In this subsection, I present my currently implemented detectors.

**Extract Method.** Our detecting algorithm for extract method is based on the observation that after a refactoring finishes, a new method declaration will contain part of a previously existing method. By comparing the new method against the removed part of the previously existing method, this detector determines whether an extract method refactoring took place. The refactoring detection algorithms are based on string comparisons. Exact string comparisons lead to a higher false negative rate, especially when developers reformat their code while refactoring or interleave several refactorings. Therefore, I designed the refactoring detection algorithms by using approximate string comparisons [Nav01] to improve their flexibility. This does not suggest the GhostFactor technique is by nature probabilistic. The extract method detection algorithm consists of following steps:

1. Given two versions of the same source file (\(v_{old}\) for the old version and \(v_{new}\) for the new one), the detector categorizes method declarations in \(v_{new}\) into two categories: (I) those are also in \(v_{old}\), and (II) those are newly added.
2. In the method bodies of category (I), the detector searches method invocations of methods in category (II).
3. Suppose a method \(m_1\) in category (I) is invoking another method \(m_2\) in category (II), I replace the method body of \(m_2\) to the body of \(m_1\), resulting in a flattened method.
The detector finds the previous version of $m_1$ in $v_{old}$, which I refer to as $m_{old}$.

4. Compute the edit distance between $m_{old}$ and $m_1$ as $ed_1$, and compute the edit distance between $m_{old}$ and $m_f$ as $ed_2$.

5. Compute the edit distance between an empty string and $m_2$ as $ed_3$.

6. An extract method refactoring is detected if and only if $(ed_1 - ed_2)/ed_3 > T$. The default value of $T$ is 0.5.

**Change Method Signature.** For change method signature, GhostFactor warns developers about call sites that fail to be updated after a method signature changes. In most cases, conventional compiler warnings are enough to achieve this, as the example in Section 3.2 illustrated. However, when the changed signature has exactly the same parameter count and types with its previous version, conventional compiler warnings are not able to help, as illustrated in the manual refactoring study in Chapter 2. Our algorithm detects these situations.

Before diving into the detail of my algorithm, I first define parameter access string as follows: For a parameter $p$ in a method declaration $m$, its parameter access string consists of 0 and 1, where position $i$ of the string is 1 iff the $i_{th}$ parameter access in the method body of $m$ is accessing parameter $p$, otherwise position $i$ in the string is 0. The algorithm to detect change method signature performs the following steps:

1. Given two versions of the same source file $v_{old}$ and $v_{new}$, the detector first maps each method declaration in $v_{new}$ with their previous version in $v_{old}$ by matching method names.

2. The detector filters out method pairs whose parameter types and counts differ from each other. For each of the remaining pairs of method declarations $m$ and $m'$, suppose that $m$ has parameters $p_1, p_2, p_3, ...$; $m'$ has parameters $p'_1, p'_2, p'_3, ...$; Parameters on the same positions share the same type, that is $typeof(p_n) == typeof(p'_n)$.

3. For each parameter $p_i$ taken by method $m$, calculate its parameter access string $s_i$.

4. For each parameter $p'_i$ taken by method $m'$, calculate its parameter access string $s'_i$. 
5. For each possible pair of $p_i$ and $p'_j$, calculate the edit distance $ed_{ij}$ between their parameter access strings.

6. For each $p_i$, find the smallest value of all $ed_{ij}$, which I refer as $ed_{ik}$.

7. If $k$ does not equal to $i$, a change method signature is detected.

**Inline Method.** Our inline method detection algorithm is based on the observation that simply replacing the invocations of $n$ with $n$’s method body in $m$’s body before the refactoring will result in a method that is similar to $m$ after the refactoring. The algorithm takes the following steps:

1. Given two versions of the same source file $v_{old}$ and $v_{new}$, the detector first categorizes the method declarations in $v_{old}$ into two categories: those also exist in $v_{new}$, which I refer to as $m_c$; those removed in $v_{new}$, which I refer to as $m_r$.

2. For each pair of $m_c$ and $m_r$ where the former is calling the later, I first find the version of $m_c$ in $v_{new}$, for which I refer to as $m'_c$. Next, the detector flattens $m_c$ by replacing its invocations of $m_r$ with the method body of $m_r$, and I refer the flattened method as $m_f$.

3. The detector calculates the edit distance between $m_f$ and $m'_c$, which I refer to as $ed_1$; then calculate the edit distance between $m_c$ and $m'_c$, which I refer to as $ed_2$; and calculate the edit distance between an empty string and $m_r$, which I refer to as $ed_3$.

4. An inline method is detected if and only if $(ed_2 - ed_1)/ed_3 > T$. I currently set $T$ to be 0.5.

**3.4.3 Condition Checkers**

After detecting a manual refactoring, I next check whether the refactoring preserves the software’s external behavior. Multiple ways exist to check correctness, such as formal verification, testing, and checking refactoring conditions [Opd92]. Formal verification proves the behavior-preserving of software before and after refactorings, however this technique is prohibitively heavy, potentially jeopardizing the goal of providing developers instant feedback [GM06]. Testing, or more specifically regressing testing, can expose software’s behavior changes, though a recent study from Kim and colleagues shows that the test cases associated with a
project are often insufficient to expose refactoring errors [Kim12]. As another option, automatically generating tests instead of using the existing ones likely leads us to the other problems like the object-creating issue [Xia11].

Therefore, I chose to use similar condition checking to existing refactoring tools [Ecl]. Checking refactoring conditions, although ad hoc in nature, has two advantages that other ways do not: (a) checking refactoring conditions is more time efficient, aligning with the requirement of instant feedback; and (b) condition checking allows us to issue informative refactoring warnings convenient for developers to resolve, rather than only telling that the external behavior of the software has been changed.

Similar to the refactoring tools integrated into the mainstream IDEs, condition checkers in GhostFactor check whether a given manual refactoring violates a predefined set of conditions. However, not every condition requires an implementation in GhostFactor, because some violations can trigger conventional compiler errors. Before implementing GhostFactor, I categorized the refactoring conditions into three categories: (1) those whose violations trigger conventional compiler errors; (2) those whose violations do not; (3) those whose violations sometimes do and sometimes do not. Interested readers can find the categorization in my project website\(^2\). In summary, I manually surveyed the 10 most frequently used refactoring types, and none of them have all its conditions in category (1).

GhostFactor dynamically loads condition checkers at runtime to allow developers to easily add new checkers by implementing my predefined interfaces. We selected three refactoring conditions to implement. The criteria for selecting these conditions were: (1) the condition should be associated with a refactoring type that my implementation can detect; (2) violation of the condition does not trigger conventional compiler errors; and (3) violation of the condition should happen frequently according to my study in Chapter 2. In the rest of this section, I present the implementation of the checkers for these three conditions.

### 3.4.3.1 Return Value Checker

When performing extract method, statements to extract may modify the values of some local variables that are accessed after the extracted statements. The modified values of these variables need to be returned from the extracted method; otherwise the code after the extracted statements in the original method will get incorrect values for these variables. Take the code in Figure 3.5 as an example, if I intended to extract statements from line 4 to line 8, the up-

\(^2\)https://sites.google.com/site/ghostfactorstudy/
Figure 3.5 Code before extract method refactoring.

```java
private double InternalFunction(double[] X) {
    // ...
    for (int i = 0; i < this._SimplexVariableList.Length; i++) {
        if (this._SimplexVariableList[i].Fixed == false) {
            this._ExternalVariables[i] = X[varFreeVarIndex] * this._SimplexVariableList[i].ScaleFactor;
            varFreeVarIndex++;
        }
    }
    // ...
}
```

Figure 3.6 Missing return value error.

```java
private double InternalFunction(double[] X) {
    // ...
    for (int i = 0; i < this._SimplexVariableList.Length; i++) {
        LoopBody(X, varFreeIndex);
    }
    // ...
}
```

```java
private void LoopBody(double[] X, int varFreeIndex) {
    if (this._SimplexVariableList[i].Fixed == false) {
        this._ExternalVariables[i] = X[varFreeVarIndex] * this._SimplexVariableList[i].ScaleFactor;
        varFreeVarIndex++;
    }
}
```

dated value of `varFreeVarIndex` needs to be returned and assigned back to the local variable `varFreeVarIndex`. Without returning this value, as illustrated in Figure 3.6, although incorrect as a refactoring, the change will not result in compiler errors. To help developers recognize such errors, I implement a checker for missing return values.

The idea of the return value checker is straightforward: (1) it first applies data flow analysis to find local variables accessed by the code after the extracted statements in the original method; (2) it applies data flow analysis to find local variables written in the extracted statements; and (3) local variables that are found in both steps (1) and (2) need to be returned from the extracted
private double InternalFunction(double[] X) {
    // ...
    for (int i = 0; i < this._SimplexVariableList.Length; i++) {
        varFreeVarIndex = LoopBody(X, varFreeIndex);
    }
    // ...
}

private int LoopBody(double[] X, int varFreeIndex) {
    if (this._SimplexVariableList[i].Fixed == false) {
        this.ExternalVariables[i] = X[varFreeVarIndex] * this._SimplexVariableList[i].ScaleFactor;
        varFreeVarIndex++;
    }
    return varFreeVarIndex;
}

Figure 3.7 Extract method errors fixed.

method. Finally, I check whether the manually extracted method has all these variables returned and assigned back. If not, GhostFactor will issue a refactoring warning for a missing return value at the extracted method’s declaration.

GhostFactor provides a quick fix option with the warning. After the developer invokes the quick fix, a return statement will be added automatically and the returned value will also be assigned back to the modified local variable, as illustrated in Figure 3.7. If multiple return values are needed, GhostFactor makes the parameters pass-by-reference.

3.4.3.2 Parameter Checker

When performing the extract method refactoring, the newly extracted method needs to have correct parameters. Taking the program in Figure 3.5 as an example, to extract code from line 4 to line 8, the extracted method needs to take \( X \) as a parameter. Failing to do so changes the code’s external behavior. In most cases, if the extracted method accesses variables that are not passed, the undefined symbols will trigger compiler errors. However, compiler errors are not always reliable; consider the case when the extracted method needs a parameter whose name is identical to a field of the containing class. Code in Figure 3.8 illustrates this situation.

To ensure the extracted method has the correct parameters, the parameter checker gets the needed parameters by applying data flow analysis to the extracted statements. Next, it checks if
the extracted method has all of these needed variables as parameters. If not, GhostFactor issues a refactoring warning for the missing parameters. Similar to the return value warnings, a warning for missing parameters comes with a quick fix option. After the developer invokes the quick fix, the correct parameters will be added to the declaration of the extracted method and the correct arguments will be added to its call sites. Figure 3.7 illustrates the code after fixing the error.

### 3.4.3.3 Stale Invocation Checker

The change method signature detector described in Section 3.4.2 detects signature changes of a method declaration where a compiler will not issue errors. The stale invocation checker for the change method signature refactoring simply finds invocations of the changed method, and issues warnings to those that have not been updated. The warning also has a quick fix option that can automatically update all of these invocations.

### 3.4.3.4 Modified Variable Checker

This checker aims at identifying two kinds of errors when performing inline method refactorings, which are *incorrectly updated variables* and *incorrectly non-updated variables*. 

---

```
private double[] X; // A field named X exists.
private double InternalFunction(double[] X) {
  // ...
  for (int i = 0; i < this._SimplexVariableList.Length; i++) {
    varFreeIndex = LoopBody(varFreeIndex);
  }
  // ...
}

private int LoopBody(int varFreeIndex) {
  if (this._SimplexVariableList[i].Fixed == false) {
    this.ExternalVariables[i] = X[varFreeVarIndex] * this._SimplexVariableList[i].ScaleFactor;
    varFreeVarIndex++;
  }
  return varFreeIndex;
}
```

Figure 3.8 Missing parameter error.
When performing the inline method refactoring, after the developer replaces the invocation(s) of the inlined method with its method body, any variable updates that were inside the method body may now change variables’ values in its caller, possibly causing the modification to the caller’s behavior. I illustrate this incorrectly updated variable error in Figure 3.9. Notice that in the method body of GoToDefinitionCPlusPlus, the parameter target is updated at line 15. Suppose a developer tries to inline the method GoToDefinitionCPlusPlus, after she replaces its invocation in method GoToDefinition with its method body, the logged value of target at line 6 may differ from its previous version (when target == null and span.IsSome() == true), introducing a subtle bug that compiler warnings will not alert the developer about. Line 8 in Figure 3.10 illustrates this refactoring error.

Another error in inline method refactorings is incorrectly non-updated variables. Also taking the code in Figure 3.9 as an example, the return value of GoToDefinitionCPlusPlus is assigned to a local variable successful at line 3. To guarantee the semantic equivalence, after replacing the invocation with its method body, the developer needs to assign the value

Figure 3.9 Code before inline method refactoring.

```csharp
public void GoToDefinition() {
    // ...
    successful = GoToDefinitionCPlusPlus(text, target);
    // ...
    if (successful)
        logger.Info(target);
}

private bool GoToDefinitionCPlusPlus(ITextView textView, string target) {
    if (target == null) {
        // ...
        var span = wordUtil.GetFullWordSpan(WordKind.NormalWord, caretPoint);
        // Update parameter.
    }
    // ...
    return SafeExecuteCommand(CommandNameGoToDefinition);
}
```
public void GoToDefinition() {

    // ...

    if (target == null) {
        // ...
        var span = wordUtil.GetFullWordSpan(WordKind.NormalWord, caretPoint);
        // Error 1.
    }
    SafeExecuteCommand(CommandNameGoToDefinition); // Error 2.
    // ...

    if (successful) {
        logger.Info(target);
    }
}

Figure 3.10  Inline method errors.

public void GoToDefinition() {

    // ...

    var originalTarget = target; // Fixing error 1.
    if (target == null) {
        // ...
        var span = wordUtil.GetFullWordSpan(WordKind.NormalWord, caretPoint);
        // Error 1.
    }
    // Fixing error 2.
    successful = (SafeExecuteCommand(CommandNameGoToDefinition));
    target = originalTarget; // Fixing error 1.
    // ...

    if (successful) {
        logger.Info(target);
    }
}

Figure 3.11  Inline method errors fixed.

of SafeExecuteCommand to successful. Failing to do so may also cause unintentional changes to the logged message. Line 10 in Figure 3.10 illustrates this inline method error.

To help developers identify these inline method errors, the modified variable checker again performs data flow analysis that consists of the following steps:
1. I refer to the method to be inlined as \( m_i \); the caller of \( m_i \) before inlining is \( m_c \); after inlining \( m_i \), \( m_c \) turns to \( m_c' \).

2. The checker first finds the invocation(s) of \( m_i \) in the method body of \( m_c \), for which I refer to as \( i(s) \).

3. By applying data flow analysis, the checker calculates the variables that are written by \( i(s) \) and read by the statements after \( i(s) \) in the method body of \( m_c \). I refer to these variables as \( v \).

4. The checker next extracts the statements that replace \( i(s) \) in the method body of \( m_c' \), for which I refer to as \( s \).

5. By applying data flow analysis, the checker calculates the variables that are written by \( s \) and read by the statements after \( s \) in the method body of \( m_c' \). I refer to these variables as \( v' \).

6. The checker finally compares the variables in \( v \) and \( v' \), any variables that exist in \( v' \) but not in \( v \) are *incorrectly updated* by the inlined code; any variables that exist in \( v \) but not in \( v' \) are *incorrectly non-updated* after inlining.

After identifying any of these errors, GhostFactor presents a refactoring warning to the developer with the problem declaration, and also provides a quick fix option to help him automatically resolve this issue. After the developer clicks the quick fix option, GhostFactor performs the following changes:

- For any *incorrectly updated variable* \( v \), GhostFactor introduces a new local variable \( v' \) to store the value of \( v \) before the inlined statements, and assigns the value of \( v' \) back to \( v \) after the inlined statements. Line 3 and Line 12 in Figure 3.11 illustrate how GhostFactor resolves the problem in Figure 3.10.

- For any *incorrectly non-updated variable*, GhostFactor first finds out return statements in the inlined method body before refactoring; next, GhostFactor finds the local variables that are supposed to save these returned values after refactoring; finally, GhostFactor inserts statements that assign the returned values to these local variables at proper positions. Line 11 in Figure 3.11 illustrates how GhostFactor resolves this problem in Figure 3.10.
3.4.4 Limitations

In this subsection, I summarize the known limitations of GhostFactor.

Snapshots. GhostFactor’s detection of manual refactorings relies on the snapshots taken from the developer’s code changes, which in turn rely on IDEs’ notifications of such event [Ecl; Vs]. Different IDEs notify plugins about source code changes at different frequency. If the frequency is too low, some significant snapshots may be missing, leading to manual refactorings remaining undetected. To deal with this limitation, I plan to investigate mechanisms that optimize the likelihood of manual refactorings being detected without losing significant computing resources. Another limitation is that currently GhostFactor only compares the snapshots of a single file to detect refactoring. For some refactoring types, such as move method, the refactorings can only be detected by synergically comparing snapshots across file boundaries. I plan to add this feature in future work.

False Positives. Like most static analysis techniques, false positives happen in GhostFactor. Especially when developers interleave their refactoring changes with non-refactoring changes, a violation detected by GhostFactor may actually be what the developer intended [MH09b]. False positives may also lead to stubborn warnings. As I show in Section 3.3, the removal of a refactoring warning relies on the successful detection of a corrected refactoring. If GhostFactor successfully detects an erred refactoring while failing to detect its later correction, a refactoring warning may persist longer than appropriate.

To deal with these false positives, I plan to improve GhostFactor in the following two ways: (1) add a user interface affordance that allows developers to dismiss false positive warnings and (2) design more sophisticated algorithms for refactoring detection utilizing data collected in (1).

When developers dismiss false positives in GhostFactor, the refactoring detection algorithms could adapt future violations accordingly. For instance, when several refactoring errors are consecutively accepted by the developer, she is likely performing root-canal refactorings where non-refactoring code changes are less often interleaved [MH09b]. In that situation, GhostFactor can more aggressively assume detected refactorings are actual refactorings. On the other hand, if refactoring errors consistently marked by the developer as false positives, GhostFactor should lower the sensitivity of refactoring detection to maintain the developer’s confidence in future error messages.

False Negatives. Developers may perform multiple refactorings together, which may hinder GhostFactor from detecting some of them. One example is when a developer first extracts
some statements to a new method and afterwards renames the variables used in these statements. Because GhostFactor detects the extract method refactoring by measuring text similarity, the detection component may not recognize this as a refactoring.

### 3.4.5 Example

I next use an example to illustrate how GhostFactor works. Suppose Susan is a C# developer working on the DotNumeratic open source project [Dot]. She has installed GhostFactor into her IDE. One day, Susan noticed that the method in Figure 3.12 was undesirably long. Therefore, she decided to extract part of the method into a new method. After she manually extracted the code in the box, Susan transformed the code in Figure 3.12 to the code in Figure 3.13. Susan thought she had finished the refactoring correctly. Right before she started coding somewhere else, GhostFactor detected the refactoring by comparing the code in Figure 3.13 and Figure 3.12. Further analysis performed by the return value checker of GhostFactor found that the extracted method failed to return \( s \). After that, GhostFactor issued a refactoring warning to the newly extracted method, as illustrated in Figure 3.14.

Susan noticed this warning message and was aware of this refactoring error. So she invoked the quick fix options associated with the refactoring warning, as illustrated in Figure 3.15. GhostFactor automatically added a return statement and assigned \( s \) back to the proper local variable, resulting in the code shown in Figure 3.16, where the code automatically changed.
by GhostFactor is in boxes. Susan refactored correctly this time with the help of GhostFactor, which afforded her the benefit of refactoring tools without requiring her to explicitly use these tools.

