LESS IS SOMETIMES MORE IN THE AUTOMATION OF SOFTWARE EVOLUTION TASKS

BY

MOHSEN VAKILIAN

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph E. Johnson, Chair
Professor Vikram S. Adve
Associate Professor Brian P. Bailey
Professor Michael D. Ernst, University of Washington
Abstract

Software rapidly evolves. A refactoring is a code change that preserves the behavior of the program. There has been much interest in automation to make refactoring more efficient and reliable. Although modern Integrated Development Environments (IDEs) provide many automated refactorings, studies suggest that programmers underuse automated refactorings. Based on our studies of programmers’ refactoring practices, we argue that usability problems are the common reasons of underusing automated refactorings. We introduce compositional refactoring, a new paradigm of automating refactorings. In this paradigm, the tool designer decomposes the large refactorings into a set of smaller, primitive changes and automates the primitive changes. Then, programmers compose the primitive changes to make larger changes. We have used the compositional paradigm to automate two classes of refactorings: the refactorings supported by modern IDEs and type qualifier inference. Type qualifiers augment a type system to check more properties of the software. Automated inference of type qualifiers can reduce the cost of using type qualifiers. The compositional paradigm enabled us to build the first universal type qualifier inference system. The system takes an existing type qualifier checker as an input and uses it to assist programmers in inserting type qualifiers. Our studies show that compositional refactoring is natural to programmers, gives programmers more control, and makes the automation more predictable and usable than the existing wizard-based and batch paradigms. The compositional paradigm achieves higher usability by automating less. Although this phenomenon may seem counterintuitive, it is not uncommon in automation design. The promising results of the compositional paradigm suggest that other software development automation technologies may also achieve a wider adoption by reducing the level of automation.
Acknowledgments

First and foremost, I praise God. He gave me many gifts including health, a great family, excellent education, unique mentors, colleagues, and friends.

I was privileged to be advised by Ralph Johnson for several years. He has been a great source of support and wisdom. I felt stronger, more knowledgeable, and mature after every meeting with him. Ralph is best known for pioneering design patterns and refactoring. I always wondered how Ralph managed to make such lasting contributions to the field. Ralph has a passion for doing high impact research. He is good at spotting good ideas. Ralph reads a lot. He is good at communicating ideas by telling stories. More importantly, Ralph does not let the rules of academia distract him from pursuing high impact research. I have to admit that working with an influential character such as Ralph is not easy, mainly because he does not push for the usual, short-term measures of success. Rather, he is a great role model for becoming an independent thinker with a passion for changing the world.

During my graduate program, I collaborated with Vikram Adve and members of his research group including Robert Bocchino, Stephen Heumann, and Alex Tzannes. This collaboration helped me gain substantial research experience as part of a research team. Vikram has been always a great source of support and advice for me. He is passionate and ambitious about research, gives students autonomy, and treats them with dignity.

Darko Marinov is like a friend of mine. I have known him since I joined Illinois, and I got to meet him more frequently during my last year at Illinois when his students and I were seated in the same room. His sense of humor makes him easy to approach and fun to work with. He has been often the first person who I consulted for pragmatic advice. Darko and his students, including Milos Gligoric, Qingzhou Luo, Steve Lauterburg, Vilas Jagannath, Farah Hariri, Alex Gyori, Owolabi Legunsen, Lamyaa Eloussi, August Shi, and Yu Lin, helped me in improving my presentations and I enjoyed working with them.
Brian Bailey helped me with the HCI aspects of my thesis. He referred me to the HCI literature related to my work, which improved the presentation of my work significantly. Brian’s deep and insightful comments on my research and presentations had a great impact on the quality of my work.

Mike Ernst played an important role in my type qualifier inference project. I was impressed by his discipline and genuine effort to help. Mike met with me whenever I called for a meeting even if he was traveling. He contributed to the design of the inference tool and the user study. My inference tool depends on some of the infrastructure developed by Mike’s group. Mike led his group to make the changes to the infrastructure that I needed for developing the inference tool. I thank Werner Dietl and Jonathan Burke for making these changes happen.