3.5 Evaluation

To evaluate the effectiveness of GhostFactor in improving manual refactoring’s correctness, I conducted a study with participants from both academia and industry. With this study, I intend to answer the following three research questions:

• **Q1.** When manually refactoring, can GhostFactor help developers to refactor more correctly compared to not using it?

• **Q2.** When manually refactoring, can GhostFactor help developers achieve correct refactorings more quickly?

• **Q3.** How can I improve GhostFactor to better assist developers with manual refactorings?
Our study consists of three parts: (1) a pre-study questionnaire that collects participants’ demographic data, (2) several refactoring tasks for participants to manually finish, either with or without GhostFactor, and (3) a post-study questionnaire that collects participants’ opinions on GhostFactor. In this section, I present the design of the study and summarize the study results.

### 3.5.1 Pre-study Questionnaire

In total, I recruited 8 participants, 6 from the computer science department of North Carolina State University and 2 from local IT companies. I required the participants have a college degree in computer science. As compensation, each participant received a 10 dollar gift card after finishing the study. Before I asked them to perform any refactorings, I asked participants to complete a pre-study questionnaire. The questions collected demographic data relevant to my study. These questions include:

1. How many years have you been a programmer?

```c#
void ConsoleReadLine(
    Characters destination,
    int length) {
    string s = Console.ReadLine();
    TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
void TrimString(string s,
    Extracted method needs return value: string s
    if (s.Length > length)
        s = s.Substring(0, length);
}
```

Figure 3.14 GhostFactor error message.
2. How familiar are you with Java programming language? Please rate yourself from 1 to 5, 1 for not at all and 5 for expert.

3. How familiar are you with C# programming language? Please rate yourself from 1 to 5, 1 for not at all and 5 for expert.

4. How familiar are you with refactoring practices? Please rate yourself from 1 to 5, 1 for not at all and 5 for “I refactor every time I program”.

5. What is the percentage of your programming time involving with refactoring? Please specify the percentage.

6. What is the percentage of your refactorings finished by applying refactoring tools? Please specify the percentage.

Table 3.1 presents the participants’ responses, where the number of opaque stars indicates participants’ ratings on scales from 1 to 5. The participants had a median of 5.5 years for
void ConsoleReadLine(
    Characters destination,
    int length) {
    string s = Console.ReadLine();
    s = TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}

string TrimString(string s,
    int length) {
    if (s.Length > length)
        s = s.Substring(0, length);
    return s;
}

Figure 3.16 Code after GhostFactor fixes.

programming experience, a median Java proficiency of ★★★★☆, a median C# proficiency of ★★, and a median refactoring proficiency of ★★★★☆. All participants considered refactoring an integral part of their programming. Two participants, one in the treatment group and one in the control group, indicated that they do not use refactoring tools at all.

3.5.2 Refactoring Tasks

After participants finished the pre-study questionnaire, I asked them to manually finish a set of refactoring tasks in Visual Studio 2010. I did not mention that the purpose of the study is to evaluate my tool; I only informed participants that I were interested in how correct manual refactorings are. Afterwards, I randomly assigned participants to the treatment group (T in Table 3.1) or the control group (C in Table 3.1). The treatment group refactored with GhostFactor’s assistance, while the control group refactored without it. The only difference between these two groups’ development environments was whether GhostFactor was running. Both groups of participants were allowed to use conventional compiler warnings. I also disabled
Table 3.1 Pre-study questionnaire results for both treatment (T) and control (C) group.

<table>
<thead>
<tr>
<th>Years as programmer</th>
<th>Java proficiency</th>
<th>C# proficiency</th>
<th>Refactoring proficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>4</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>⭐⭐⭐⭐ ⋆⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>⭐⭐⭐⭐ ⋆⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>⭐⭐⭐⭐ ⋆⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
<td>⭐⭐⭐⭐⭐ ⋆⭐⭐⭐⭐</td>
</tr>
</tbody>
</table>

GhostFactor’s quick fix options to avoid the bias introduced by automatic code completion, which could enable participants in the treatment group achieve correct refactorings faster.

I selected real code examples from an open source project called DotNumerics [Dot]. Written in C#, DotNumerics implements multiple algorithms for linear algebra, differential equations and optimization problems. I selected this project for two reasons: (1) DotNumerics has a non-trivial code base of over 10K lines of code, in which I easily found real refactoring opportunities; and (2) DotNumerics contains procedures for solving well-known mathematical problems that I expected participants to easily understand.

I selected 6 code examples from DotNumerics where refactoring could be properly performed. I selected these code examples based on five criteria: (1) refactoring these code examples should be particularly error-prone; (2) the refactorings to be performed on these code examples should be of the types detectable by GhostFactor (mentioned in Section 3.4.2); (3) the errors that participants may make when refactoring these code examples should include those checkable by GhostFactor (mentioned in Section 3.4.3); (4) the code examples should not be too complex to refactor manually; and (5) the code examples should not manifest any syntactic differences between C# and Java, as all 8 participants had less experience in C# than in Java.

For all of the code examples used in my study, I refer readers to my project website for detail. Of these 6 code examples, I asked participants to perform the extract method refactoring on 4 and the inline method refactoring on 2. I chose more examples for the extract method refactoring because I have implemented twice as many checkers for that refactoring. I did not include any refactoring tasks for the change method signature refactoring. In exploratory iterations of this study, although participants correctly performed change method signature
refactorings with GhostFactor’s assistance, they refactored exactly as though they were using compiler warnings. This indicated that including these tasks in the study may not produce interesting results, hence I excluded them from the study described here.

The first author led participants through the code examples sequentially and asked the participants to manually complete the refactoring tasks as correctly as possible. Before a participant started to refactor, I reminded her or him of the definition of refactoring correctness mentioned previously in Section 3.4.2. I recorded the screencast of participants’ manual refactorings using CamStudio [Cam]. I used these recordings to compare the treatment and the control group’s refactorings to answer research questions Q1 and Q2.

Q1. With GhostFactor, can developers refactor more correctly? I answer this research question by comparing the correctness of refactorings assisted by GhostFactor and those that were not. Figure 3.17 shows the number of correct refactorings for each task performed by participants in each group. EM stands for extract method and IM stands for inline method. Overall, I collected 24 refactoring assisted by GhostFactor, and 24 refactorings performed without GhostFactor’s assistance.

23 of the 24 refactorings performed with GhostFactor were correct. In contrast, only 7 of the 24 refactorings performed without GhostFactor were correct. GhostFactor performed
especially well when assisting inline method refactoring which no participant in the control group did correctly. GhostFactor improved my participants’ likelihood of performing correct manual refactorings by 66.7% ((23 − 7)/24). To further evaluate the improvement, I performed the Mann-Whitney U test on the collected data, which is a statistic test that assumes neither that the data follows the normal distribution nor the minimum sample size [MW47]. Using R, I found $p$ to be .00326 ($< .01$), which indicates that **GhostFactor significantly improves the correctness of manual refactorings** [R D08].

**Q2. With GhostFactor, can developers achieve correct refactorings more quickly?**

I answer this research question by comparing the time needed to perform correct refactorings with and without GhostFactor. Figure 3.18 plots the average time, in seconds, taken by participants in each group to correctly finish tasks. Since no participant in the control group performed the inline method refactoring correctly, Figure 3.18 does not include these tasks for participants in the control group. According to this data, the participants who used GhostFactor finished faster in three out of four tasks. To further evaluate the time difference, I again apply the Mann-Whitney U test. Using R, I found $p$ to be .343 ($> .01$), suggesting that **GhostFactor may not significantly shorten the time needed to perform correct manual refactorings** [R D08].
3.5.3 Post-study Questionnaire

After participants completed the refactoring tasks, I asked them to take a post-study questionnaire. The post-study questionnaires, administered to participants in the treatment group, collected participants’ opinions about their experiences with GhostFactor. Before they answered these questions, I used the screencast of their refactorings to show the participants which “IDE warnings” they encountered were issued by GhostFactor. This ensured that participants differentiated GhostFactor’s warnings from other warnings in the Visual Studio IDE, such as those for undefined variables and syntax errors.

The post-study questionnaire asked participants to rate their agreement (from 1 to 5, 1 for not at all and 5 for totally agree) with the following statements:

- Refactoring warnings are useful in identifying refactoring errors.
- Refactoring warnings are a desirable feature in IDEs.

Overall, the median rating for participants’ agreement with each of these statements was 4, suggesting that developers found GhostFactor useful in helping manual refactorings and that GhostFactor may be a desirable feature in IDEs.

The second part of the post-study questionnaire was an open question about how I could improve refactoring warnings. Participants raised several interesting points. I use their answers to answer the research question Q3.

Q3. How can I improve GhostFactor? One participant suggested that “refactoring warnings are not intuitively understandable”. Like error messages from conventional refactoring tools [MH07], the warning messages issued by GhostFactor are difficult to understand in the current implementation. One way to tackle this problem is by giving developers code examples, or by visualizing these errors. To meaningfully compare the time to finish the refactorings, I intentionally disabled GhostFactor’s quick fix suggestions during my study, but when developers use GhostFactor in the wild, the quick fix preview (as shown in the gray box in Figure 3.15) may help developers understand refactoring warnings better. However, better ways to convey warnings’ meaning remains an open question for future exploration.

Another participant suggested that “refactoring warnings should come earlier than the refactoring’s end”. He would prefer if refactoring warnings reminded him of possible refactoring pitfalls before or while he refactored, instead of providing corrective messages after he has already made an error. While I agree that showing potential refactoring errors early may
help developers perform correct refactorings faster, knowing a developer’s intent to refactor before he actually starts refactoring is technically difficult, if not impossible.

To better answer this research question, I also examined the refactoring videos of those participants who used GhostFactor, but did not refactor correctly. One participant in the treatment group failed to complete IM2 correctly, as illustrated in Figure 3.17. Delving into the causes of her error, I found that she accidentally deleted a statement in the method where another method should be inlined to. Even though her refactoring passed all of GhostFactor’s condition checkers, my tool failed to detect that the mistakenly deleted statement changed the program’s behavior. This observation suggests a need for new condition checkers that guarantee no code has been unintentionally removed by a manual refactoring.

### 3.5.4 Discussion

Our study showed that GhostFactor can effectively improve the correctness of manual refactorings. I next discuss some other interesting observations.

**Learning Effects.** The data in Figure 3.17 and Figure 3.18 suggests that participants generally performed later refactoring tasks more correctly and faster than earlier ones. This observation holds for participants in both study groups. I speculate that this performance enhancement was due to learning effects: participants may have applied knowledge gained from earlier tasks to the later ones. The knowledge gained might include what code elements need more attention during manual refactoring, the meanings of refactoring warnings, and how to resolve refactoring warnings.

**Attitude towards GhostFactor.** I observed that participants sometimes over rely on GhostFactor warnings. One participant, when refactoring manually with GhostFactor, told the first author that “[Warning] messages are quite informative, I feel like I am not really thinking”. This statement suggests that she feels GhostFactor warnings are guiding her refactorings, rather than correcting them. Ironically, one of the reasons I designed GhostFactor was to decrease developers’ reliance on conventional compiler warnings in refactoring; for this developer, GhostFactor became the tool that she relied on too much. We believe that manually inspecting GhostFactor warnings before addressing them is a better strategy than totally relying on them while refactoring.

In contrast to the over-reliant participant, another participant became much more careful when performing refactoring tasks after GhostFactor first warned him of a refactoring error. He manually examined the correctness of each complete refactoring even when GhostFactor
found no errors. I speculate that using GhostFactor increased his awareness of the error-prone nature of manual refactoring. I did not anticipate this benefit to using GhostFactor.

**Refactoring Time Improvement.** Although the refactoring data I collected suggests that GhostFactor does not improve the time taken to refactor manually, I speculate that, as developers become expert GhostFactor users, they can use GhostFactor’s assistance to refactor faster. Also, to reduce bias introduced by quick fixes, I disabled them in the study. I postulate that developers can correct refactoring errors faster by applying quick fixes. Furthermore, GhostFactor helps developers assess the correctness of refactorings, potentially saving time spent testing and inspecting refactored code. I believe that a more comprehensive, long-term study of GhostFactor use may show that using GhostFactor can improve the efficiency of manual refactorings.

### 3.5.5 Threats to Validity

Although the study gives us confidence about the usefulness of GhostFactor, several threats need to be considered when interpreting the study results.

The first threat is the limited number of refactoring tasks as well as the recruited participants (6 tasks for each of 8 participants), which externally threatens the results’ generalizability to other developers’ refactoring tasks in the wild. Also, the participants’ fair proficiency in refactoring, as suggested in the demographic data in Section 3.5, implies that GhostFactor may not be equally helpful to the developers with more refactoring experience. In addition, by solely studying extract method and inline method refactorings, I cannot conclude that GhostFactor can perform equally well when helping developers perform other types of refactorings.

Another threat is that the tasks I picked can always lead to refactoring mistakes that GhostFactor detects. In spite that these mistakes, according to my collected data in Chapter 2, are frequent among developers, failing to consider other refactorings performed in the wild may lead to the hasty conclusion.

The third threat lies in other reasons why participants in two groups refactor with varied correctness and speed. GhostFactor may not be the only reason for the differences. Since I randomly assign participants to the two groups, the participants in the treatment group may by themselves have better programming skills than those in the control group, allowing them to refactor more correctly and quickly regardless of GhostFactor’s existence. To eliminate this threat, in the future, I plan to assign participants according to their reported expertise to the groups under study so that the knowledge gap between the groups are minimized.
3.6 Future Work

I have implemented the GhostFactor technique as a plug-in Visual Studio IDE. Although the initial study yields promising results, I plan to further explore the possibility of adopting GhostFactor as an integral part of modern IDEs. In this section, I summarize possible future work.

**Better tool.** At this point, GhostFactor only supports three types of refactorings, namely extract method, change method signature and inline method. Existing IDEs usually support more than 20 different refactoring types [Ecl]. To improve the usefulness of GhostFactor, adding more refactoring types is necessary. Also, as the cornerstone of GhostFactor, we plan to improve the refactoring detection algorithms in terms of reducing the false negative and the false positive rates; thus developers can benefit from the tool more often without being frequently disrupted by spurious warnings.

Currently, GhostFactor only supports checkers for refactoring errors that were manifested in my previously conducted studies. However, these errors may be only a small fraction of all the refactoring errors developers could make. In order to better guarantee refactoring correctness, we need to investigate more refactoring error patterns. I plan to apply data mining techniques to software repositories to find these error patterns. In the future, I also plan to summarize a catalog for frequent refactoring errors like FindBugs does for commonly occurring defects.

**Richer Study.** To evaluate GhostFactor, I conducted an in-lab study session participated by 8 developers and collected limited amount of data. To further investigate the benefit of GhostFactor as well as its limitations, a field study to observe what happens when professional developers use GhostFactor under in real-world development can be beneficial. In such a setting, developers frequently interleave refactorings with non-refactorings, potentially leading to GhostFactor’s false positives and false negatives; measuring these two indicators are important to evaluate GhostFactor’s effectiveness.

Another goal of this richer study is to investigate the refactoring detection algorithms. In my controlled human study, the detection algorithms successfully detected all of the manual refactorings performed by the participants. However, when used in the wild, the detection algorithms’ performance is still an open question, such as their precision and recall. The richer study may also compare GhostFactor with other tools assisting developers’ refactorings, such as DNDRefactoring [Lee13] and WitchDoctor [Fos12], in terms of the usability and the use-
fulness. Although these tools assist refactoring in different ways, such a study could compare developers’ effort saved during refactoring.
Chapter 4

ReviewFactor

With a common purpose of improving refactoring correctness, BeneFactor and GhostFactor both use refactoring detection techniques. BeneFactor detects the starting point of a manual refactoring and proposes to finish it automatically; GhostFactor, on the other hand, detects a manually finished refactoring and finds errors in it. In this chapter, I will show that the refactoring detection techniques do not only help implementing refactoring, but also helps developers understand code changes after refactorings being implemented.

In software development, code review is a quality-improving and knowledge-transferring practice where developers manually inspect one another’s code changes. Unfortunately, code review tools treat behavior-preserving changes, or refactorings, and behavior-altering changes, or non-refactorings, the same way, so developers have to spend effort differentiating between the two before they can evaluate the behavioral impact of a given commit. I designed and implemented a refactoring-aware code review tool, called ReviewFactor, that differentiates the refactoring and non-refactoring parts and allows developers to focus on the latter. To evaluate ReviewFactor, I conducted a case study on two open-source projects and a survey study of 35 developers. My case study suggests that ReviewFactor detects refactorings in 39% of the commits, and identifies 4.6% of the total lines of code change as refactorings. My survey study suggests that a refactoring-aware code review tool could help developers review code more quickly and correctly. Moreover, the survey study also shows what information needs a refactoring-aware code review tool should meet to best support software developers.
4.1 Introduction

Code review is a commonly-adopted practice to ensure code quality in both open-source and industrial contexts [BB13]. Code review consists of the examination of code by developers other than the author to ensure the codebase remains correct and understandable [BB13; KP09]. In addition to enhancing code quality, code review helps developers transfer knowledge among team members, and helps newcomers become familiar with new codebases [BB13]. Code review should not be confused with the more formal practice of code inspection or code walkthrough, which requires multiple team members to meet and examine the codebase one line after another [BP94].

Code review tools are widely used to support developers conducting code reviews. For instance, Gerrit facilitates online code reviews for projects that use the Git version control system [Ger]. As another example, Google’s Mondrian tool allows code reviews to be conducted entirely on the web [Mon]. By using these tools, software developers can conveniently send code review requests, comment on each others’ code changes, send comments to the authors, and reject or approve the changes asynchronously, without physical meetings.

These code review tools assume every line of code change deserves reviewers’ attention equally: the changed files are simply displayed side by side with the changed lines highlighted. However, based on the observation that different categories of code changes impact the codebase differently, I reason that behavior-preserving code changes, or refactorings, deserve code reviewers’ attention less than non-refactoring changes do [Fow].

More specifically, code reviewers can invest less effort in reviewing refactorings than non-refactorings because: (1) the most important goal of code review is to find defects [BB13]; however, properly-executed refactorings do not modify the software’s external behavior, so they cannot introduce defects; (2) once the motivation of the refactoring is clear, such as updating a class name to a more readable one, a reviewer can easily infer how the refactoring modifies the codebase; therefore developers do not have to review every line of code change contributing to the refactoring; and (3) other potential goals of reviewing refactorings, such as checking conformance to coding conventions, can be easily automated.

This chapter investigates the feasibility and potential usefulness of the refactoring-aware code review tools proposed by Kim and colleagues in their study on refactorings in Windows 7 [Kim12]. In particular, I investigate whether differentiating the refactoring and the non-refactoring parts in code changes is technically feasible, and if the differentiation could
improve a developer’s effectiveness and speed in reviewing that change. Before implementing a refactoring-aware code review tool, I conducted a formative study with 35 real-world developers to find out the developers’ attitudes toward refactoring changes during code review.

Applying the knowledge from the formative study, I propose and implement a refactoring-aware code review tool called ReviewFactor. ReviewFactor takes two versions of the source code as input and finds the refactoring part of the code change by analyzing log files and applying refactoring detection algorithms. When displaying the code change to code reviewers, ReviewFactor can differentiate the refactoring and non-refactoring parts, if the code reviewer chooses to. To evaluate ReviewFactor, I conducted two studies: the first study evaluates the impact of being refactoring-aware during code review; the second study evaluates the precision and recall of the refactoring detection algorithms in ReviewFactor. In summary, I make the following contributions:

- A formative study of 35 developers, described in Section 4.3. The study shows that, while reviewing refactorings is usually necessary, most developers think automatically identifying refactorings in code review could be useful.

- The approach and the open-source implementation, described in Section 4.4, of a refactoring-aware code review tool called ReviewFactor.

- A case study, described in Section 4.5, to determine the effectiveness of ReviewFactor in detecting manual refactorings in two open-source projects. The study shows that, on average, 4.6% of lines of code change can be identified as refactorings.

- A case study, described in Section 4.6, to evaluate the refactoring detection algorithms used by ReviewFactor. The study measured the precision and recall of ReviewFactor and found the causes of the false positives and negatives among the detected refactorings.
4.2 Motivation

I next motivate ReviewFactor with a real-world code example that reflects my experience in conducting code reviews. Suppose Liam is a software developer working on the TuxBlocks open-source project [Tux]. One day, a code review request from Emma, another developer working on TuxBlocks, arrives in his inbox\(^1\). Liam clicks the link and starts reviewing this change using Eclipse [Ecl]. Figure 4.1 shows part of this change, where the left and right sides, respectively show the code before and after Emma’s change. In addition to aligning lines, Eclipse also highlights the changed parts in lines.

As a meticulous developer, Liam reviews these highlighted lines one after another until he feels comfortable approving or rejecting the change. Later, Liam realizes that the only changes on some highlighted are caused by renaming `inverse` to `block`. Liam has to identify the refactoring part before he can effectively review the code change. Refactoring, as shown in this case, not only visually distracts code reviewers, but also places extra cognitive burden on them.

Based on this observation, I posit that code review tools could reduce visual distraction and cognitive load by allowing reviewers to automatically separate of the refactoring and non-refactoring parts of a code change. To test this speculation, I propose and implement a refactoring-aware code review tool, called ReviewFactor.

To facilitate further discussion, I call the code on the left side the codebase before change, the code on the right the codebase after change, the developer who made the code change, in this case Emma, the developer, and the developer who reviews the code change, in this case Liam, the code reviewer. I call projects, source packages, type declarations, and all other nodes in abstract syntax trees software entities. Each entity belongs to a generic entity type such as project, package, or the various node types in abstract syntax trees. For the node types in abstract syntax trees, I use the terms provided by Eclipse Java development tools, including class declaration, method declaration, statement, qualified name, simple name and so on [Jdt]. All software entities in a codebase comprise a codebase tree whose root is the project and whose leaves are literal abstract syntax tree nodes.

\(^1\)http://go.ncsu.edu/n2jxgc
<table>
<thead>
<tr>
<th>ID</th>
<th>Statements</th>
<th>Never</th>
<th>Rarely</th>
<th>Sometimes</th>
<th>Often</th>
<th>All the Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>The refactoring part slows down my evaluation of the non-refactoring part</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>S2</td>
<td>The refactoring part makes my correct evaluation of the non-refactoring part difficult</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>S3</td>
<td>Automatically identifying the refactoring part could accelerate my review process</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>S4</td>
<td>Automatically identifying the refactoring part could help me evaluate the code change more correctly</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>S5</td>
<td>Under time pressure, I want to review the non-refactoring part first</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>S6</td>
<td>Not reviewing the refactoring part is acceptable</td>
<td>14</td>
<td>16</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### 4.3 Formative Study

Before designing a tool to allow code reviewers focus more on the non-refactoring part, I conducted a formative study to answer the following research questions:

- **RQ1**: What are real-world developers’ opinions towards identifying refactorings in a code change under review?

- **RQ2**: Why do developers review refactoring changes, and how do their reasons for doing so differ from their reasons for reviewing other changes?

- **RQ3**: When do code reviewers ignore refactoring changes?

I found developers to recruit by mining the source repository of the Gerrit open-source project, identifying 189 developers who submitted code changes to the code review tool [Ger]. I assumed that these developers are likely to be frequent code reviewers too, since they contribute to the development of a code review tool. Of these 189 developers, 24 of them had unreachable email addresses, so only 165 Gerrit developers received survey invitations.

The study consists of three parts: questions about demographics, questions asking for agreement with a number of statements and free-response questions [Ge14]. The demographics questions asked the length of developers’ experience in software development in years, how often developers conduct code review, and how often they refactor their codebase. The statement agreement rating questions asked respondents to rate how often they think several statements about the relationship between refactoring and code review are true. Respondents responded
Table 4.2 Respondents’ frequency for activities.

<table>
<thead>
<tr>
<th>Activities</th>
<th>Never</th>
<th>Rarely</th>
<th>Sometimes</th>
<th>Often</th>
<th>All the Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refactoring</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>Code review</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>17</td>
<td>14</td>
</tr>
</tbody>
</table>

to each of these questions with one of: never, rarely, sometimes, often, and all the time. The second column of Table 4.1 shows the statements in this section. In the third part of the survey, I asked respondents to answer two free-response questions regarding refactoring’s role during code review.

4.3.1 Demographics

Of the 165 developers invited to participate in the survey, 35 developers responded and finished the survey (response rate 21%). The length of respondents’ experience as software developers had a median of 13 years, with a minimum of 3 years and a maximum of 35 years. Respondents’ self-reported experience with refactoring and code review are shown in Table 4.2. Almost all respondents reported conducting code review and refactoring at least sometimes.

4.3.2 Results

RQ1. The dichotomy’s usefulness. S1, S2, S3 and S4 helped us answer this research question. In S1, 94% (33) of respondents indicate that refactorings slows down their reviews of the non-refactoring changes at least sometimes. Similarly, in S2, 86% (30) of respondents indicate that refactoring makes their evaluation of the non-refactoring change less correct at least sometimes. Both questions indicate that refactorings hinder code review, either in terms of speed or quality, which confirms my motivation for ReviewFactor as stated in Section 4.2.

Developers’ responses to S3 and S4 indicate that automatic identification of refactorings would assist code review. In response to S3, 91% (32) of respondents said automatically detecting refactorings could accelerate their review process at least sometimes. Similarly, in response to S4, 86% (30) of respondents said that such a technique could help them review code changes more correctly at least sometimes. In summary, my answer to RQ1 is:
RQ1: Developers believe that identifying refactorings in a code change under review could improve their effectiveness and efficiency of reviewing the change.

RQ2. The purposes of reviewing refactorings. In response to S6, 86% (30) of respondents said they think refactorings can rarely or never be ignored during code review. This indicates that refactorings, though they are behavior-preserving, still attract code reviewers’ attention. To delve into the reasons developers review refactorings, I asked the developers this free-response question: “During code review, what information can you get from reviewing refactorings?”.

In total, 25 respondents answered the question. Their answers fall into two categories: detail purposes, which could only be achieved by reviewing the details of the refactoring; existence purposes can be achieved by simply knowing that the change is a refactoring. In the following discussion, I summarize the responses to the question, with quotes from respondents in italics.

For detail purposes, eight respondents reported that reviewing refactorings helps transfer knowledge about design principles and coding conventions, so future contributions can meet these standards:

I hope to share design principles, coding conventions and better programming practices for the team when I refactor code. I would like others to read the code changes and use the same way of coding in the future.

For detail purposes, four respondents reported that they review refactorings to make sure the refactoring actually improves the quality of the codebase:

Good refactoring changes means simplification of the code, to make it more understandable and more readable. If during review it’s not obvious, then something wrong with proposed code.

For detail purposes, four respondents reported that reviewing refactorings helps them view the codebase in a different way, which helps them find problems that were less obvious before being refactored.

For existence purposes, three respondents reported they would reject the code change under review if it contained a mixture of refactoring and non-refactoring changes:
Refactoring… should nevertheless never be part of a review where functionality has been altered. The refactoring in itself does not hold any information, refactoring should be done to make information (code functionality) more visible in and make the code easier to read.

For existence purposes, three respondents reported that refactoring is an indicator of previous, unsuccessful code review sessions. For these respondents, heavily refactored code implies that previous code reviewers did not successfully identify the issues in previous code changes, requiring the developer to address those issues by refactoring. In summary, my answer to RQ2 is:

RQ2: In addition to the known purposes of conducting code review [BB13], developers review refactorings for several other reasons: (1) refactorings conveys design decisions and conventions, (2) the motivation for refactoring needs to be justified, (3) refactoring exposes previously hidden problems, (4) refactoring indicates that a single change has a mixture of multiple purposes, and (5) refactoring warns code reviewers about the quality of previous code review sessions.

RQ3. When to ignore refactorings. Respondents’ answers to S6 suggest that they rarely ignore refactorings when they review code changes. However, in their answers to the other free-response question, which is “During code review, when do you ignore refactorings?”, some respondents indicated that ignoring refactoring is acceptable in some circumstances. In total, 27 respondents answered the question. 16 of them simply indicated that they do not ignore refactorings under any circumstances. I next summarize the responses from the remaining 11 respondents; the quoted responses are shown in italics.

Four respondents felt they could ignore rename refactorings:

…if it is only limited to these changes: renaming class name…, or renaming member variables.

Four respondents reported that they could ignore refactorings if they merely rearranged software entities’ layout in the source code:
If the refactoring is extremely unlikely to change behavior... for example, rearranging members in a class...

In addition, two respondents reported that they could ignore small refactorings, and one respondent reported they could ignore automatic refactorings. Thus, in summary, my answer to RQ3 is:

RQ3: Developers rarely ignore refactorings during code review. However, some refactorings, such as rename refactorings, reordering software entities, and obviously automatic refactorings, are more likely to be ignored than other refactorings.

4.3.3 Implications

The results of my survey show that identifying refactorings in a code change may help developers review that change more correctly and more quickly. In addition, the results indicate some guidelines for building refactoring-aware code review tools.

Separating refactoring and non-refactoring. During code review, developers have different concerns for refactoring changes and non-refactoring changes. Therefore, separating them allows developers to focus on one part without the distraction from the other. Also, reviewers can focus on refactoring-specific information, such as those discovered in my answer to RQ2, when they choose to specifically review refactorings. One caveat of the refactoring-awareness is that code review tools cannot entirely discard the refactoring part due to its lower priority than the non-refactoring part.

Prioritizing different refactorings. Although code reviewers are interested in reviewing both refactoring and non-refactoring part of a given change, they prioritize the non-refactoring part over the refactoring part, especially under time pressure. Also, some refactorings are less informative than others, as I discussed in my answer to RQ2. Using this result, a refactoring-aware code review tool needs to categorize some refactorings as ignorable, such as small renames, reordering fields/methods in a class declaration, and the refactorings that are applied by tools.
A textual summary of refactorings. When developers review refactorings for an existence purpose, they only need to know that a refactoring happens, without further investigating details of the refactoring. Therefore, I speculate that a textual summary of the refactorings in a code change under review can help code reviewers by providing a high-level summary of changes in the change under review, rather than depending on them to determine the nature of the change by manual inspection.

Quantifying a refactoring’s effectiveness in improving code quality. One reason for developers’ review of refactorings is to decide whether such a change is justified. If the refactorings do not improve the code’s quality, the reviewer may reject the change in which they are contained. However, the refactoring’s effectiveness can potentially be measured by calculating software metrics, which is amenable to automation [Sim01]. By leveraging these techniques, tools may save code reviewers manual effort and help them make good design choices for the quality of their code.

The results show that automatically separating the refactoring and the non-refactoring parts of a code change into two commits can please strict code reviewers. These reviewers prefer to have each commit serve a singular purpose: it either functionally extends or structurally improves the existing codebase. However, existing study shows that developers frequently interleave the refactoring and the non-refactoring changes [MH09b]. I envision that allowing developers to interleave these two types of changes, but automatically separating them when submitting for code review, could allow developers to continue interleaving refactorings with other changes, then automatically create clean, single-purpose commits.

4.4 Approach

The formative study suggests that separating the refactoring and non-refactoring part expedites code review, and that code reviewers have different concerns towards the refactoring and non-refactoring part. Guided by these findings, I further designed and implemented a refactoring-aware code review tool called ReviewFactor. ReviewFactor can automatically separate the refactoring and non-refactoring part in a given change, and allow code reviewers focus on one type of change at a time.

Various ways to separate the refactoring and non-refactoring part exist. The current implementation incorporates two-phase code review: during Phase I, the code review tool only highlights the non-refactoring part of the change; during Phase II, the code review
tool only highlights the refactoring part. The reason that ReviewFactor only shows non-refactoring/refactoring in Phase I/II is that code reviewers do not need to visually separate them further, so that they can focus on evaluating the highlighted part as if it is the only change. This section mainly focuses on how ReviewFactor implements Phase I.

ReviewFactor works in three stages: the automatic refactoring replay, the manual refactoring detection, and the refactoring validation. The input of ReviewFactor is the codebase before change, the codebase after change, and the log file of the developer’s refactoring tool usage. Its output is a code review user interface allowing reviewers to differentiate the refactoring and non-refactoring parts of a change under review and focus only on the latter.

Figure 4.2 depicts how ReviewFactor’s components interact with each other. Given the codebase before change and the log file, ReviewFactor first replays the logged automatic refactorings that was played by the developer, generating an intermediate codebase. Next, ReviewFactor detects manual refactorings by comparing the intermediate codebase with the codebase after the developer’s change. Finally, ReviewFactor validates the detected manual refactorings by checking a set of conditions. The output of these components is the refactoring part of the developer’s code change. Using the refactoring part, ReviewFactor displays the code change and gives the reviewer the option to differentiate the refactoring and the non-refactoring part.

Figure 4.2 ReviewFactor components.

https://github.com/DeveloperLiberationFront/edu.dlf.refactoring.review
4.4.1 Replaying Automatic Refactorings

The essential task of ReviewFactor is determining which part of a code change consists only of refactorings. The refactoring part could be further categorized as the manual refactoring and the automatic refactoring, and ReviewFactor identifies them using different strategies. Refactoring tools, such as those integrated into the Eclipse IDE, keep a log of the commands a developer uses, including refactoring commands [Ecl]. Thus, ReviewFactor finds the automatic refactorings a developer used by mining the refactoring tools’ log files. A log entry of an automatic refactoring includes the refactoring’s type, the software entity being refactored, and the user input configuring the refactoring. Using the log entry, I can easily replay the refactoring.

Not all automatic refactorings are reflected in the code change under review. For instance, a developer may rename a class to a new name, then later rename the class back to the original name. Although both renames have been logged by refactoring tools, neither are reflected in the code change under review. To account for situations such as these, ReviewFactor analyzes the log files and codebase before and after change and filters out the intermediate refactorings; thus, ReviewFactor will only collect the refactorings appearing in the code change under review. For instance, ReviewFactor collects a logged rename refactoring only if the original name and the updated name both appear in the codebase before and after change, respectively. After collecting the automatic refactorings, ReviewFactor replays the automatic refactorings on the codebase before change, resulting in an intermediate codebase.

4.4.2 Detecting Manual Refactorings

At the next stage, ReviewFactor detects manual refactorings. The current implementation of ReviewFactor detects manual refactorings of the following five refactoring types: rename type, rename method, move method, extract method and inline method [Fow]. Types in Java could be classes, interfaces or enums. The selection of these refactoring types are based on the following criteria: (1) according to Murphy-Hill and colleagues’ study, these refactoring types are among the most frequent ones [MH09b]; and (2) these refactoring types, especially rename type and method, can change many lines across many files in the codebase, so code reviewers may benefit more from identifying them.

ReviewFactor detects manual refactorings by comparing the intermediate codebase with the codebase after change. By using the intermediate codebase instead of the codebase before change, ReviewFactor ensures the detected refactorings are manual. This means their correct-
ness must be further validated (the validation process is detailed in Section 4.4.3). In Chapter 2, I proposed a manual refactoring detection algorithm that monitors the developer’s code editing behavior; ReviewFactor uses a similar technique combined with software entity mapping. The mapping allows ReviewFactor’s detection algorithm to work efficiently on a large code change without knowledge of the developer’s detailed edit steps towards the change.

(a) Code structure Mapping

(b) Mapping at the method level

(c) Mapping at the statement level

Figure 4.3 Examples of mapping software entities.
4.4.2.1  Mapping Software Entities

Given the intermediate codebase and the codebase after change, ReviewFactor first maps the software entities from the former to the software entities in the latter, using the similar but simplified algorithm with Change Distilling [Flu07]. The mapping determines what entities are changed and how they are changed [Kim07; KN06]. To facilitate discussion, I will call a non-existing software entity a null entity, and an existing entity a solid entity. For each changed entity, the change can be one of: added, removed and updated. In my mapping rules, an added entity is a null entity mapping to a solid entity; a removed entity is a solid entity mapping to a null entity; and an updated entity is a solid entity mapping to another solid entity with the underlying source changed.

To better illustrate the mapping algorithm, I define mapped entities as an entity pair having the following attributes: (1) one entity comes from the intermediate codebase and the other from the codebase after change; (2) the two entities have the same entity type; for instance, they are both class declarations; and (3) their respective parents are a pair of mapped entities. The mapping algorithm processes the codebase trees recursively from the roots to the leaves. More specifically, the algorithm takes the following steps to collect mapped entities:

1. The root of the intermediate codebase tree maps to the root of the the codebase tree after change.

2. Suppose entity $e_i$ and $e_a$ are from the intermediate codebase and the codebase after change, respectively; if $e_i$ maps to $e_a$, ReviewFactor conducts from the steps 3 to the step 5 on their children of the identical entity type. If $e_i$ and $e_a$ have no children; that is, if they are literals, $e_i$ maps to $e_a$ and the mapping algorithm stops.

3. Given two sets of software entities of the identical type, ReviewFactor first maps the entities in the first and the second sets by name. Two entities with identical names are mapped, such as two classes both named “event” or two methods “getTime”.

4. ReviewFactor next tries to map the remaining entities by content. The definition of content varies with the entity types. For example, the content of a package is the contained types’ names, and the content of a statement or an expression is the source code. If the content similarity is above a threshold, ReviewFactor maps the two entities.
5. For the remaining entities, ReviewFactor maps each of them with a null entity. After processing all entities, ReviewFactor repeats from step 2 to step 5 on each pair of mapped entities that contain only solid entities.

Figure 4.3a shows an example of the mapping results. Software entities are represented as rounded rectangles. Rectangles with dashed borders represent entities in the intermediate codebase, and rectangles with solid borders represent entities in the codebase after change, with mapped entities superimposed on one another. The dashed edges connecting rectangles indicate a parent-child relationship between entities in the intermediate codebase, and the solid edges indicate the same relationship between entities in the codebase after change. Mapped entities, such as Method Declaration 21, correspond to updated entities in the codebase, and entities that map to null entities, such as Method declaration 11 and Method declaration 22', correspond to those added or removed from the codebase. Similarly, Figure 4.3b depicts the mapping of the different entities in the method declaration level; Figure 4.3c depicts the mapped entities at the statement level.