I collaborated with Danny Dig in several projects. His dedication to research and attention to the clarity of presentations is remarkable. Cosmin Radoi, Fredrik Kjolstad, Caius Brindescu, Mihai Codoban, Michael Hilton, and Semih Okur were some of Danny’s students who gave me suggestions to improve the quality of my written and oral presentations.

Although I first met Tao Xie only during my last year at Illinois, he gave me excellent feedback when discussing my work at the software engineering seminar. Xusheng Xiao, Blake Basset, Wei Yang are some of Tao’s students who were in the same lab that I was. They gave me feedback on my presentations and made the lab a fun place to work at.

I would like to thank several faculty at Illinois for their special support. Mehdi Harandi encouraged me to come to Illinois. Rob Rutenbar, despite being busy as a department head, met with me when I approached him to seek career advice. Alex Kirlik has an extraordinary ability in delivering the right feedback the right way. He gave me excellent advice on my research and career whenever I asked.

Nick Chen, Stas Negara, and Samira Tasharofi were close collaborators of mine for several years. Nick and Stas contributed significantly to the development of CODINGSPECTATOR and conducting the field studies. Stas, Samira, and I collaborated on a static analysis project. I meet with them frequently in the lab and enjoyed fun, deep discussions with them during our collaborations.

Munawar Hafiz, Maurice Rabb, John Brant, Yun Young Lee, Binh Le, and Jeff Overbey were some of Ralph’s other students who helped me in many ways. Munawar shared his experiences with me like a brother. Maurice was a charming person that made the lab a happier place. John continued to visit us and give us feedback long after he left the department. It was nice to see one of the people who
made the first refactoring tool around. Yun Young continued to help me by giving me feedback on my presentations and prototype tools during my last year at Illinois when the rest of Ralph's students had left.

Amarin Phaosawasdi started to work with me on type qualifier inference shortly after he joined Illinois. He quickly started to contribute significantly to the development of CASCADE. His motivation made it possible to release CASCADE on time for the user study.

Four excellent undergraduate students worked with me on my research projects. Kyle Doren and Yalan Meng created the mockup prototypes of the compositional refactorings that we used in a survey study. Devesh Jhunjhunwala and Jiwoong Youn evaluated three existing batch type qualifier inference tools.

I thank all the graduate students, including Peter Dinges, Hamidreza Chitsaz, Barış Aktemur, Rajesh Karmani, Omid Fatemieh, Arash Termehchy, Hossein Ahmadi, Amir Nayyeri, and Ali Farhadi, who supported me by sharing their experiences with me or making comments on my presentations.

I thank those who helped us in running user studies. Kent Beck, Wayne Beaton, Chris Aniszczyk, and Deepak Azad helped us in recruiting study participants. Fausto Spoto shared a copy of JULIA with us under an academic license. Nathan Hirtz helped us with statistical analyses. I thank the people who participated in our field studies.

I got to know many great people through my internships. When I interned at IBM Research, I interacted with many researchers including Bob Fuhrer, Evelyn Duesterwald, and Mandana Vaziri. My discussions with Bob made me curious about some of the research questions that I later answered through empirical studies.

I enjoyed my time at Google as an intern so much that I went back for a second internship in the same team. My team members, David Morgenthaler, Raluca Sauciuc, Robert Bowdidge, Eddie Aftandilian, Misha Gridnev, Caitlin Sadowski, Tom Szymanski, Dana Dahlstrom, Ben St. John, and Collin Winter contributed to my pleasant experience at Google. Many other Googlers, including Sanjay Bhansali, John Penix, Kevin Bourrillion, Jeremy Manson, Jeffrey van Gogh, and John Micco, gave me suggestions or provided engineering support.
This dissertation was partially supported by the United States National Science Foundation grant number CCF 11-17960 and the United States Department of Energy under contract number DE-F02-06ER25752.

I am indebted to many great teachers that I have had, especially those who showed me the beauty of math and problem solving. Marjan Sirjani showed me the joy of research, and Ramtin Khosravi introduced me to software engineering.

I thank the Iranian community for making Urbana-Champaign feel like home.

I thank my parents. My mother seeded in me the passion for learning, and my father inspired me to pursue a meaningful life. I thank my brother and sister for their support. I thank my spouse, Roshanak, for being such a great companion. She was the closest to me during the graduate program. I do not remember of any major piece of work or decision that she did not help me with. During the difficult times, she provided me with the confidence that I needed to succeed. Her support, both technical and personal, made the graduate program much easier.