After mapping all software entities in the intermediate codebase and the codebase after change, ReviewFactor generates a mapping tree with pairs of mapped entities as nodes. Because ReviewFactor detects refactorings only on the changed code, the tool next trims the mapping tree by removing the non-changing pairs. The trim algorithm post-visits the nodes of the map tree and removes a visited node under if the entities of the pair are both literals and have the identical source code, or if the entity pair has children and all of the children are removed. For example, Figure 4.4a depicts the mapping tree of the code change in Figure 4.3b after trimming.
Figure 4.4 Mapping tree examples.
4.4.2.2 Searching on Mapping Trees

ReviewFactor next detects manual refactorings on the mapping tree. I coded the supported refactoring types as partial tree structures whose appearance in the mapping tree indicates the refactoring of the corresponding type. I call these partial tree structures PTSs. For instance, Figure 4.4b shows the PTS for detecting extract method refactorings, where in a single type declaration, several statements map to null entities and a null entity maps to a method declaration. As another example, Figure 4.4d depicts the PTS for detecting move method refactorings in a source package, where, in a single source package, two type declarations respectively have a method declaration map to a null entity and a null entity map to a method declaration. Similarly, Figure 4.4c, Figure 4.4e, and Figure 4.4f depict the PTSs for detecting inline method, rename type and rename method, respectively.

By searching for these PTSs in the mapping tree, ReviewFactor identifies the instances of these PTSs as refactoring candidates. However, these refactoring candidates are not necessarily refactorings; for example, an instance of the move method PTS depicted in Figure 4.4d could be the removal and adding of two different methods in two unrelated type declarations.

To further refine these candidates, ReviewFactor conducts a text-based similarity test on the collected instances [Bie11]. Passing the similarity test indicates that the changed software entities in a PTS instance are cohesively related, thus they make up a meaningful refactoring.

Given two software entities, the similarity test calculates the ratio between the text distance and the average length of the entities’ underlying source code [Nav01]. If the ratio is below a threshold, the candidate refactoring passes the similarity test and ReviewFactor considers the candidate a refactoring; otherwise, ReviewFactor discards the refactoring candidate. Table 4.3 presents the configuration of the similarity test for the supported refactoring types. Because ReviewFactor detects refactorings conservatively to prevent non-refactoring changes from being incorrectly highlighted in Phase II, the current implementation uses these low threshold values. Optimal thresholds for the similarity test deserve future investigation.

4.4.3 Validating Manual Refactorings

After passing the similarity test, a refactoring candidate is considered by ReviewFactor as a detected manual refactoring. However, my study in Chapter 3 shows that manual refactorings are error-prone. Thus, simply ignoring them during code review may lead to unnoticed defects. To ensure the detected manual refactorings are indeed behavior-preserving, ReviewFactor vali-
Table 4.3  Similarity test examples.

<table>
<thead>
<tr>
<th>Type</th>
<th>Entity Before</th>
<th>Entity After</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract</td>
<td>removed statements</td>
<td>added method’s body</td>
<td>0.1</td>
</tr>
<tr>
<td>Inline</td>
<td>removed method’s body</td>
<td>added statements</td>
<td>0.1</td>
</tr>
<tr>
<td>Move</td>
<td>removed method</td>
<td>added method</td>
<td>0.1</td>
</tr>
<tr>
<td>Rename</td>
<td>method’s body</td>
<td>method’s body</td>
<td>0.2</td>
</tr>
<tr>
<td>Rename type</td>
<td>type’s body</td>
<td>type’s body</td>
<td>0.2</td>
</tr>
</tbody>
</table>

dates them by performing two checking procedures: the refactoring condition checking and the refactoring integrity checking. ReviewFactor only considers a refactoring candidate a refactoring if it passes both of the checking procedures.

Checking refactoring conditions. Refactoring tools automatically transform the codebase under development without altering its external behavior. The correctness of an automatic refactoring can be ensured by either formally proofs of correctness, or heuristic-based checking. The former is impractical for most use cases, so refactoring tools usually adopt the latter technique, called refactoring condition checking [Ecl; Opd92; OJ11; SM10; Sch09]. For instance, after extracting statements to a new method, the new method must return the value used by the code below the extracted statements in the original method, and the new method must take the parameters accessed by the extracted statements. Another example is when renaming a local variable in the body of a method, the new name must not mask any fields in the containing class.

Shown in Chapter 3, Refactoring conditions can be further categorized as compiler-noisy ones and compiler-silent ones; the violation of the compiler-noisy conditions cause the codebase uncompilable, while the compiler silent conditions do not. Code changes developers submit for code review usually needs to be compilable; therefore, ReviewFactor only checks the compiler-silent conditions. When implementing ReviewFactor’s condition checking component, I reused some source code from the Eclipse IDE’s refactoring tools [Ecl]. If a detected manual refactoring fails any of these conditions, ReviewFactor discards the refactoring. Otherwise, ReviewFactor considers the refactoring is indeed behavior-preserving.

Checking refactoring integrity. In addition to checking the refactoring conditions suggested by conventional refactoring tools, ReviewFactor also needs to check the integrity of the refactoring, that is, whether the refactoring includes superfluous changes or omits necessary changes. Either case leads to the refactoring failing of the integrity checking. I implemented
unique integrity checks for each refactoring type. For example, for extract method refactorings, an extra statement, other than the necessary return statement, appearing in the new method but not in the original method is a superfluous change; a statement in the original method that is not extracted to the new method is a necessary change that has been omitted. Similarly, for rename type, a type reference that is not updated to the new name is an omitted necessary change, and leads to a failed integrity check.

![Figure 4.5 Beyond-file refactoring hiding.](image)

### 4.4.4 User Interface

ReviewFactor combines the validated manual refactorings with the logged automatic refactorings to collect the refactoring part of a given change. The rest of the change is the non-refactoring part. During code review, ReviewFactor differentiates the refactoring part with the non-refactoring part by only highlighting the latter, or *hiding* the former. Certainly other types of user interfaces can differentiate the refactoring and non-refactoring parts, we use hiding...
specifically to align with Eclipse’ ignore white-space functionality. To hide refactorings, the reviewer needs to click a toggle button.

The hiding consists of two levels: the beyond-file level and the in-file level. The beyond-file hiding level hides files whose changes are all refactorings. The in-file hiding level hides specific changed lines in a single file where the refactoring and the non-refactoring changes mix.

For the different refactoring types, ReviewFactor hides different code changes contributing to the refactoring; for instance, for a rename refactoring, ReviewFactor hides all the updated lines that change the original name to the new name. Table 4.4 illustrates the hidden code changes with respect to the currently supported refactoring types, where the parenthesized trailing letter A, R, and U indicates added, removed, and updated lines, respectively. For code reviewers, hiding refactorings can be optional, depending on the implementation of ReviewFactor’s user interface.
Figure 4.5 shows an example of beyond-file hiding. ReviewFactor shows multiple files changed by the developer. Some contain only refactorings, as indicated by “R” icons decorating the file icons. Since the reviewer knows that, for instance, “DlfFileUtils.java” contains only correct refactorings, she can choose not to open the file for inspection.

Figure 4.6 illustrates an example of in-file hiding. In this code change, ReviewFactor collected two rename refactorings: (1) renaming the class JUnitCore to JUnitCoreClass and (2) renaming the field fNotifier to fNotifierRefRem. ReviewFactor highlights all changed code, except the lines that only contribute to these refactorings. For a line of code change containing both refactoring and non-refactoring change, ReviewFactor does not hide the entire line but, only the refactoring part of the line. For instance, when a developer renames a method declaration and also adds an extra parameter to the declaration, both of these two changes applied to the same line of code. In this situation, ReviewFactor hides the renaming while keeping the added parameter highlighted. Thus, the code reviewer may choose to review the behavior-modifying parts of the code change without being distracted by refactorings.

Figure 4.5 and Figure 4.6 only demonstrate my current implementation of ReviewFactor’s user interface; there are certainly other alternatives, such as differentiating refactoring and non-refactoring changes with different highlighting colors. I leave designing the most effective user interface for refactoring-aware code review tools as future work.

4.4.5 Limitations

ReviewFactor has the following three limitations that may reduce its effectiveness.

**Mixing different refactorings.** ReviewFactor cannot detect refactorings that mix with each other in the same code snippet. For instance, a developer may extract a group of statements to a new method and then rename a local variable in the new method. Usually, ReviewFactor can detect neither of these refactorings because: (1) for the extract method refactoring, the renamed local variable may lead to the failure of the similarity test; and (2) for the rename local variable refactoring, the variable in the new method cannot be mapped with an to variable, since the containing method is new, so the entities have different parents and cannot be mapped. Thus, in this situation, ReviewFactor cannot detect any refactorings.

**Floss refactoring.** Existing study shows that developers often interleave refactoring changes with non-refactoring ones [MH09b]. Generally, ReviewFactor works fine if these two change types are not deeply mixed. However, in some mixtures, ReviewFactor cannot effectively detect refactorings for the reasons mentioned previously. For instance, if a developer
Table 4.4 Hidden code change for different refactoring types.

<table>
<thead>
<tr>
<th>Refactoring Type</th>
<th>Hidden Entities before Change</th>
<th>Hidden Entities after Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract method</td>
<td>statements (R)</td>
<td>method declaration (A), method invocations (A)</td>
</tr>
<tr>
<td>Inline method</td>
<td>method declaration (R)</td>
<td>statements (A), method invocation (R)</td>
</tr>
<tr>
<td>Move method</td>
<td>method declaration (R), method invocations (U)</td>
<td>method declaration (A), method invocations (U)</td>
</tr>
<tr>
<td>Renames</td>
<td>names (U)</td>
<td>names (U)</td>
</tr>
</tbody>
</table>

extracts some statements to a new method, then adds several new statements to the extracted method, ReviewFactor cannot detect the extract method refactoring.

**Refactoring tools’ reliability.** ReviewFactor depends upon existing refactoring tools’ condition checking mechanism. However, existing study shows that these refactoring tools are not always reliable [Soa13]. Defects in refactoring tools may result in validated refactorings that are actually behavior-altering, thus leaving them unnoticed by reviewers.

### 4.5 Impact Study

The answer I collected in the formative study shows that separating the refactoring and non-refactoring part could accelerate code review. However, two potential undesirable outcomes of using ReviewFactor may prevent the tool from being useful: (1) the detected refactorings only consist a tiny fraction of a given change, thus separating them cannot significantly help code reviewers; and (2) ReviewFactor frequently categorizes lines of code change that are behavior-altering as refactorings or vice versa. Therefore, the evaluation of ReviewFactor consists of two parts: (1) to measure the impact of being refactoring-aware during code review, and (2) to measure the part of a given change that is falsely categorized as refactoring/non-refactoring after detecting refactorings.

To evaluate the effectiveness of ReviewFactor in identifying refactorings in code changes under review, I ran my tool on the revisions of two popular open-source projects, namely the JUnit [Jun] unit testing framework and FBReader for Android [Fbr], a mobile e-book reader app. I chose these projects from GitHub’s list of the most watched Java projects. We read this list from the top and chose the first two projects that met the following criteria: (1) the commit history of the project was complete and publicly available; (2) the project had a non-trivial
codebase consisting of more than 10,000 lines of code; (3) multiple developers had contributed to the project; and (4) the project contained a ‘’.project’’ file, allowing us to easily import it to the Eclipse IDE. By applying ReviewFactor on these two projects, I aimed at answering the following research questions:

- RQ4: What is the percentage of commits that ReviewFactor can detect refactorings in?
- RQ5: What is the percentage of code change, in terms of lines of code change, that can be detected by ReviewFactor as refactorings?

4.5.1 Study Setting

Due to the lack of code review data, the open-source project study uses what-if analysis on these selected projects; that is, assuming the developers of these projects use ReviewFactor to review each submitted code change, I retrospectively measured proportion of changed lines it could distinguish as containing only refactorings. To quantify this the measurement, given a specific commit, I define the refactoring ratio as the refactoring lines divided by the total lines of code change.

I first cloned the repository of a project under study to my local machine. Next, I checked out multiple versions of the project in the machine from the latest commit to the initial one, consecutively. To avoid name collisions, I renamed each version of the project to its commit number. After checking out these project versions, I imported them into the Eclipse IDE with ReviewFactor installed.

My what-if analysis assumes developers review each commit to the codebase, so I ran ReviewFactor on every pair of consecutive project versions. According to the criteria in Table 4.4, I calculated the number of refactoring lines of changed code that ReviewFactor could identify as refactorings. For simplicity, I did not distinguish between the three types of code change that are adopted by the existing code review tools, which are updated, added and removed lines [Ger]. I aggregated the numbers of added, removed, and updated lines detected, and counted the sum as the number of refactoring lines of the code change. I did not count lines of code change containing both refactoring and non-refactoring parts as refactoring lines.
After calculating the number of refactoring lines, I next calculated the total lines of code change in each commit. Even though the project’s Git repository gives the exact size of the commit, I cannot use the data directly because Git only calculates two types of code change: added lines and removed lines, as illustrated in Figure 4.7. In contrast, code review tools, such as Gerrit [Ger], calculate updated lines as well, as illustrated in Figure 4.1. If I only considered the added and removed lines, we could overestimate the lines of code change in a commit. I reused Gerrit’s diff algorithm to calculate the lines of code change for a given commit and aggregated the number of the added, removed, and updated lines as the total lines of the commit.

4.5.2 Study Results

As of March 1st, 2014, JUnit and FBReader had, respectively, 1774 and 6461 commits. For my study, I downloaded 1500 commits from each project. Note that some commits change only non-Java files, such as configuration files. I skipped these commits and downloaded 1500 commits containing changes to Java source code. After downloading these commits, I ran ReviewFactor to detect and validate refactorings in them. ReviewFactor detected refactorings in 628 to the JUnit project and 541 commits to the FBReader project. This suggests that 39% ((628 + 541)/(1500 + 1500)) of code review sessions can benefit from ReviewFactor, assuming every commit needs to be reviewed before the developer can push the change to the repository.
RQ4: On average, ReviewFactor detects refactorings in 39% of the commits under review.

![Graph](image)

(a) Commit counts for refactoring ratio ranges

(b) Contribution by refactoring types

**Figure 4.8** Open-source projects study results.
I also calculated the refactoring ratio for each commit. My results suggest that on average, ReviewFactor detects 4.6% of the lines of code change as refactoring. For those commits where ReviewFactor detects some refactorings, Figure 4.8a plots the numbers of commits falling into the specified ranges of refactoring ratios. For example, JUnit and FBReader have 94 and 116 commits, respectively whose refactoring ratios are between 10% (exclusively) and 20% (inclusively). As suggested in the Figure 4.8a, in more than half of the commits with non-zero refactoring lines, ReviewFactor can detect less than 10% of lines of code change as refactorings. As the refactoring ratio increases, the number of commits in that range decreases radically. Note that Figure 4.8a does not include the commits that contain no refactoring lines, that is, those whose refactoring ratio is 0%.

RQ5: When a code reviewer evaluates only the functional impact of a given commit, ReviewFactor reduces the lines of code change that need to be reviewed by an average of 4.6%.

I also investigated how much different refactoring types contribute to the refactoring lines of code change. In summary, I found a similar pattern in the two projects under study. For ReviewFactor’s currently supported refactoring types, rename type refactorings contribute the most refactoring lines, followed by rename method, move method, extract method and inline method refactorings. Figure 4.8b plots these refactoring types’ relative contributions in detail. For example, in the JUnit project, 41% of the lines of detected refactoring changes are rename type refactorings; similarly, for the FBReader project, rename type refactorings make up of 49% of the total lines of detected refactorings.

I propose two possible explanations for the prevalence of renames: (1) rename refactorings cross file boundaries more often than in-class refactorings such as extract method and inline method; and (2) compared to rename refactorings, ReviewFactor’s detection algorithms for extract method and inline method may be overly strict, leading to fewer instances of these refactoring types being detected.
Detecting 4.6% of the lines of code change as refactorings may seem insignificant. However, I collected the refactorings without analyzing the log files associated with refactoring tool usage, so some automatic refactorings may go undetected. Also, I only evaluated one refactoring detection algorithm; future versions of ReviewFactor could use other algorithms and may detect more. The third reason to explain this seemingly small percentage is that I only supported five refactoring types; in the future, extending ReviewFactor to cover more refactoring types may lead to more refactoring candidates detected. The fourth reason may be due the conservativeness of my validation mechanism. Future refinements on this component could also help ReviewFactor validate more refactoring candidates as actual refactorings. In summary, my study data reflects the state of ReviewFactor’s current implementation, but refactoring-aware code review tools in general may perform more effectively in the future.

### 4.6 Precision Study

In addition to studying how many commits can benefit from the refactoring-awareness of the code review tool, I also investigated whether the detected refactorings are behavior-preserving, assuming the code reviewer would like to review the functional part first. To answer this question, I applied my refactoring detection algorithms on the revisions of the JUnit open source project; for the detected refactorings, I manually examined whether they were false positives, that is, whether these changes alter the external behavior of the source code. In addition to the false positives, I also evaluated the false negatives of ReviewFactor, that is, how many refactorings performed by the JUnit developers failed to be identified as refactoring by ReviewFactor. In summary, I aimed at answering the following research questions:

- **RQ6.** What is the false positive rate of the refactoring detection algorithms in ReviewFactor?
- **RQ7.** What causes the false positives of the refactoring detection algorithms in ReviewFactor?
- **RQ8.** What is the false negative rate of the refactoring detection algorithms in ReviewFactor?
- **RQ9.** What causes the false negatives of the refactoring detection algorithms in ReviewFactor
4.6.1 Study Setting

I conducted the study on the JUnit open source project [Jun]. The reasons for choosing this subject are same to the study described in Section 4.5.

The first step of collecting the refactorings in the JUnit source code is to locate the revisions that update the Java source files. Using JGit, I wrote a script to traverse the revision history of JUnit. For a visited revision, the script pulls the changed file names and checks if any of the names are with the “.java” extension. If such files exist, meaning that the revision updated java source files, I record the revision number for later use. Because manually examining these revisions to detect refactorings is a labor-intensive task, I only collect one hundred revision numbers.

Using the recorded revision numbers, the second step is to check out the JUnit codebase reflecting those changes. For a given revision number, I check out the codebase immediately before the revision number and the exact revision of the revision number. Next, I import these two codebases into the Eclipse IDE and run the refactoring detection algorithms in ReviewFactor on them. The output of the refactoring detection is a log file including the information of the detected refactorings. The information facilitates human interpretation instead of machine manipulation; thus, I can easily read the refactoring and find them in the revision manually. For instance, for a rename refactoring, the log file contains the name before and after the rename; for an extract method refactoring, the log file contains the new method name.

After getting the log files, I wrote a script to automatically analyze the log files to build a HTML page facilitating later analysis. The HTML page helps me easily navigate from a ReviewFactor-detected refactoring to the actual diff from where the refactoring is detected. Figure 4.9 illustrates a snapshot of the HTML page. Each line indicates one refactoring detected by ReviewFactor. For instance, the first line in Figure 4.9 indicates a rename type refactoring whose original name and new name are Notifying and RunTestNotifier respectively. Clicking the line will navigate to the Github page where the diff is presented; in this case, the URL is https://github.com/junit-team/junit/commit/de43b277b8f2e63051368a36e27a35c5ed0f0e0b. The generation of the URL is straightforward. Since I have the revision number, the URL is the string concatenating https://github.com/junit-team/junit/commit/ and the revision number.
After creating the HTML page, I can use the page to evaluate the false positives conveniently. For each detected refactoring, I click the URL associated with it to navigate to the diff on Github. Using the refactoring information, I can easily find the exact location of the detected refactoring; after finding the location, I manually judge whether the detected refactoring is actually behavior-preserving, using the same inspection methodology adopted by Murphy-Hill and colleagues [MH09b]. If a detected “refactoring” alters the behavior of the codebase, I mark the “refactoring” as a false positive. Section 4.6.2 details the detected refactorings that fall into the false positives. To evaluate the false negatives of ReviewFactor, I manually went through every commit of the collected one hundred revisions, trying to identify the refactorings that are missed by ReviewFactor. Section 4.6.2 summarizes the findings. The detail of the collected data is also publicly available.