Despite my efforts, odds are that I have forgot to acknowledge some people who helped me in finishing this thesis. I apologize in advance and thank all those who helped me in some way.
Table of Contents

Chapter 1  Introduction .................................................. 1
  1.1 Underuse of Automated Refactorings ........................................ 2
  1.2 Research Questions ..................................................... 3
  1.3 Thesis Statement ....................................................... 4
  1.4 Detailed Refactoring Tool Usage Data ..................................... 4
  1.5 Reasons of Underusing Automated Refactorings .......................... 5
  1.6 Revealing More Usability Problems ...................................... 6
  1.7 A Compositional Paradigm of Refactoring ................................. 7
  1.8 A Universal Type Qualifier Inference System ............................... 9
  1.9 Contributions .................................................................. 11
  1.10 Organization .................................................................. 11

Chapter 2  The Need for Richer Refactoring Usage Data .................... 13
  2.1 Limitations of Existing Data ............................................... 14
  2.2 Research Questions ....................................................... 18
  2.3 CODING SPECTATOR and CODING RACKER .............................. 23

Chapter 3  Use, Disuse, and Misuse of Automated Refactorings ............ 25
  3.1 Research Methodology ...................................................... 26
  3.2 Use of Automated Refactoring Tools ...................................... 29
  3.3 Disuse of Automated Refactoring Tools .................................... 32
  3.4 Misuse of Automated Refactoring Tools ................................... 40
  3.5 Limitations .................................................................... 44

Chapter 4  Alternate Refactoring Paths Reveal Usability Problems ......... 45
  4.1 The Origins of the Critical Incident Technique ............................ 47
  4.2 Methodology .................................................................. 48
  4.3 Usability Problems .......................................................... 53
  4.4 Lessons Learned .............................................................. 65
  4.5 Limitations .................................................................. 66

Chapter 5  A Compositional Paradigm of Automating Refactorings .......... 72
  5.1 Frequent Refactoring Sets .................................................. 75
  5.2 Refactoring Composition Patterns ......................................... 78
  5.3 Design of a Compositional Refactoring Tool ............................... 85
  5.4 Survey Study ................................................................. 90
  5.5 Lab Study .................................................................. 96
  5.6 Limitations ................................................................. 102
## Chapter 6  A Compositional Type Qualifier Inference Tool .......................... 106
  6.1 Type Qualifiers .................................................. 108
  6.2 The Illusion of Fully Automatic Type Qualifier Inference .................. 110
  6.3 Design Goals ...................................................... 119
  6.4 Design Process ................................................... 121
  6.5 The Design of CASCADE ........................................ 123
  6.6 Universality Assumptions ........................................ 128
  6.7 Implementation .................................................. 130
  6.8 Evaluation ........................................................ 135
  6.9 Limitations ....................................................... 147

## Chapter 7  Related Work .................................................. 148
  7.1 Empirical Studies of Refactoring .................................... 149
  7.2 Finding the Usability Problems of Refactoring Tools ................. 150
  7.3 Compositional Refactorings ...................................... 152
  7.4 Type Qualifier Inference ......................................... 154
  7.5 Speculative Analysis ............................................. 155

## Chapter 8  Future Work ................................................... 157
  8.1 Appropriate Levels of Automation for Software Engineering Tasks .... 157
  8.2 Finding the Usability Problems of Refactoring Tools ................. 158
  8.3 Compositional Refactoring ...................................... 158
  8.4 Type Qualifier Inference ......................................... 159

## Chapter 9  Conclusions .................................................... 160

## References ............................................................. 163
Chapter 1

Introduction

Automation design is hard, especially for software development tasks. This dissertation focuses on the design of automation support for software evolution tasks, more specifically refactorings, which are behavior-preserving code changes.

Software evolution is inevitable. Any useful piece of code continuously evolves. Developers frequently transform one version of a program to another to fix bugs, accommodate new features, improve the readability of code, etc. For example, the code base of Google receives more than 20 changes per minute and about 50% of the code base changes every month [101]. Such a high rate of code changes incurs a significant engineering cost.