4.6.2 Study Results

In this section, I detail the study results by answering each research question.

**RQ6. False Positive Rate.** To validate the behavior-preservation of the refactorings detected by ReviewFactor, I manually reviewed the refactorings detected by ReviewFactor on Github. Among the 227 detected refactorings, 17 of them fall into the category of false positives, meaning that they alter the behavior of the codebase. Delving into the refactoring types, I found 16 “rename” refactorings are false positives; 1 “extract method” refactoring is a false positive; for inline method refactoring and move refactoring, I have not found any false pos-

---

3[https://sites.google.com/site/reviewfactorstudy/](https://sites.google.com/site/reviewfactorstudy/)
Table 4.5 Data for ReviewFactor’s refactoring detection algorithms.

<table>
<thead>
<tr>
<th>Refactoring Type</th>
<th>True Positives</th>
<th>False Positives</th>
<th>False Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract method</td>
<td>20</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Inline method</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Move method</td>
<td>42</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rename</td>
<td>144</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>Overall</td>
<td>210</td>
<td>17</td>
<td>13</td>
</tr>
</tbody>
</table>

itives. As the data presented in Table 4.5, the current implementation of ReviewFactor has a precision of 92.5% \((210/(210 + 17))\) and a recall of 94.2% \((210/(210 + 13))\).

RQ7. Causes of False Positives. Among the 186 detected refactorings, 18 refactorings are false positives. To decide if a particular refactoring is a false positive, the author manually examined the code change. The majority of the false positives (17) are from the rename detectors; only one instance is from the extract method detector; for the move refactoring detector and the inline method detector, I have not found any instances of false positives. To delve into the exact causes of the false positives, I manually examined each instance.

The first cause of false positives is **interleaving refactoring with non-refactoring**. To detect a rename method refactoring, ReviewFactor maps the methods before and after change by their contents; if the contents share similarity beyond a predefined threshold, the name change of the methods is considered as a rename method refactoring. However, since ReviewFactor does not perform semantic analysis on the method body, the content change may significantly alter the behavior of the method, resulting in the name change not only rename but also reflecting behavior change. To illustrate this case, consider the code example shown in Code 3 and Code 4.\(^4\) Method `testIsNull` in Code 3 appears in the revision before change while Method `testIsZero` appears in the revision after change. The change is detected by ReviewFactor as a rename method refactoring because of the content similarity and the updated name. However, the name change of the method reflects the content change as well. Thus, this is a not a pure refactoring but a combination of refactoring and non-refactoring. To eliminate false positives of this case, ReviewFactor needs to perform semantic analysis in the method content to further decide whether the method behaves identically before and after the change.

\(^4\)Commit: 4b1869ebbb002e5d0b82ab55460f6126043c9ec4

98
As another example of the interleaving refactoring with non-refactoring case, consider the code example shown in Code 5 and Code 6, appearing in the codebases before and after change, respectively. ReviewFactor detects a rename refactoring from `testStarted` to `testFinished` because of the similar contents after mapping the two methods. However, the name change reflects more than a refactoring but indicates that the method will be called at the different time; the important behavior change needs to be noticed by code reviewers. To eliminate the false positives like this, ReviewFactor may carefully map the method declarations that override several declarations in the super class.

The second cause of the false positives is updated library usage. ReviewFactor detects rename refactorings also by references, especially when the declaration of the renamed entities

---

5Commit: 77ecd2119d3124b72935038e0f7b98eccd9
@Override
public void testStarted(Description description) throws Exception {
    fCount++;
}

Code 5  Method before change

@Override
public void testFinished(Description description) throws Exception {
    if (!fTestIgnored)
        fCount++;
    fTestIgnored = false;
}

Code 6  Method after rename and content change.

is unavailable. However, if ReviewFactor does not delve into the declaration, the updated references may actually bind to different software entities in the library, leading to false positives reported. To illustrated this case, consider the examples in Code 7 and Code 8, respectively appears in the codebases before and after change. In this case, ReviewFactor reports a rename type refactoring from “out” to “err”. However, what the developer intents to do is to redirect the error message from the standard output stream to the standard error stream. As another example for the case of updated library usage, consider the code shown in Code 9 and Code 10, respectively from the codebase before and after change. ReviewFactor considers a rename method refactoring happens when the developer updates “JScrollPane” to “ScrollPaneConstants”; however, the developer actually corrected a misused API in the Swing library.

To eliminate the false positives of this case, ReviewFactor may ignore all detected rename refactorings whose corresponding declarations are unavailable. Alternatively, ReviewFactor can explore the source code of the libraries used in the project under analysis to confirmed whether the declared names are updated as well.

---

6Commit: 255f47231f5e49373d4b331e3a946e0005f1e0f4
The third cause of the false positives is **co-evolving of the type and identifier of a variable**. In this case, ReviewFactor first maps an old identifier of a variable to a new identifier in the declaration. ReviewFactor next checks the references of the declaration, finding that all references are updated accordingly. Therefore, ReviewFactor reports a rename variable refactoring. However, my tool fails to check whether the declared type of the variable stays the same. Thus, if the type changes as well, the rename refactoring is potentially a false positive. Exceptionally, updating the type to one of its super types does not lead to a false positive, for instance, to update “ArrayList” to “Iterable”.

To illustrate this case, consider the code example shown in Code 11 and Code 12, extracted from the codebases before and after the change, respectively. ReviewFactor detects a rename parameter refactoring from “constructor” to “theClass”. However, since the type of the parameter also changes, the behavior of the method “addTestMethod” alters accordingly. To

---

**Commit: 0807a8cd01b6d63d54f9a3580540fa35b7fb292e**
eliminate the false positives like this, I need to add a checker to the rename variable detector in ReviewFactor. The checker shall filter out all detected refactorings whose declared types are updated.

In addition to the false positives in the rename detectors, ReviewFactor also reports several false positives in the extract method detector. The cause is that the developer introduces an anonymous class to enclose several statements that were used to be inside another method; therefore, ReviewFactor mistakes the enclosed statements to the extracted ones. To illustrate
private void addTestMethod(Method m, Vector names, Constructor constructor) {
    String name= m.getName();
    if (names.contains(name))
        return;
    // ...
}

Code 11 Co-evolving type and identifier before change.

private void addTestMethod(Method m, Vector names, Class theClass) {
    String name= m.getName();
    if (names.contains(name))
        return;
    // ...
}

Code 12 Co-evolving type and identifier after change.

this case, consider the code examples in Code 13 and Code 14, which are from the codebase before and after change, respectively. Based on mapping, ReviewFactor considered the method run as an added method and the statements inside aboutToStart were removed. Further comparing the removed statements with the body of the added method, ReviewFactor detected an extract method refactoring. To eliminate the false positives like this case, ReviewFactor needs to check whether the “extracted method” is actually called in the method declaration from where the statements are extracted.

In summary, my answer to RQ7 is:

---

8Commit: 4cf0c7d9db0f7b29f65d9dab047fef9805dae671
protected void aboutToStart(Test testSuite) {
    for (Enumeration e= fTestRunViews.elements(); e.hasMoreElements(); ) {
        TestRunView v= (TestRunView) e.nextElement();
        v.aboutToStart(testSuite, fTestResult);
    }
}

Code 13 Extract method false positive before change.

protected void aboutToStart(final Test testSuite) {
    SwingUtilities.invokeLater(
        new Runnable() {
            public void run() {
                for (Enumeration e= fTestRunViews.elements();
                    e.hasMoreElements(); ) {
                    TestRunView v= (TestRunView) e.nextElement();
                    v.aboutToStart(testSuite, fTestResult);
                }
            }
        });
}

Code 14 Extract method false positive after change.

RQ7. ReviewFactor detects “refactorings” that are actually behavior-altering because of (1) the interleaving of refactoring and non-refactoring changes, (2) mistaking updated library usage to rename, (3) co-evolving of the types and identifiers, and (4) not handling anonymous classes.

RQ8. False Negative Rate. To investigate how many refactorings performed by the developers were not detected by ReviewFactor, I manually went through the one hundred commits"
public void testAssertNaNEqualsNaN() {
    assertEquals(Double.NaN, Double.NaN, 0.0);
}

Code 15 Rename false negative before change.

under study and tried to find the undetected refactoring that fall into the refactoring types supported by the current implementation of ReviewFactor. In summary, I found 13 false negatives. Among these undetected refactorings, 7 of them were rename; 4 of them were extract method; 1 was move; and 1 was inline method.

RQ9. Causes of False Negatives. All of the seven false negatives I found for the rename detectors are because the interleaving of refactoring and non-refactoring changes makes mapping incorrect. To illustrate this case, consider the examples in Code 15 and Code 16, respectively from the codebases before and after change. The developer renamed a method from testAssertNaNEqualsNaN to testAssertNaNEqualsNaNFails. When trying to map the methods in the containing class, ReviewFactor does not map the two methods as the same method with the different names but one removed method and one added method, due to the similarity of the method body is below the predefined threshold.

As another example of the interleaving of refactoring and non-refactoring changes leading to false negatives, consider the code examples in Code 17 and Code 18, which are respectively from the codebases before and after change. The developer renamed a method called addError to testField, and updated the method body to handle both errors and failures. However, instead of mapping these two methods, ReviewFactor considers addError is a removed method while testField is an added method, leading to a rename method not being detected.

9Commit: a19a2d6c0c454c83d368a561252bf7f4b202f8ea
10Commit: 8817825612894d11856312a777281b6c7cf81672
1 public void testAssertNaNEqualsNaNFails() {
2     try {
3         assertEquals(Double.NaN, Double.NaN, 0.0);
4     } catch (AssertionFailedError e) {
5         return;
6     }
7     fail();
8 }

Code 16 Rename false negative after change.

1 public void addError(final Test test, final Throwable t) {
2     SwingUtilities.invokeLater(
3         new Runnable() {
4             public void run() {
5                 fCounterPanel.setErrorValue(fTestResult.errorCount());
6                 appendFailure("Error", test, t);
7             }
8         });
9 }

Code 17 Rename false negative before change.

More than the rename detectors, the interleaving of refactoring and non-refactoring changes leads to the false negatives of the detectors for other refactoring types as well. For the extract method detector, consider the code snippets in Code 19 and Code 20, which are respectively from the codebases before and after change. The developer extracted `fParameters.get(fParameterSetNumber)` to a new method called `computeParams` with a try-catch block to handle any exceptions internally. However, ReviewFactor failed to identify this extract method refactoring because the removed expression is textually only a small fraction of the newly declared method. To eliminate the false negatives like this case, ReviewFactor needs to adopt more sophisticated similarity tests that ignore the trivial difference like adding or removing a try-catch block.

\[\text{Commit: 2807ff63d6b408a927fd3dd684a42a2366677657}\]
public void testFailed(final int status, final Test test, final Throwable t) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            switch (status) {
            case TestRunListener.STATUS_ERROR:
                fCounterPanel.setErrorValue(fTestResult.errorCount());
                appendFailure("Error", test, t);
                break;
            case TestRunListener.STATUS_FAILURE:
                fCounterPanel.setFailureValue(fTestResult.failureCount());
                appendFailure("Failure", test, t);
                break;
            }
        }
    });
}

public Object createTest() throws Exception {
    return fConstructor.newInstance(fParameters.get(fParameterSetNumber));
}

For the only false negative I found for the move method detector, the cause is also the interleaving of refactoring and non-refactoring changes. Consider the code snippets shown in Code 21 and Code 22, which are from the codebases before and after change, respectively. In this example, the developer removed a class enclosing the method `suite`, essentially moving the method from `Compatibility` to `AllTests` and next deleting `Compatibility`. However, ReviewFactor failed to detect the move method refactoring, because according to

12Commit: aa7047aacc6dee8bc7f07da92ffa4a661737ca63
public Object createTest() throws Exception {
    return fConstructor.newInstance(computeParams());
}

private Object[] computeParams() throws Exception {
    try {
        return fParameters.get(fParameterSetNumber);
    } catch (ClassCastException e) {
        throw new Exception(String.format("%s.%s() must return a Collection of arrays.", fTestClass.getName(), getParametersMethod().getName()));
    }
}

Code 20  Extract method false negative after change.

public class AllTests {
    public static class Compatibility {
        public static Test suite() {
            return new JUnit4TestAdapter(AllTests.class);
        }
    }
}

Code 21  Move method false negative before change.

my mapping rules to detect such refactorings, both the original and the new containing classes need to exist. To eliminate the false negatives like this, ReviewFactor needs to generalize the mapping rules to allow a removed containing class.

Another cause of the false negatives is that the composite of multiple refactorings lead to ReviewFactor’s failing to detect either one of them. Developers usually perform several refactorings together [MH09b; Pre10]; for instance, she may first move a method from a class to another class and next rename the method. In the study, I found one case where a
rename and an inline method refactoring happened together, so that ReviewFactor failed to detect either of them. Consider the code snippets shown in Code 23 and Code 24, respectively from the codebases before and after change. In this example, the developer first inlined the method `addToAppropriateEnd` to the caller `addToAnnotationList`, next she renamed the local variable `list` to `methods`. Ideally, ReviewFactor should detect the inline method refactoring, however it reports no refactoring in this change. Delving into the cause, although ReviewFactor mapped `addToAnnotationList` with its later version and identified `addToAppropriateEnd` as a removed method, the diff failed the similarity test due to the updated name of `list`. In other words, ReviewFactor does not consider the method body of `addToAppropriateEnd` was integrated into the method body of `addToAnnotationList` because of the noise introduced by a rename refactoring.

On the other hand, ReviewFactor cannot detect the rename refactoring either. To detect the rename from `list` to `methods`, ReviewFactor needs to map their containing software entities first, in this case, the methods `addToAppropriateEnd` and `addToAnnotationList`. However, due to the methods with the name `addToAnnotationList` exist in both the codebases before and after change, these two methods mapped, leaving the method `addToAppropriateEnd` map to null. Therefore, ReviewFactor cannot detect the rename refactoring.

In summary, my answer to RQ9 is:

---

13Commit: 9f52d1c93e0f8246ab5430007107e0c2599563dd
private void addToAnnotationList(Class<? extends Annotation> annotation, FrameworkMethod testMethod) {
    ensureKey(annotation);
    addToAppropriateEnd(annotation, testMethod);
}

private void addToAppropriateEnd(Class<? extends Annotation> annotation, FrameworkMethod testMethod) {
    List<FrameworkMethod> list = methodsForAnnotations.get(annotation);
    if (testMethod.isShadowedBy(list))
        return;
    if (runsTopToBottom(annotation))
        list.add(0, testMethod);
    else
        list.add(testMethod);
}

Code 23 Inline method false negative before change.

RQ9. ReviewFactor fails to detect refactorings because of (1) the interleaving of refactoring and non-refactoring changes, and (2) the composite of multiple refactorings.

4.6.3 Threats to Validity

Although I have shown that ReviewFactor can achieve comparatively high precision and sensitivity, several threats may prevent the generalization of my study results.

Internally, the first threat is my lack of a golden set of refactoring and non-refactoring changes. To measure the false positives of ReviewFactor, I have to manually examine each detected refactoring to evaluate its behavior-preservation. On the other hand, to measure the false negatives, I have manually examined each commit of JUnit where I applied ReviewFactor to identify the refactorings missed by my tool. Manual refactoring detection may involve bias and mistakes, potentially resulting in imprecise data collected. Alternatively, Prete and colleagues
private void addToAnnotationList(Class<? extends Annotation> annotation, FrameworkMethod testMethod) {
    List<FrameworkMethod> methods = getAnnotatedMethods(annotation);
    if (testMethod.isShadowedBy(methods))
        return;
    if (runsTopToBottom(annotation))
        methods.add(0, testMethod);
    else
        methods.add(testMethod);
}

Code 24 Inline method false negative after change.

used the code examples in Fowler’s refactoring catalog as a golden set [Pre10]. However, I believe that applying ReviewFactor on actual codebases leads to more realistic results.

Externally, I only examined one hundred commits in the repository of JUnit. Although JUnit is a popular and well written tool for software developers, the sample size may be overly small, preventing the study results from being generalized. Also, some language features in Java, such as anonymous classes, may limit my study results to be Java-specific. For other programming languages, ReviewFactor may perform differently. In addition, I studied only five refactoring types; after integrating more supported refactoring types, the precision and sensitivity of ReviewFactor may vary.

4.6.4 Discussion

Code review is a chance for developers to find defects and transfer knowledge. Because developers sometimes intend to evaluate the functional impact of a given change, refactoring-aware code review tools have the potential of enhancing the efficiency and effectiveness of this practice. However, automatically detecting refactoring is a nontrivial task. If a false positive happens, that is, several lines of non-refactoring change have been highlighted during Phase II instead of Phase I, finding the defects lying in these lines may also be postponed. Therefore, the key attribute of the refactoring detection technique in ReviewFactor is the high precision. Ideally, the precision shall be 100% implying no false positives at all.

In the JUnit study, I show the precision of the current implementation of ReviewFactor is 92.5%. Although the false positives still happen, the precision is higher than most known refac-
toring detection techniques. For instance, Prete and colleagues proposed RefFinder and conducted several studies to evaluate its precision and recall [Pre10]. The precision of RefFinder on Fowler’s code examples of refactorings is 97%. However, when applying on real-world codebases, the precision of RefFinder is 74%, lower than that of ReviewFactor. As another example, Dig and colleagues’ refactoring detection technique shows the precision of 100% in two of the projects under study and the precision of 90% in the third project [Dig06]. Although Dig and colleagues’ technique may have the higher overall precision than ReviewFactor, their technique handles only component-level refactorings.

In summary, my current evaluation shows that the combination of conventional refactoring detection techniques and condition checking mechanisms can lead to a refactoring detection technique with the comparatively higher precision. The precise refactoring detection algorithm is suitable for applications like code review tools, where the false positives lead to the worse consequence than the false negative do.

### 4.7 Future Work

Although my study shows some promising results, there are still many open questions about ReviewFactor that deserve future study. In addition to the implications shown in Section 4.3.3, I invite the software engineering community to explore the following problems:

**Two-phase review.** According to my formative study reported in Section 4.3, developers rarely ignore the refactoring part in a given code change. Therefore, a comprehensive refactoring-aware code review tool shall separate the refactoring and non-refactoring part in a given change without emphasizing on either of them. The current implementation of ReviewFactor only supports highlights the non-refactoring part. In the future, I plan to implement the other phase of the refactoring-aware code review, which is highlighting the refactoring part.

**Richer study.** My evaluation of ReviewFactor is based on a what-if analysis, rather than an evaluation of use by code reviewers. To fill this gap, researchers should investigate how code reviewers react to ReviewFactor when reviewing actual code changes in professional settings. I hypothesize that code reviewers with different personalities will find ReviewFactor more or less useful to them, based on existing evidence suggesting that some code reviewers are more meticulous than others [DCG07].

**Measuring Review Difficulty.** My formative study in Section 4.3 suggests that the interleaving of refactoring and non-refactoring may slow down code reviewers’ evaluation on either
of them. ReviewFactor is useful because it separates these two parts automatically for developers, however developers still need to manually review each of them. As a piece of potential future work, we can empirically measure the effort a code reviewer spends on reviewing the refactoring part, the non-refactoring part, and separate these two parts respectively, to more precisely evaluate the improved efficiency by using ReviewFactor.

**Difference among refactoring types.** As shown by my study results, the different refactoring types contribute differently to the number of refactoring lines. In the future, ReviewFactor could be expanded to support more refactoring types. Then, further research could explore the difference between those types and determine which refactoring types save the most effort when detected. In addition, I am also curious about code reviewers’ opinions towards different refactoring types. Although my survey study suggests that some refactoring types are more likely to be ignored than others, I do not have a theory to explain this phenomenon. Future empirical work could determine what refactoring types are most important to developers, in general and in different circumstances.

**Other refactoring detection algorithms.** In the present chapter, I extend and reuse GhostFactor’s algorithms for refactoring detection. However, this component can be replaced by other algorithms, such as RefFinder and RefactoringCrawler [Pre10; Dig06], as long as the detected refactorings are validated before being visually presented to code reviewers. Comparing my study results reported in Section 4.6 with the studies reported in RefFinder and RefactoringCrawler, the precision of ReviewFactor is higher than that of both RefFinder and RefactoringCrawler. However, to further test this hypothesis, I plan to run all three tools on the same set of code revisions.

**Refactoring tools’ another benefit.** Tool-supported refactorings have been shown to be more reliable than manual ones [MH09b]. In addition to this known benefit, using refactoring tools often might help developers review code changes as well, since the refactorings that are logged by these tools can assist ReviewFactor to highlight them during Phase II. Unfortunately, this chapter did not evaluate this component of ReviewFactor. In the future, I plan to investigate refactoring tools’ role in helping code review practice in more detail.

**Prioritizing lines of code change.** Based on the observation that not all code changes are created equal, ReviewFactor reduces the code reviewers’ burden by identifying refactorings. Extending this idea leads to prioritizing the lines of code change in general. Fault localization researchers proposed similar techniques: when developers are debugging, lines of code can be ranked by their likelihood of containing defects [PO11]. In the future, I plan to investigate
whether prioritizing lines of code change can improve the code reviewers’ effectiveness and efficiency.
Chapter 5

Related Work

Due to the necessity of continuously improving the internal structure of software systems, refactoring is an intensively studied area in both academia and industry. In this section, I will briefly summarize the existing work as well as their relationship with my dissertation. I categorize the existing work into refactoring theory, refactoring opportunities, source differencing techniques, refactoring detection, refactoring studies, refactoring tools, refactoring conditions, and code review.

5.1 Refactoring Theory

Software is highly prone to changes. Due to the ever-evolving environments and requirements, developers have to maintain their codebase continuously to make the software function well. Failing to do so may lead to software decay or even the discard of the entire system [Eic01; Par94]. Although software design can somehow predicts the future changes and incorporate them, the unanticipated changes happen regularly and raise the maintenance cost to an unbearable level. If the cost of maintenance keeps adding up, rebuilding the software may be a more cost-efficient solution than continuing to maintain it.

Maintenance tasks fall into two categories: the functional and the nonfunctional. The functional maintenance adds new features to the software system to meet the new requirement, or fixes the defects in the system to more reliably support the old requirement. On the other hand, the nonfunctional maintenance improves the design of the software system to incorporate emerging changes. Refactoring, for instance, belongs to the nonfunctional maintenance.
Before introducing the term “refactoring”, researchers and developers used “restructuring” to refer to the same activity. For instance, Griswold’s dissertation addressed how software restructuring can assist the software maintenance process [Gri92]. Griswold proposed several types of “meaning-preserving transformations” including moving an expression, renaming a variable, inlining an expression and so on. Furthermore, Griswold applied the graph transformation to ensure that local structure changes do not break the global features, thus the automation of the restructuring is technically feasible.

The term of “refactoring” first appears in Opdyke’s dissertation which defines a set of program restructuring operations that support the design, evolution and reuse of object-oriented application frameworks [Opd92]. Opdyke observed that to facilitate the in-project code reuse, a developer has to make some preparations on her codebase, and these preparations fall into several predictable patterns. For instance, generalizing methods into a super type prepares the addition of another subtype. Similarly, developers adopt the convergent strategies to simplify logics, such as introducing new subtypes. To summarize these behavior-preserving code changes, Opdyke presented a catalog of refactoring types. Also, he introduced the building blocks for these refactoring types. Furthermore, he defined the concepts of pre-condition and post-condition of refactoring, which are still used in most refactoring tools today to correctly automate this process.

Martin Fowler wrote the first book introducing the refactoring techniques to the developers in the wild [Fow]. According to Fowler, refactoring is “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure”, which is the commonly accepted definition for this term. In Fowler’s book, he gives a comprehensive overview of over one hundred popular refactoring types as well as the code problems each one of them can address. In another book written by Kerievsky, the author adopts a different strategy that centers around design problems and design patterns [Ker04]. By applying refactorings, Kerievsky transforms the poorly-designed code snippets to the elegance ones by using design patterns.

Because refactoring can fix the design issues after a design has been implemented, agile community embraces this activity enthusiastically. In contrast to the conventional software development model such as water fall model, agile processes consider change as an inherent part of software development, thus developers have to prepare for and even welcome changes. Refactoring, as a weapon to deal with stale designs, is integrated to several agile processes such as extreme programming [BA04].
5.2 Refactoring Opportunities

Chapter 2 presents the BeneFactor tool which detects the starting point of a manual refactoring and helps the developer finish the refactoring automatically. Similar to BeneFactor, existing works also help developers identify refactoring opportunities, with or without the further support for finishing them. In this section, I detail two types of tools falling into this category, namely code smell detectors and clone detectors.

In addition to preparing for the emerging software changes, another usage of refactoring is to remove code smells. Code smells are not functional defects but flaws that may increase the maintenance cost in the future. For instance, code duplication is a code smell because duplicated code snippets potentially contain duplicated defects. Another example is god class, which is a oversize class declaration with multiple responsibilities. A god class should be refactored to several classes with less responsibility so that a developer can focus on one feature at a time while ignoring others. Dijkstra’s famous critique against goto statements leads to spaghetti code smell which is an analogy of intertwined logic; the spaghetti code smell is so bad that mainstream programming languages today stopped supporting goto statements [Dij79].

Folwer and colleagues’ book also gives a catalog of code smells and the refactoring types that can address them [Fow]. However, their classification of code smells is based on practitioners’ personal experience, which may vary from developer to developer. To understand code smells better, researchers conducted multiple empirical studies in the real world setting. For instance, to investigate why and how developers copy code, Kim and colleagues observed the copy-paste operations during a developer’s workflow; they found out that over 70% of the C&P operations are performed on text shorter than a line of code, that is, they save typing efforts. Regarding the intentions of the C&P operations, Kim and colleagues found that developers copy and paste code to regroup, relocate and reorganized program elements [Kim04]. Kapser and Godfrey noticed that the research community presumably considers code clone is bad; however they conducted studies showing that code clone is frequently used as a principled engineering tool [KG08]. For instance, developers sometimes clone stable code to a sandbox and incrementally add new features to the sandbox.

Later, Kim and colleagues conducted another study on the conventional wisdom regarding code clones. For long time, researchers assume code clones are inherently bad and need to be refactored out immediately. However, Kim and colleagues found that code clones possibly need no removal for two reasons: (1) some code clones exist only in a short period of time
because they will functionally diverge soon, and (2) some code clones cannot be removed by refactoring due to the language limitations [Kim05]. Based on the observation that some code clones need refactoring while others do not, Cai and Kim conducted a following study in an attempt to build a decision tree to predict a clone’s survival time; they found out that the size of clones and the number of clones do not correlate with the survival time, while the number of developers who have modified the clone and the last time of a clone addition or removal do [CK11]. Similarly, based on the observation that some cloned code does little harm since they evolve independently from each other, Wang and colleagues used Bayesian models to predict the harmfulness of a specific copy/paste operation [Wan12]. The features Wang and colleagues used fall into the three categories including history features, code features and destination features.

Code clones require no more maintenance effort from developers if they are stable. Assuming that all code clones are bad implies that cloned code is less stable than noncloned code. To test this assumption, researchers conducted multiple studies showing the different results. For instance, Mondal and colleagues measured four different stability metrics on both cloned code and noncloned code, finding that (1) cloned code is generally less stable than noncloned code, (2) the stability of cloned code correlates with the language of the codebase, and (3) the development strategy may decide the stability of the cloned code [Mon12]. To the contrary, Krinke conducted another study on three large open source projects to measure the longevity of cloned code and found out that cloned code is older than noncloned code, implying that cloned code is more stable than noncloned code [Kri11].

Another assumption researchers made about code clones is that cloned code will stay identical for a long time which requires developers to perform the consistent changes on each copy. To test this assumption, researchers conducted multiple studies showing the divergent results as well. For instance, Gode and Harder proposed a technique to automatically detect consecutive changes to cloned code and conducted an empirical study on three software systems [GH11]. They found out that (1) consecutive changes do occur, implying that developers sometimes need to update the multiple copies of the cloned code, (2) consecutive changes contain only few unwanted inconsistencies, and (3) consecutive changes are not a precise enough indicator for the unwanted inconsistencies. In contrary, Krinke conducted an empirical study on five open source projects and found that developers consistently update only half of the clone code and the inconsistency among cloned copies last long time, implying that the cloned code gradually become too different to be considered as clone any more [Kri07].
Similarly, Xie and colleagues conducted a code clone study on three long-lived open source projects and analyzed the error-proneness of three different code clone types [Xie13]. They found that the mutation of type-2 and type-3 clones are more error-prone than that of type-1 clones, and that increasing the distance of cloned segments increases the likelihood of introducing defects. Based on the observation that developers frequently clone code to related software components, Rahman and colleagues proposed the use of coupling metrics before the application of clone detectors to reduce the search space of the detectors [Rah13]. Park and colleagues investigated the bug reports of three large open source projects, finding a significant portion of the resolved bugs involves more than one bug fix attempt [Par12]. However, few of these bug refixes are due to fixing cloned code snippets, that is, cloned code is not always maintained consistently.

Beyond the scope of code clone, researchers also conducted multiple studies regarding other code smell types. For instance, to investigate whether code smell will finally lead to buggy software, Hall and colleagues studied five smell types in three open source projects and correlated the smells with the bug reports [Hal14]. Hall and colleagues found that (1) switch statements have no effect on the number of defects at all, (2) message chains and data clumps increase defects in two projects while reduce in one, and (3) middle man and speculative generality respectively reduce defects in one system. Arcoverde and colleagues empirically studied why certain code smells are ignored and they found that the fear of breaking the client code results in that most library developers postpone their refactoring smells out of the codebase [Arc11].

To understand how code smells are introduced into the codebase as well as how they are removed, Olbrich and colleagues studied god class and shotgun surgery in the commit history of two large open source systems [Olb09]. They found that (1) software maintenance includes both smell-increasing and smell-decreasing phases, (2) disregarding the smell types, smell-infected code snippets are updated by developers more often than others, implying that change frequency can serve as an indicator of code smells.

In addition to studying code smells, researchers proposed multiple techniques that automatically detect these design issues to help developers fix them. A large category of these tools is clone detectors. For instance, Kamiya and colleagues proposed a token-based tool called CCFinder [Kam02]. CCFinder detects cloned codes in the following steps: it first reads source files into memory and transforms tokens in the files by adding the informative tokens or removing the trivial ones; CCFinder next replaces all variable and type names by the special
tokens; thirdly, CCFinder performs string matching on the preprocessed token streams to find the common and popular substrings between the source files; finally, CCFinder translates the common token streams back to the original locations in the files and reports the locations as cloned code. Since CCFinder is entirely token-based, developers can easily port the tool to detect clones in other programming languages. To improve the memory efficiency of CCFinder, Basit and colleagues proposed using suffix array instead of suffix tree to match the common strings [Bas07].

Also being token-based, Yuan and Guo presented an accurate and scalable clone detector called Boreas [YG12]. Boreas used different count metrics for keywords, variable names, and symbols; for instance, the count metrics for variable names include if the variable is used, if the variable is in an if-expression, if the variable is in the second-level loop, and so on. After calculating the count matrix for all variables, keywords and operators, Boreas compare the similarity of the matrices of different code snippets; if the similarity achieves certain threshold, the two code snippets are considered as clones.

Another category of clone detectors is based on tree similarity. Differing from the token-based techniques, tree-based detectors take parse trees as input. For instance, Baxter and colleagues proposed a clone detector that compares the similarity between the subtrees in the codebase [Bax98]. The comparison is based on the hash value of each subtree. The hash function ignores the syntax detail of each subtree such as identifier names, thus their technique can also find near clones in addition to the exact ones.

Similarly, Jiang and colleagues developed a tool called DECKARD that detects code clones by finding alike subtrees [Jia07]. Instead of using hash values, Jiang and colleagues used characteristic vectors to describe a subtree. A characteristic vector contains the values for the number of loops, the number of assignment, and so on. After calculating the vector for each leave in the parse tree, DECKARD traverses the tree in post-order to calculate the vectors for each non-leave node in the parse tree. Next, the similarity of subtrees is measured by the Euclidean distance between the corresponding vectors.

Also being tree-based, Lee and Doh proposed using tree-patterns to detect the similar code snippets [LD09]. Unlike the aforementioned techniques, Lee and Doh’s clone detector does not transform subtrees to comparable text, such as hash code or vectors. They consider a subtree as a shape defined by the internal node types. For instance, two subtrees are similar if their first children are both assignments and their second children are both function invocations. If two
subtrees share the common pattern, they further look into the content of the trees and to decide whether they are promising clone candidates.

The third category of clone detectors attempts to find the semantically similar code snippets disregarding of their syntax. As programming languages evolving, a developer can freely choose her favorite syntax to implement the same feature. However, such freedom poses challenges to clone detectors if they only analyze the syntactic similarity. To address this shortcoming, Gabel and colleagues proposed a scalable algorithm that extracts the semantic context of the snippets, that is, the program dependency graphs, and apply the tree-based clone detectors on the graphs [Gab08]. Park and colleagues proposed a program-similarity measurement based on code abstraction [Par13]. For a given codebase, their technique first filters out the tokens that have no semantic implications, such as keywords; next their technique summarizes the abstraction of the codebase by analyzing annotation, variable declaration, format string and so on; finally, they compare the abstractions of two given codebases to detect clones.

Similarly, Wang and colleagues used the software birthmarks to detect code plagiarism [Wan12]. They proposed a technique to encode the call graph of a code snippet as the birthmark of the snippet; using the birthmark, they can detect not only whole software theft but also certain components in software that are stolen. Choi and colleagues proposed a tool called runtime abstract memory context based tool, which can collect the semantic information at runtime [Cho13]. Using the runtime information, their tool can further analyze the memory context of the running software, and compare the contexts to detect software plagiarism without taking source code as input. Also trying to detect the code clones at the binary level, Sabjornsen and colleagues proposed to decompile and normalize the binary code thus the source-code level clone detectors can be applied on the binary representation [Sæb09]. Their normalization step generalizes the specific instructions to the generic ones.

Different from the aforementioned techniques, researchers also proposed techniques to supplement the existing code clone detectors. For instance, Biegel and Diehl used the pipeline model in a set of code detection APIs, allowing developers to parameterize and customize the existing components as well as adding the new ones [BD10]. Their pipeline consists of five parts including parsing, preprocessing, pooling, comparing, and filtering. Noticing that most clone detectors generate a large amount of false positives, Juergens and Gode proposed to use clone coupling as a criterion to filter the detected clones; their study supports their claim by showing an improved accuracy by using the criterion [JG10].
To achieve both accuracy and scalability, Chen and colleagues used a geometry characteristics, called centroid, of the dependency graphs to measure the similarity between code snippets in two given applications [Che14]. Geometrically, centroid is the point at which a cardboard of the region could be perfectly balanced at the tip of a pencil; Chen and colleagues found that comparing the centroid of two methods can fairly decide whether they are clones of each other, saving much computing effort in analyzing the detail of the method pair. Conventional clone detectors compare two code snippets at a time, leaving the summarization of multiple clone instances to the users. To solve this problem, Lin and colleagues proposed a technique that compares multiple clone instances simultaneously; thus the reported clones include not only pairs, but also triples, quadruples, and so on [Lin14b]. To improve users’ understanding of the report of clone detectors, Tairas and colleagues proposed a visualization tool to annotate the cloned snippets using various colors [Tai06].

In addition to code clone detectors, researchers also proposed the detectors for other types of code smell; these tools assist developers to locate the refactoring opportunities by statically analyzing code metrics. For instance, Murphy-Hill and colleagues presented an in-editor tool that visualizes the intensity of code smells; when a code snippet with certain smell appears in the editor, the tool visualizes the intensity of the smell by using various shapes and colors [MHB10]. Later, Murphy-Hill and Black summarized the characteristics of the usable code smell detectors; these characteristics include availability, unobtrusiveness, context-sensitivity, relativity, scalability, relationality, and expressiveness [MHB08c].

### 5.3 Source Differencing Techniques

The common theme of the tools presented in Chapter 2, Chapter 3 and Chapter 4 is the application of refactoring detection techniques. Taking a codebase before and after a change as input, refactoring detection techniques output the performed refactorings. However, the first step to detect refactoring is to refine and represent diffs, for instance, ReviewFactor detects refactorings on mapping trees. This section details the existing source differencing techniques and various representations of diffs.

Measuring codebase changes is useful for multiple purposes, such as version control, code review, and refactoring detection. Source differencing techniques can be categorized as line based and tree based. The line based techniques are widely adopted in version control systems.
due to its straightforwardness and high efficiency; while the tree-based techniques are widely used in software static analysis because of its insightfulness about software structure.

Line based techniques treat source code similarly with other text files. Any change to a source code file is formalized as added and removed lines. For instance, originating from Bell Labs, Hunt and McIlroy proposed the diff utility that is integrated into the Unix operating system [HM76]. The diff utility solves the “longest common subsequence problem” to find the unchanged part across versions. To further improve the efficiency of calculating the shortest edit script, Myers proposed an algorithm based on the observation that finding the shortest path in edit graph is equivalent to finding the longest one [Mye86]. Built on Unix diff, Canfora and colleagues demonstrated Ldiff to show updated lines in addition to added and removed lines [Can09]; as an alternative to line based techniques, Kamiya and colleagues proposed a token-based technique called CCFinder to detect code clones between two versions of software [Kam02].

To visually assist developer understand line based code change, researchers proposed improved user interface of diff. For instance, modern code review tools like Gerrit [Ger] and Mondrian use web-based presentation of line changes; Gomez and colleagues proposed the torch dashboard to assist developers to make release decisions when facing multiple change candidates [Gom10].

Tree based techniques respect the structure of programming languages and translate text edition to tree update. The core concept of tree based techniques is to map abstract syntax tree nodes before and after change, thus the updated, removed or added nodes can be extracted. Comparing to line based techniques, tree based techniques are computationally heavy, however they can facilitate the semantic analysis of code change.

For instance, Horwitz proposed an extension to Unix diff to represent semantically changed components [Hor90]; Apiwattanapong and colleagues presented a technique that compares different versions of source code written in object-oriented languages [Api04]; Xing and Stroulia proposed a differing algorithm based on design graphs [XS05]; Chawathe and colleagues discovered the need of representing changes of structural data and proposed an algorithm to calculate the minimum edit script that transforms one data tree to another [Cha96]; inspired by Chawathe and colleagues’ work, Change Distilling, proposed by Fluri and colleagues, compares the fine-grained abstract syntax tree nodes and generates the minimum edit script that transforms one source tree into another [Flu07]; to better presenting the results of mapped code elements, Kim and colleagues infer high level change rules after mapping [Kim07].
5.4 Refactoring Detection

As a research subfield that is most related to my dissertation work, refactoring detection techniques compare a codebase before and after change to infer refactorings that are performed either manually or automatically. The detected refactorings can serve various purposes in software maintenance, such as generating commit messages, migrating libraries, regressing testing, and assisting source code comprehension. Although several other techniques can also detect refactorings, such as mining the log files of refactoring tools, this section mainly focuses on those techniques that take only source code as input.