A refactoring is a code change that does not affect the behavior of the program [75, 82, 127]. Refactorings rename, move, split, and join program elements such as fields, methods, packages, and classes. Refactoring facilitates piecemeal growth and the evolutionary design of software. It is a key to modifiable and readable code [75, p. xiii]. Thus, agile software processes such as eXtreme Programming (XP) prescribe refactoring [46].

Refactoring is not only recommended, but also commonly practiced by programmers (Section 3.2) [112,121,169]. For example, based on a case study on the code evolution of the Eclipse Java Development Tools (JDT), Xing and Stroulia found that about 16% of changes were due to standard refactorings [169]. In addition, they found that Eclipse lacks support for some of the complex refactorings that are also frequent, e.g., Inline Interface, Inline Class, and Extract Subclass. Weißgerber and Diehl automatically identified 9,011 refactorings in a total of 7,237 commits of two pieces of software: Tomcat and JEdit [164]. Dig and Johnson found that over 80% of API-breaking changes are refactorings [65]. A survey study at Microsoft shows that more than 95% of programmers refactor across all software development milestones and not only those dedicated to fixing bugs and cleaning up code [97].
Some refactorings are tedious and error-prone to perform manually. Thus, the first refactoring tools were invented more than a decade ago to make the process of refactoring more efficient and reliable [82, 140]. Today, a modern Integrated Development Environment (IDE), such as Eclipse [20], NetBeans [27], IntelliJ IDEA [25], Xcode [33], and ReSharper [28], supports 20–30 automated refactorings.

Studies suggest that automated refactoring is more reliable than manual refactoring. In a study of 12 programmers, Ge et al. found that one third of the manual refactorings introduced bugs [77]. Görg and Weiβgerber found 12 incomplete refactorings in the software repositories of JEdit and Tomcat [79]. On the other hand, studies suggest that modern refactoring tools are relatively reliable. Gligoric et al. automatically invoked the Eclipse refactoring tool at thousands of places in several open-source projects. They reported that the failure rate of the Eclipse refactoring tool was 1.7% for Java and 7.0% for C [78].

Software engineering researchers have proposed many tools to make the process of changing code faster and more reliable [66, 99, 143, 158, 167]. There has been much interest in improving the reliability of existing automated refactorings and building new ones to automate sophisticated program transformations [61, 66, 99, 129, 142, 143, 158, 167]. This is not surprising, given the tedium and error-proneness of some refactorings and the perceived benefits of their automation. Nonetheless, the existing automated refactorings are not used as much as experts like them to.

1.1 Underuse of Automated Refactorings

Programmers refactor frequently [121, 169] and modern IDEs automate many refactorings. However, recent studies show that programmers greatly underuse automated refactorings (Section 3.3) [121,125] and Rename and Extract refactorings constitute most uses of the refactoring tools [112,121]. Murphy-Hill et al. reported that about 90% of the refactorings were performed manually, even though Eclipse provides automated support for these refactorings [121]. This finding was based on a manual inspection of the Eclipse refactoring histories and 40 commits of 12 Eclipse developers. Kim et al. conducted a survey study with 328 engineers at Microsoft [97]. The results indicate that 55% of the participants had automated refactorings in their development environments, 86% of the refactorings were performed manually, 51% of the participants do not use any automated refactoring, and, on average, 56% of the participants do not use an automated refactoring despite being aware of its availability in the development environment. To measure the underuse rate of automated refactorings, we employed a
reliable automatic analysis of a large corpus of data containing over 5,300 manual and automated refactoring [125]. The analysis focused on ten kinds of most frequently used automated refactoring of Eclipse. Despite focusing on the most frequently used automated refactoring, we found that 52% of these refactorings were performed manually.

Complex refactorings are more tedious and error-prone than simple ones to perform manually. Thus, one may expect programmers to rely on the tools for performing complex refactorings more. However, programmers use the refactoring tools mostly for performing simple refactorings such as Rename, Extract Local Variable, and Extract Method (Section 3.3.5.1) [121, 125].

1.2 Research Questions

The software engineering community has focused on identifying recurring program transformations, designing algorithms, and improving the reliability of program transformation tools. On the other hand, the human-automation interaction community has focused on making the design of automation usable by human operators in fields other than computer science. However, none of these works has addressed the challenges of automation design for program transformation tools. Despite the excellent work on designing sophisticated algorithms for automating complex changes, the software engineering community is still faced with major open research questions:

1. Why do programmers underuse the existing refactoring tools?

2. What is an appropriate user interaction model for refactoring tools?

3. How can we design usable tools for supporting refactorings that are more complex than the ones supported by today’s IDEs?

Without pursuing answers to these questions, we are left with an inadequate understanding of the design criteria for usable program transformation tools. This lack of knowledge creates the conditions for a self-sustaining cycle of wishful thinking, inappropriate levels of automation, and superficial claims about the benefits of automating program transformations.

This dissertation fills this knowledge gap and introduces compositional refactoring, a new paradigm of automating refactorings that fits programmers’ practices.
1.3 Thesis Statement

Our thesis statement has two parts:

- Programmers underuse refactoring tools mostly because of usability problems.
- We propose a compositional paradigm of refactoring that makes refactoring tools more usable by automating less.

Through several elaborate studies, we argue that in contrast to previous assumptions, in fact less automation sometimes leads to more reliability and efficiency in program transformation.

This phenomenon is not unique to software engineering. Designers in other fields such as aviation, health-care, and manufacturing have repeatedly observed that highly automated systems that do not integrate the human operator well lead to low productivity and quality. The human-automation interaction community has shown that often a modest design that provides immediate feedback outperforms a highly automated one, concluding that less is sometimes more in automation design [41, 98, 161].

1.4 Detailed Refactoring Tool Usage Data

A barrier to a better understanding of how refactoring tools are used in practice and why they are underused is the lack of detailed refactoring tool usage data. The lack of a detailed usage data makes conducting reliable empirical investigations on the practice of refactoring difficult. Existing refactoring usage data sets are incomplete, coarse-grained, or inconsistent.

For example, the usage data set that the Eclipse foundation has released [23, 24] is incomplete, because it misses some refactoring invocations. In addition, this data set is coarse-grained, because it captures little information about each refactoring invocation beyond the refactoring kind and invocation time.

The Eclipse refactoring histories are another source of refactoring usage data. The Eclipse refactoring histories record more detailed data about the use of the refactoring tool than the Eclipse foundation’s data set. However, the refactoring histories are still coarse-grained and not detailed enough to answer questions about various aspects of automated refactoring, e.g., the cost of configuration and frequency
of previews. These histories are not collected in a central repository, and researchers have access to such refactoring histories of few programmers.

Because different refactoring usage data sets have been collected from different programmers and over different periods of time, they are inconsistent.

To better understand the use of automated refactorings in practice, we developed a custom refactoring usage data collector called CODINGSPECTATOR. To develop CODINGSPECTATOR, we instrumented the Eclipse source code to collect detailed data about the use of the refactoring tool, e.g., the configuration time, selected code snippets, use of the refactoring previews, cancellations, and reported messages.

We designed CODINGSPECTATOR to be unobtrusive and not interfere with programmers’ development activities. The main interaction that programmers have with CODINGSPECTATOR is to use it to submit the data that it collects to a central repository.

We used CODINGSPECTATOR to conduct a field study of programmers’ use of the Eclipse refactoring tool in practice. We deployed CODINGSPECTATOR and invited programmers to install and use it during their daily programming activities. Over 25 programmers installed our Eclipse plug-in. The plug-in recorded more than 2,500 refactorings that were performed using the Eclipse refactoring tool during more than 1,200 programming hours over three months. After an analysis of this quantitative data, we interviewed nine of the programmers to gain insight about their refactoring practices. Our quantitative analyses of CODINGSPECTATOR’s data combined with our qualitative analyses of our interviews with programmers shed light on some of the open research questions related to the practice of refactoring, especially the barriers to the low adoption of refactoring tools.