The first subcategory of refactoring detection techniques compares a codebase before and after change to infer refactoring. In general, these techniques use two steps: code element mapping and similarity checking. The code element mapping step serves two purposes: filtering out the code elements that are not changed and collecting the change part in fine-grained detail; and the similarity checking step ensures that different parts in a given change form a meaningful refactoring, for instance, two methods containing the similar body however existing in different classes before and after the change form a refactoring of moved method.

As examples of these techniques, Weissgerber and Diehl proposed a refactoring detection technique and conducted a study to evaluate its precision and recall [WD06b]; to help developers understand code change, Gorg and Weissgerber proposed a tool called REFVIS to visualized detected refactorings [GW05]; Dig and colleagues proposed a technique that combines a fast syntactic analysis to collect refactoring candidates and a more expensive semantic analysis to refine them [Dig06]; to improve the extensibility of existing refactoring detection techniques, Prete and colleagues proposed a tool called Ref-Finder that uses logic rules to encode complex refactorings and detect them in two codebases [Pre10; Kim10]; by trying to interpret a source code change through refactorings, Mahouachi and colleagues proposed a search-based approach to detect refactorings in that change [Mah13]; Biegel and colleagues compared the effectiveness of three different similarity checking techniques, which are text-based, AST-based, and token based, in measuring the structural relevance of different code elements when detecting refactoring, and they conclude that the three techniques perform with a comparable quality [Bie11].

In a higher level than these techniques, another subcategory of refactoring detection techniques exploit the API changes to help client codebase co-evolve with the libraries. A typical use case of these tools is that after the libraries being refactored, such as methods being re-
named or functions being moved, the client codebase that uses these libraries can automatically update the stale way of using the libraries to the updated one.

As examples of these techniques, Antoniol and colleagues proposed a technique to trace the classes after they are split, merged, or moved by developers, so that the client codebase can adapt accordingly [Ant04]; Henkel and Diwan presented a lightweight approach called CatchUp! that records how a developer manually evolves APIs and replays the script to evolve other usage of the same APIs automatically [HD05]; Dig and colleagues found out that over 80% of the disruptive changes in libraries are due to refactoring and proposed a technique that detects the API-level refactorings with a precision over 85% [DJ05]; Taneja and colleagues proposed RefacLib that detects refactorings across the different versions of libraries by combining syntactic analysis and various heuristics [Tan07].

The third subcategory of refactoring detection techniques take artifacts other than the source code as input. For instance, Xing and Stroulia proposed a technique that analyzes unified model language to detect refactorings on the design level [XS06a]; Negara and colleagues monitored the fine-grained editing steps performed by developers to identify the frequent code change patterns, many of which are refactorings [Neg14]; Demeyer and colleagues exploited software metrics to identify refactorings to assist reverse engineering [Dem00]; Murphy-Hill and colleagues, in their study of refactoring tool usage, mined the log files of refactoring tools to infer when developers refactor their codebases automatically [MH09b]; they further analyzed the pros and cons of four methods of collecting the refactoring data, which are mining commit logs, analyzing code histories, observing programmers and logging refactoring tool use [MH09b]; Vakilian and colleagues developed a tool called CodingSpectator to collect more informative data of refactoring tool usage than conventional data collectors, leading to more usability problems of refactoring tools exposed [Vak11].

These existing techniques differ from my work in the following aspects: (1) BeneFactor detects the starting step of a manual refactoring instead of completed refactorings; (2) GhostFactor continuously monitors the revisions of the source file under editing and efficiently detects code changes that are structurally similar to refactorings; and (3) to help code reviewers separate the refactoring and non-refactoring part in a given code change, ReviewFactor detects the performed refactorings with a higher precision than that of existing refactoring detection techniques.
5.5 Refactoring Studies

As a change type that frequently happens in software development, researchers studied refactoring intensively to answer questions from various aspects. In this section, I categorize existing refactoring-centered studies by their purposes; more specifically, these categories include refactoring’s impact, refactoring practice, and refactoring support.

Most existing studies show that refactoring, although maintains the behavior of software systems, can improve their nonfunctional attributes. For instance, Du Bois and colleagues investigated whether several types of refactorings can improve the cohesion and reduce the coupling of software, and they also proposed the guidelines when refactoring can deliver such benefits [DB04]; Kataoka and colleagues quantitatively measured the maintainability enhancement by refactoring existing programs, and their analysis methods can also help developers in all stages from planning to executing [Kat02]; focusing on reusability, Moser and colleagues investigated in a close-to industrial, agile environment to analyze the benefit of refactoring [Mos06].

Geppert and colleagues reported a case study where refactoring improves the changeability of a legacy system, leading to less defects filed by customers and less effort from developers to make changes [Gep05]; Silva and colleagues analyzed three different embedded software systems and found that inline method refactoring not only improves their performance but also reduces their energy consumptions [SB10]; Najjar and colleagues empirically investigated the opportunities, benefits and problems of refactoring the constructors of Java classes, and they concluded that refactoring can improve comprehension and reduce duplication [Naj03].

In contrary to the aforementioned works, Stroggylos and Spinellis analyzed the code changes that are marked as refactoring in version control systems and concluded that, according to software metrics, refactoring does not improve the internal structure of software, and in certain cases, even worsens it [SS07]; similarly, Weissgerber and Diehl proposed a technique that relates bug data to refactoring changes extracted from version control systems, and they found out that refactoring activities are sometimes followed by the increasing number of bug reports [WD06a].

Another category of studies investigates how real-world developers perform refactoring, aiming at improving the tool support for such activity. For instance, Counsell and colleagues studied the refactoring activities in seven open source projects and concluded that the most commonly used refactoring types are pull up method, move method, add parameter, move field,
and rename method [Cou06]; similarly, Advani and Hassoun studied what types of refactoring are performed more often than others, and their top five most frequent types are rename field, rename method, move field, move method and add parameter [Adv06]; in addition to studying the frequency of different refactoring types, Advani and Hassoun also found that the massive structure-changing refactorings are less common than the simple ones [Adv06].

To investigate the impact of refactoring on API evolution, Dig and Johnson studied three frameworks and one library, and they found out that the majority of the client-breaking library changes are refactorings [DJ05]; Kim and colleagues studied code clones in source code repositories and found out that code clones are not always refactored because either the clones are transient or the limitation of the programming language prevents the developer from refactoring [Kim05]; in another field study, Kim and colleagues interviewed the professional developers from Microsoft and found out that (1) according to developers, the definition of refactoring does not strictly adhere to “behavior-preserving” code changes, (2) the professional developers consider refactoring involves substantial cost and risks, and (3) refactoring does reduce defects and dependencies [Kim12].

The third category of refactoring studies are centered among refactoring tools. For instance, Xing and Stroulia conducted a case study on Eclipse and found out that many refactoring types which happen frequently in the real world lack the support from refactoring tools in the IDEs [XS06b]. Murphy-Hill and colleagues investigated the datasets spanning more than 13 thousand developers, 240 thousand tool-assisted refactorings, 2.5 thousand developer hours, and 3.4 thousand version control commits. Their study concluded that (1) developers rarely indicate refactoring changes in commit messages, (2) developers frequently interleave refactoring changes with behavior-altering changes, (3) the most frequently applied refactoring type is rename, (4) the low-level refactorings are more frequent than the high-level ones, (5) the majority of refactorings are performed by hand, and (6) certain refactoring types are more likely to be automated than other ones [MH09b].

Murphy-Hill and colleagues’ study also revealed that refactoring tool usage is often batched, that is, developers often invoke refactoring tools continuously with a short period of time [MH09b]. Further, the study found out that toolsmiths, who developed and designed refactoring tools, use these tools no more frequent than average users [MH09b].

Following Murphy-Hill and colleagues’ study, Vakilian and colleagues argue that existing data collectors for refactoring tools are inadequate for answering the question why refactoring tools are underused, and they proposed two novel data collectors called CodingSpectator and
CodingTracker, which can answer the potential research questions they proposed in the paper [Vak11]; Using the novel data collectors, Vakilian and colleagues reported a study of how refactoring tools are used, underused and misused in the real world [Vak12]. Their finding confirmed the results of Murphy-Hill and colleagues’ study that refactoring tools are underused, and developers seldomly configure refactoring tools.

Furthermore, their study found that developers prefer the lightweight method of invoking refactoring tools, and that developers avoid refactoring tools because (1) certain refactoring types are rarely applied, disregarding of the way of the invocation; (2) some developers do not know that IDEs provide refactoring tools; (3) some developers’ vocabulary of refactoring types differs from the one used in IDEs; and (4) developers sometimes distrust refactoring tools due to their low predictability. In addition, Vakilian and colleagues’ study also shows that developers may invoke refactoring tools in an unsafe way. For instance, developers sometimes ignore the warnings of potential risks by performing certain refactorings automatically; and developers occasionally use refactoring tools to perform non-refactoring changes, such as modifying the visibility of a method by invoking change method signature refactoring [Vak12].

In another study using the data collected by CodingSpectator, Vakilian and colleagues revealed some usability problems of conventional refactoring tools [VJ14]. They used the critical incident technique to analyze the events that stop developers from invoking refactoring tools correctly; and they summarized several alternative refactoring paths, which differ from the primary paths. The usability problems they identified include vague messages, overly strong pre-conditions, name conflicts, unintuitive configuration options, and invalid code selections [Vak11].

### 5.6 Refactoring Tools

Due to the frequency and usefulness of refactoring, researchers and developers implemented many refactoring tools to automate this activity. In this section, I summarize the existing tools from different aspects, including tools for various programming languages, tools with the enhanced usability, and tools with the novel refactoring types. These tools are similar to BeneFactor in Chapter 2 in terms of purposes; however, none of these refactoring tools can solve the late awareness problem. I will omit those refactoring tools available in mainstream IDEs such as Eclipse [Ecl], Visual Studio [Vs], NetBeans [Net], IntelliJ [Int] and XCode [Xco].
Due to the syntactic and semantic difference among various programming languages, refactoring tools need to be tailored accordingly. Robert and colleagues implemented the first-known refactoring tool that integrates multiple refactoring types for a language called Smalltalk, and conducted several case studies to investigate its usefulness [Rob97]; also on Smalltalk, Unterholzner analyzed the problems caused by the dynamic typing system of Smalltalk and proposed a static analysis technique to infer types to assist refactoring tools on Smalltalk [Unt12].

Similar with Smalltalk, JavaScript is a dynamic typed language on which the implementation of refactoring tools faces several difficulties. To address these difficulties, Feldthaus and colleagues proposed a technique based on static pointer analysis to specify and implement JavaScript refactorings [Fel11]. The key insight of their technique is that the correctness of JavaScript refactorings can be validated by querying the results of pointer analysis. However, the pointer analysis framework does not scale well to a large codebase under refactoring. To improve the scalability, Feldthaus and colleagues later proposed a semi-automatic way of renaming JavaScript identifiers; this technique relates identifiers by examining programming elements statically and asks developers’ explicit answer regarding whether a potentially related name should be updated [FM13]. To summarize existing refactoring tools for dynamic languages, Schafer argued that dynamic languages need refactoring tools more than static languages do, however checking behavior-preserving conditions in dynamic languages is more challenging due to the implicit type bindings and type hierarchy [Sch12].

In addition to typing systems, some other choices in designing a programming language may also pose difficulties on the implementation of refactoring tools for that language. For instance, reflection allows developers to modify the structure and the behavior of the program at runtime. Many major object-oriented programming languages like Java and C# support this feature. However, using reflection may prevent refactoring tools from working correctly. For instance, to invoke a method in the reflective fashion, a developer puts the method name as a string literal in her Java source code. After renaming the method by invoking refactoring tools, the string literal does not update accordingly because refactoring tools have no idea that the string literal binds to the original method name. To solve this problem, Thies and Bodden proposed a reflection-aware refactoring tool for Java called REFAFLEX [TB12]. REFAFLEX first runs dynamic analysis to log the reflective calls during test runs and next uses the collected logs to prevent developers from renaming a method that may be invoked by reflection.

Another limitation of conventional refactoring tools is dealing with polyglot projects. For instance, many projects written in Java have XML files integrated serving configuration pur-
posed; the invocation of refactoring tools on the Java source code does not update the code in XML source, even though they are inherently related. To break this limitation, Chen and Johnson presented a tool to rename automatically beyond the boundary of programming languages [CJ08].

Borrowing the refactoring browser idea from Robert and colleagues, Spinellis proposed a tool called CScout that assists C programmers to identify refactoring opportunities and to apply them [Spi10]. Applying the concept of refactoring to logic programming, Serebrenik and colleagues developed a catalog of prolog refactorings as well as a tool called ViPRes that semi-automates the refactorings, showing that refactorings on logic programming languages are not only possibly but also desirable [Ser08]. Functional programs are inherently hard to refactor. Li and colleagues implemented a refactoring tool for Haskell called HaRe and a refactoring tool for Erlang called Wrangler [LT08; LT12]. To improve the usability of Wrangler, Li and colleagues further proposed a domain-specific language that allows developers to encode composite refactorings, to test them, and to apply them automatically [LT12].

End user programming continuously gets attention from academia. For instance, the end users who write spreadsheet programs significantly outnumber the trained developers who write professional software. Similar with professional software, spreadsheet programs also have structures and their structures are under continuous improvement. Focus on this area, Badame and Dig studied a large number of corpus of spreadsheets and categorized seven refactoring types on spreadsheet formula [BD12]. In addition, they implemented a plugin for Microsoft Excel to automate these refactorings.

The second category of refactoring tools aim at increasing the usability of conventional refactoring tools. Although almost all mainstream IDEs integrate refactoring tools as an necessary part, these tools are not used as often as they should be, as shown by Murphy-Hill and colleagues’ study on how developers in wild actually refactor their codebase [MH09b]. To diagnose the usability issues of conventional refactoring tools, Murphy-Hill distilled the model of refactoring tool usage which includes the steps of refactoring identification, code selection, tool configuration, refactoring execution and so on [MH09a]. Many of these steps can be merged, paralleled or reordered to improve the usability of refactoring tools.

To improve the communication between refactoring tools and developers, Murphy-Hill and Black proposed three new refactoring tools, which are (1) a selection assist that helps developers select a meaningful group of statements before their being extracted to a new method; (2) a box view that shows nested groups of statements to ensure that the start and the end point of
the developers’ selection are at the same level, and (3) refactoring annotations that show the
data flow into or flow out of the selected statements to keep developers aware of the parameters
and the return values of the extracted method [MHB08b].

In addition to the communication problem, other usability issues also haunt the conven-
tional refactoring tools, and researchers actively propose solutions to these problems. For in-
dstance, Foster and colleagues found out that using conventional refactoring tools always re-
quires developers to switch mental and physical contexts, and such cognitive pre-condition
may prevent developers from refactoring automatically; thus they proposed a novel refactoring
tool called WitchDoctor that can detect manual refactoring on the fly, finish the refactoring
automatically in the background, and propose the completion long before the developers man-
ually finish [Fos12].

Another usability problem of conventional refactoring tools is that they only support the
limited number of refactoring types. However, developers may intend to perform other refactor-
ing types that are not commonly defined or slightly different from the ones that are commonly
defined. Either case will lower the usefulness of conventional refactoring tools. To solve this
problem, Raychev and colleagues proposed the concept of refactoring with synthesis that al-
 lows a developer to conduct the initial steps of the intended refactoring and ask the tool to com-
plete it. The synthesis system uses elementary refactoring types as building blocks to match the
developer’s manual steps and applies these elementary refactorings sequentially [Ray13]. Sim-
ilarly, Lee and colleagues proposed a novel refactoring tool called DNDRefactoring which can
figure out the intended refactoring by the developer’s drag and drop gestures [Lee13]. Without
requiring the developer to remember the names of her intended refactorings, DNDRefactoring
further lowers the cognitive burden of invoking refactoring tools.

The third category of novel refactoring tools aim to support previously unidentified refactor-
ing types. These refactoring types do not adhere tightly to the classical definition of refactoring,
which is behavior-preserving code change that improves code structure. These new refactor-
ing types improve certain attributes of software at runtime without necessarily improving the
structure of the software.

For instance, certain code snippets are not initially thread-safe. Conventionally, developers
have to manually retrofit the snippets to be thread-safe by adding locks to the shared data. Dig
and colleagues introduced a tool called CONCURRENCER that uses Java built-in libraries to
automatically identify and refactor sequential code by using either thread-safe data structures or
critical segments [Dig09]. Okur and colleagues studied how C# developers use asynchronous
constructs and showed that developers frequently misuse them; thus, they developed a refactoring tool called ASYNCIFIER that automatically transforms callback-based asynchronous to using async/await keywords in a correct way [Oku14]. Similarly, Lin and colleagues noticed that heavy computation in the UI thread of Android application is largely due to the underuse and misuse of the asynchronous keyword; therefore, they studied how developers use the keyword in real world and developed a refactoring tool to extract heavy operations to asynchronous execution [Lin14a].

In object-oriented programming, classes can be either mutable or immutable; the handling the immutable classes does not introduce any side effects, thus developers sometimes need to convert a mutable class to an immutable one to get this benefit. To meet this need, Kjolstad and colleagues presented a tool called IMMUTATOR that safely automates this transformation [Kjo11]. Another desirable attribute of a program is being reentrant, meaning running the program on distinct inputs does not affect each other. To refactor a non-reentrant program to a reentrant one, Wloka and colleagues proposed a refactoring tool called Reentrancer for Java that automatically replaces global states with thread-local states, and performs each execution in a fresh thread accessing only the thread-local states [Wlo09].

In addition to improving thread-safety, researchers also proposed new tools to improve the efficiency of software, especially for software running on mobile devices. For instance, Hilton and colleagues developed a refactoring tool called CLOUDIFIER that automatically refactors local data types into cloud data types on the touchdevelop platform, which is a programming and runtime environment on Windows phones [Hil14]. To improve the energy-efficiency of Android applications, Zhang and colleagues developed a tool called DPartner that automatically identifies code snippets that worth offloading, implements data structures that are suitable for offloading, and generates code snippets that are deployable to both local devices and servers [Zha12]. Also aiming at improving performance and energy-efficiency, Pinto’s thesis proposed a catalog of refactoring types that are specific to multicore applications [Pin13]. Brown and colleagues proposed a language-independent refactoring approach that helps developers parallelize their software under development by using high-level design patterns [Bro12].

Security is also an important concern for most software systems. Careless developers may commit code changes that expose security loopholes exploitable to attackers. Maruyama and Omori noticed that not only developers, but also refactoring tools may introduce security loopholes, therefore they presented a security-aware refactoring tools warning developers the potential risks before an automated refactoring has been applied [MO11]. Smith and Thober
showed that refactoring security-critical part in software to an independent component can facilitate an end-to-end security policy which is not feasible before refactoring [ST06]. Similar to Smith and Thober’s work, Alshammari and colleagues conducted a study showing that refactoring improves the security level of existing programs from the perspective of information flow [Als10].

Programming language evolves. New features of existing programming language may introduce new refactoring opportunities. For instance, Java 8 introduced lambda expressions which allow developers to simplify anonymous classes and unnecessary class wrappers. However, to manually convert codebase written in previous Java versions to accommodate lambda expressions is an error-prone and tedious process. To solve this problem, Gyori and colleagues presented a refactoring tool called LAMBDAFICATOR to automate two refactorings, which are converting anonymous classes to lambda expressions and converting for loops over a collection to using lambda expressions [Gyo13].

5.7 Refactoring Conditions

My work in this dissertation adopts the refactoring condition checking mechanism to ensure refactoring correctness. For instance, GhostFactor mentioned in Chapter 3 detects a manually finished refactoring and checks conditions retrospectively to find defects. ReviewFactor in Chapter 4 checks the refactoring conditions on the detected refactoring candidates for further refinement. More than checking refactoring conditions, researchers and toolsmiths proposed multiple methods to ensure refactoring correctness, each of which has their advantages and disadvantages. In this section, I briefly summarized those methods. More specifically, these methods fall into three categories including formalizing refactoring changes, pre-condition and post-condition checking, and micro-refactoring composite.

Formally checking the functional equivalence of two programs is heavy however reliable way of ensuring refactoring correctness. For instance, Cornelio and colleagues proposed using refinement to implement refactoring in object-oriented programs [MCS02]. They proved the correctness of a set of transforming rules using algebraic laws and applied these laws to finish refactoring that is guaranteed to be behavior-preserving. Estler and colleagues applied model checker to ensure that the codebases before and after refactoring are functionally equivalent [Est07]. Later, Estler and Wehrheim applied the Alloy Analyzer, which is popular constraint solver, to check whether a refactoring meets a formally specified requirement; and they
further summarized the category of refactoring types on which using constraint solvers is reasonable [EW08]. For refactoring types such as pull up method and extract super class, type constraints are crucial due to the polymorphisms at software’s runtime. To ensure the correctness of these refactoring types, Tip and colleagues formalized the rules of inferring constraints by statically analyzing codebase and used these rules to detect the condition violations of a proposed refactoring [Tip03]. Differing from these techniques, Soetens conducted a reverse-engineering project that formalizes the implementation of refactoring tools in Eclipse [Soe09].

The refactoring tools implemented in mainstream IDEs, such as Eclipse, adopt a heuristic-based method to ensure refactoring’s correctness [Ecl]. The method includes three steps to finish a refactoring, which are pre-condition checking, program transformation, and post-condition checking. The framework firstly appears in Opdyke’s dissertation and remains popular due to its simplicity and high extensibility [Opd92]. Pre-conditions are a set of analysis on either the code snippets under refactoring or the input given the developer who is invoking the refactoring tool. For instance, supposing a developer is renaming a method name, some pre-conditions can be (1) the given new name does not clash with another method name in the same scope, or (2) the method under renaming is not an overriding of methods in the super classes. Any violations of these pre-conditions can stop the refactoring tools from proceeding to the program transformation step.

After passing all pre-conditions, refactoring tools calculate the necessary code changes to implement the refactoring. Again for rename method refactoring, the code change can be updating all references and declarations referring the name. Usually, the program transformation is not applied directly on the source code files. Rather, these code changes are performed on an in-memory copy of the codebase; therefore the refactoring tools can proceed to the post-condition checking step on such in-memory copy.

The post-conditions are a set of analysis on the codebase after refactoring to ensure that the updated codebase is functionally equivalent to the codebase before refactoring. The difference between pre-condition and post-condition is often blurred. However, generally speaking, post-conditions involve sophisticated data or control flow analysis which is difficult to perform before applying the refactoring. As an example for post-condition, supposing a developer invokes refactoring tools to extract a method automatically, one post-condition for the refactoring is to check that the introduction of the new method does not alter the original data flow in the code base before extracting. If the codebase after change passes all post-conditions, the refac-
toring tools can commit such change to the underling source code files; otherwise they discard the change and report to the developer that the refactoring cannot be applied.

Although widely adopted, the pre-condition and post-condition method of ensuring refactoring’s correctness has their shortcomings. For instance, due to the heuristic nature of summarizing conditions, developers sometimes miss corner cases to check, potentially introducing defects to the codebase under refactoring. Another shortcoming is poor forward compatibility. With new features of programming languages introduced, toolsmiths have to add new condition to check or remove stale ones. To fundamentally facilitate the implementation of refactoring tools, toolsmiths have long advocated a refactoring-tool-friendly compiler or intermediate representations.

For instance, Schafer and colleagues proposed a new way to automate rename refactoring for Java [Sch08]. Differing from the condition-checking paradigm, they enriched the symbol table by allowing plugins to query all references to a given declaration, and each reference is represented by its location in the codebase instead of names. Given this symbol table, any correctly-performed rename refactoring preserves the content of the table. Therefore, the renaming tool automatically refactor the codebase by adhering to two rules: (1) the only code change is updating identifiers, and (2) no symbol tables should have updated content. The novel renaming paradigm avoids checking heuristically summarized conditions and works well for most programming languages.

Similarly, Schafer and colleagues also presented later how language extensions can facilitate the implementation of extract method tools [Sch09]. They break an extract method refactoring to several micro-refactorings including extract block, introducing anonymous method, close over variables, eliminate reference parameter, and lift method. Each of these micro-refactorings is associated with a small set of conditions to check and also easy to implement. Due to the lack of support for anonymous method in Java 5, Schafer and colleagues extended Java to make the introducing anonymous method micro-refactoring possible. If the refactoring tool can successfully finish all those five micro-refactorings, an extract method refactoring can be successfully finished as well. Otherwise, if any one of these micro-refactoring cannot be applied, the extract method refactoring also fails. By introducing micro-refactorings, toolsmiths can tackle each one individually with trivial effort and a micro-refactoring can be reused in multiple different refactoring types. Not only being useful for implementing refactoring tools, Schafer and de Moor also found that micro-refactorings can also serve as specifications for many refactoring types in Eclipse [SM10].
Based on Schafer’s work, Overbey and Johnson proposed differential pre-condition checking that combines the advantage of the paradigms of micro-refactorings and pre-condition/post-condition [OJ11]. Their technique builds the semantic model of the codebase before refactoring, simulates the proposed refactoring change, and builds the semantic model of the codebase after refactoring. Next, taking the two semantic models as input, their technique looks for the differences between them and checks a set of reusable and language-independent conditions on them. For instance, to check the behavior preservation, their technique extracts the affected parts of program graphs and encode rules on the edges and nodes in the affected parts. Although Overbey and Johnson’s work provides a method that is both extensible and language-independent, adding the step of transforming the codebase to a program graph potentially makes refactoring tools slower to invoke.

5.8 Code Review

Code review is an activity performed among peer software developers to examine and critique each others’ code changes. Existing studies show that code review is effective in improving developers’ code quality, reducing defects and enhancing the security of software projects. Not only improving code quality, code review activity is also shown as effective in helping developers comprehend the codebase, transfer knowledge among developers and warm up newcomers to catch up with the rest of the development team. I proposed and implemented ReviewFactor in Chapter 4 to facilitate the code review practice by separating the refactoring and non-refactoring part in a given change. However, most existing work on code review focuses on the human factors. In this section, I survey the exiting work.

To scientifically understand the benefits of performing code reviews, researchers conducted multiple studies in the real world setting. For instance, Kemerer and colleagues studied programs from a personal software process approach and found out that the review rate is a significant factor affecting defect removal effectiveness, even outperforming the individual developer’s capability and other important factors [KP09]. Also, Kemerer and colleagues found that 200 lines of code per hour or less is an effective rate of reviews. Bacchelli and Bird empirically explored the motivations, challenges and outcomes of code review by conducting interview and classifying review comments [BB13]. They found that in addition to defect removal, code review provides other benefits including transferring knowledge, increasing team awareness, creating alternative solutions, improving code quality and sharing code ownership. Regarding
challenges, Bacchelli and Bird found that failing to thoroughly understand code sometimes prevents code reviewers from achieving the desirable outcomes of conducting code review. McIntosh and colleagues studied the correlation between code review and software quality on three projects, and found out (1) the code review coverage reversely correlates with the numbers of post-release defects, (2) the number of reviewers and discussions reversely correlates with the number of post-release defects [McI14].

Specific to open source projects, Beller and colleagues investigated the types of problems fixed by code review [Bel14]. They found that most problems detected by code review is evolvability problems, the addressing of which leads to more maintainable software. More specifically, the evolvability problems occur in documentation, code structure, and visual representation. Following evolvability problems are functional problems; the major part of functional problems are logic flaws. Bosu and Carver studied how open source developers review code by using a code review tool called ReviewBoard, and found out that most committed changes will be reviewed, that review responsibility is centered to several star contributers, and top contributers are also top reviewers [BC12]. Rigby and Bird studied the code review data for several different projects either proprietary or open-source to find the common attributes of this practice [RB13]. They found that code review activity commonly contains the following attributes: (1) code review is lightweight; (2) review happens often and quickly, (3) change size under review is small, (4) two reviewers are the optimum in terms of defect removal, (5) review is also a group problem solving activity.

Baysal and colleagues investigated how non-technical factors influence code reviewers’ responsiveness to a certain patch [Bay13]. The significant factors include the patch size, the priority of the patch, the component under change, the organization of the contributor, the review queue length, the reviewers’ recent activity, and the patch contributor’s experience. Also investigating social factors, Da Cunha and Greatehead conducted a study showing that dimensions of personality, such as introversion-extroversion, sensing-intuition, and thinking-feeling, are fair indicators of developers’ code review competency [DCG07]. To allow researchers to study how developers review code, Mukadam and colleagues extracted the code review data of the Android open source project and developed a client-friendly repository for other researchers to retrieve such data [Muk13].

To measure individual code reviewer’s effectiveness, Uwano and colleagues proposed a method of using eye trackers [Uwa06]. They measure the eye movements of a code reviewer while she is reviewing code changes and filter those lines of code under examining by using
fixation data. Uwano and colleagues further conducted a study by using eye trackers on code reviewers and found out an effective eye move pattern called scan, which reads the entire code first and concentrates on some particular portions next. Failing to follow this pattern leads to less defects being discovered by reviewing that code.

To assist developers to review code, developers and researchers proposed novel techniques. For instance, Gerrit is a popular web-based code review platform that is integrated into the Git version control system [Ger]. Google’s Mondrion allows projects using different source control systems to be reviewed in the identical way. Thongtanunam and colleagues proposed a recommendation technique that suggests code reviewers for a given change based on developers’ work history. Their basic assumption is that code changes should be reviewed by the developers who have contributed to the files of the similar paths with the file paths under review [Tho14]. Muller and colleagues implemented a prototypical review tool using multi-touch interfaces to facilitate collaboration among team members [MÎ2].

Given the codebase before and after change, Thangthumachit and colleagues integrated the refactoring detection technique to code review tools to generate an intermediate version of codebase, which contains only the refactorings performed by the developer; by reviewing the intermediate version and the version after change, the code reviewer can efficiently evaluate the functional impact of the given diff [Tha11]. Similarly, Hayashi and colleagues presented a source-diff tool called REdiffs that allows the code reviewer to focus on the non-refactoring part of a given code change [Hay13]. Although REdiffs and ReviewFactor are conceptually similar, Hayashi and colleagues did not addressed how to ensure the correctness of detected refactorings and failed to evaluate the impact of the refactoring-awareness in real-world code review examples.

Refactoring detection techniques vary. Some have high recall but low precision (like BeneFactor and GhostFactor); some have high precision but low recall (like ReviewFactor). Certainly, not every refactoring detection technique can be used in the code review context. Although Hayashi and colleagues’ work resembles mine for the user interface and the motivation, their work has not addressed the issue of detecting refactorings precisely. Their work assumed that refactoring detection techniques can work without any false positive/negatives, which is certainly not the case. In contrast, my work of ReviewFactor aims to find out an refactoring detection algorithm that detects refactorings with a higher precision than other techniques. Ideally, the precision shall be 100%, implying no false positives. Therefore, Phase II only contains
behavior-preserving change. To achieve the ideal precision, I combined the refactoring detection technique with the refactoring condition checking.

Code review helps developers understand codebase better and transfer project-specific knowledge among them. Based on this pedagogical benefit, researchers used code review framework to teach students programming skills. For instance, Hundhausen and colleagues proposed a group review session allowing students to compare their assignments against a list of best coding practice [Hun13]. They found that code review does not only improve the quality of students’ assignments, but also positively impacts their communication skills and sense of community. Due to the time-intensiveness of in face code review session, Hundhausen later developed an online environment that enables students to review code remotely [Hun10; Hun11].

By comparing the online version with the in face version of the pedagogical code review session, Hundhausen discovered that face to face code review is much more effective in terms of self-efficacy and learning experience. Similarly, applied in operating system class, Dall and Nieh presented a code review system called GradeBoard that simplifies grading for instructors and enables students to understand and learn from their errors [DN14]. Zeller reported the use of Praktomat system allowing students read, review and assess each other’s program assignments; after being reviewed by peer students, a student can revise her submission and resubmit it afterwards [Zel00].
Chapter 6

Conclusion

Software decays. With requirement changing, software developers need to maintain the system to ensure its continuous usefulness. Maintenance can be either function-altering or function-preserving. The function-altering changes, such as fixing defects or adding features, enhance the software in a way that the end users or clients can perceive. On the other hand, the function-preserving changes, such as refactoring, enhance the internal health of software systems without directly benefiting the end users or clients. However, existing studies show that refactoring can effectively enhance the maintainability, testability, and extensibility of software systems. More than being useful, refactoring is also a frequent activity performed by developers, and integrated into agile development processes like Extreme Programming.

To support refactoring, various types of tools are available in both academia and industry. Refactoring tools, for instance, automate the application of the refactoring changes on developers’ codebase. By applying semantic analysis on the internal representation of programming languages, refactoring tools ensure the correctness of refactorings before applying them. As another example, version control systems like git support rename detection so that the files with the similar content however the different names can be tracked as the same file.

In this dissertation, I claim that refactoring detection is an essential technique in the tool support for refactoring. To illustrate this point, I present three use cases of the refactoring detection technique, namely BeneFactor, GhostFactor and ReviewFactor. By implementing these tools and conducting studies on them, I demonstrated that the refactoring detection technique can improve the usability of refactoring tools, empower IDE error messages to detect more defects, and facilitate the code review process. In summary, this dissertation makes the following contributions:
A human study that unveiled the possible reasons for refactoring tools’ underuse problem. Refactoring tools are significantly underused; manually refactoring the codebase potentially introduces more defects. Existing studies found several reasons for this underuse problem, including (1) the user interface is tedious, (2) the condition checking is overly restrictive, and (3) the refactoring engine is inherently buggy. To delve into the cognitive reasons for not using refactoring tools, I designed and conducted a study participated by 12 developers, asking them to manually refactor an open-source codebase. The findings include: (1) I confirmed that manual refactoring introduces defects very often; (2) I found that the late awareness problem, that is, a developer starts manual refactoring without knowing so, sometimes leads to refactoring tools’ underuse; (3) I distilled a set of refactoring workflow patterns, which are the edit steps adopted by a developer to manually refactor; and (4) I found that manual refactoring relies on compiler errors heavily to ensure correctness.

An approach and a tool called BeneFactor that potentially mitigates the late awareness problem of the users of refactoring tools. Based on the refactoring workflow patterns distilled from our manual refactoring study, I designed and implemented a flexible refactoring tool called BeneFactor, in an attempt to solve the late awareness problem. BeneFactor monitors the changes of the codebase under development, compares the changes with the collected refactoring workflow patterns. After detecting the start of a manual refactoring, BeneFactor adds an icon close to the editor of the IDE. The icon reminds the developer that refactoring support is available. By clicking the icon, BeneFactor internally invokes the Eclipse refactoring engine to automatically finish the manually started refactoring.

An approach and a tool called GhostFactor that delegates the refactoring concern to IDE error messages, showing that developers may benefit from refactoring tools without invoking refactoring tools. As another reason for the underuse problem of refactoring tools, developers distrust the changes that applied automatically by refactoring tools, especially when the changes scale big. To solve this problem, I take advantage of that developers frequently rely on compiler errors to guide a manual refactoring. Built on Microsoft Roslyn, I implemented GhostFactor that detects a manually finished refactoring, checks the conditions retrospectively, issues a refactoring error after finding a violation. Although GhostFactor does not automate the refactoring changes, it ensures the correctness.
of manual refactorings, allowing developers to benefit from refactoring tools without using refactoring tools.

- A controlled human study suggesting that GhostFactor could significantly improve the correctness of manual refactorings. To evaluate whether GhostFactor can help developers refactor more correctly and efficiently, I conducted a human study that 8 developers participated in. For each developer, I gave them 6 refactoring tasks to finish, either with or without GhostFactor installed. Our evaluation results suggest that GhostFactor can effectively improve the correctness of manual refactorings; however, the efficiency of manual refactorings stays the same. I also observed that some participants overly rely on GhostFactor while others become increasingly alert at manual refactorings after using GhostFactor for a while.

- A survey study investigating refactoring’s role during code review, suggesting that separating the refactoring and non-refactoring part could accelerate code review. Refactoring improves the internal structure of the codebase without altering the external behavior. With the main goals of detecting defects and transferring knowledge, code reviewers care more about the functional impact of a given change. To test the hypothesis, I conducted a survey study participated by 35 developers who review code frequently. Our study found out that refactoring changes are sometimes distracting to the code reviewers; code reviewers prefer to review the functional impact first; and code reviewers believe that automatically identifying refactorings in a change can help them evaluate that change. In addition, our study suggests that code reviewers evaluate refactorings because: (1) refactorings conveys the design decisions and the conventions, (2) the motivation for refactoring needs to be justified, (3) refactoring exposes the previously hidden problems, (4) refactoring indicates that a single change has a mixture of multiple purposes, and (5) refactoring warns code reviewers about the quality of the previous code review sessions.

- An approach and a tool called ReviewFactor that allows code reviewers to review the refactoring and non-refactoring part separately. Based on the survey study, I found that detecting refactorings in a given change can help code reviewers evaluate the change more effectively and efficiently. To achieve this, I designed and implemented a refactoring-aware code review tool that detects refactorings by reading the log files associated with the refactoring tools and applying the refactoring detection technique.
After identifying refactorings, ReviewFactor allows code reviewers to separately review the refactoring and non-refactoring part in a given change.

- Two studies on ReviewFactor suggesting that the refactoring-awareness has a reasonable impact on code review and the precision of ReviewFactor in detecting refactorings is comparatively high. The first study shows the potential benefit of the refactoring-aware code review tool. I applied ReviewFactor on two open source projects, namely JUnit and FBReader, to detect the lines of code change that can be marked as refactoring. Our study shows that 39% of commits under review can benefit from ReviewFactor; 4.6% of lines of code change in these projects can be marked as refactoring. Assuming code reviewers spend the same amount of time on each line of code, ReviewFactor can save 4.6% of their time spent on code review. The second study evaluates the precision and recall of ReviewFactor. I applied ReviewFactor on one hundred commits of JUnit, manually validated the detected refactorings, and manually examined the commits again to find the refactorings missed by ReviewFactor. Our study suggests that the current implementation of ReviewFactor is of a recall of 94.2% and a precision of 92.5%. Furthermore, I summarized the causes of the false positives and false negatives of ReviewFactor to help the tool’s future improvement.
Bibliography


<table>
<thead>
<tr>
<th>Ref</th>
<th>Title</th>
<th>URL</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tip03</td>
<td>Tip, F. et al. “Refactoring for Generalization Using Type Constraints”. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Pro-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

158