1.5 Reasons of Underusing Automated Refactorings

We framed the results of our refactoring field study in a general framework [131] from the human-automation interaction literature as use, disuse, and misuse. Use refers to invoking an automated refactoring instead of performing the refactoring manually. Disuse refers to programmers’ neglect or underuse of automated refactorings. Misuse refers to using an automated refactoring in a way not recommended by its designers. The results show that programmers underuse automated refactorings for a variety of reasons that relate to need, invocation, awareness, naming, trust, predictability, configuration.
Our quantitative analysis of the refactoring activities suggests that programmers rely mostly on the automated refactorings for performing simple changes (Section 3.3.5.1). This is almost counterintuitive, because one might expect the programmers to rely on automated refactorings for complex refactorings that are more tedious and error-prone to perform manually.

We found that a major factor that leads to the underuse of a refactoring tool is its poor predictability (Section 3.3.5). Predictability refers to programmers’ ability to foresee the change that an automated refactoring is going to perform on a piece of code before applying the refactoring. An unpredictable refactoring tool is one whose effects on the code are not easily predictable by a programmer. Tools that automate complex refactorings affect several files. It tends to be more difficult to predict which files are going to be changed by such tools and how.

Automated refactorings show previews of the changes that they are going to make, perhaps to achieve predictability. However, our participants previewed only 1% of the automated refactorings that they invoked. This result led us to ask the participants about their infrequent uses of previews. The participants mentioned several reasons for not using the previews much. Some of the problems with refactoring previews are that they only show small portions of the code at a time and that the overhead of examining large changes in a preview dialog is high. This implies that the efficiency of automated refactoring in applying large changes quickly and safely is not sufficient to encourage the use of automated refactorings.

How can we automate complex program transformations in such a way that the programmers are in control and can easily predict the impact of automated changes?

1.6 Revealing More Usability Problems

Our studies show that usability problems are the major barriers to a wide adoption of refactoring tools. This finding highlights the importance of the methodologies for revealing the usability problems of refactoring tools.

Researchers [103, 115, 121] have employed conventional usability evaluation methodologies such as lab studies, interviews, and surveys to find the usability problems of refactoring tools. Every usability evaluation methodology has its limits. Some are limited to a small number of tasks and participants. Some others are limited to what the participants remember of their interactions with the tool. Different
usability evaluation methodologies have different cost-benefit models and are appropriate for use at different stages of software development. Our goal is to add a new usability evaluation methodology to the set of methodologies that an evaluator can use to find the usability problems of refactoring tools. The more usability evaluation methodologies that an evaluator has at his disposal, the more usability problems he can identify.

We adapt the Critical Incident Technique (CIT) to refactoring tools and show that it is effective at finding usability problems. CIT focuses on critical incidents to reveal usability problems. Critical incidents are events during a task that have significant impacts on some aspect of the objective of the study. We found that events such as cancellations, reported messages, and repeated invocations of the refactoring tool are critical incidents and likely to indicate usability problems. We define an alternate refactoring path as a path of user interaction with a refactoring tool that differs from the primary, happy path by involving some critical incident, i.e., cancellation, reported message, or repeated invocations. We analyzed a sample of alternate refactoring paths in a corpus of the Eclipse refactoring tool usage data that we collected during a field study. By analyzing these alternate refactoring paths, we identified 15 usability problems, 13 of which were previously unknown. We reported these problems and proposed design improvements to Eclipse developers. The Eclipse developers acknowledged all of the problems and have already fixed four of them. This result suggests that analyzing alternate refactoring paths is effective at discovering the usability problems of refactoring tools. Although the study focused on a specific refactoring tool, namely, the Eclipse refactoring tool, we expect the results to generalize to other refactoring tools that follow a similar wizard-based paradigm of automating refactorings. Similar adaptations of CIT are likely to be effective even for other interactive program transformation tools, such as those for code completion and automated bug repair.

1.7 A Compositional Paradigm of Refactoring

The results of our field study (Section 1.4) inspired us to develop a new paradigm of refactoring, which we call the compositional paradigm. In the compositional paradigm, the tool automates the small, predictable steps of a complex refactoring and leaves it to the programmer to manually compose the steps into larger refactorings. For example, instead of a monolithic tool that applies a large refactoring such as Extract Superclass in one step, a compositional refactoring tool automates small steps of the
refactoring, in this case, “Create a new superclass for class C”, “Rename class”, and “Move member to immediate superclass”. To perform Extract Superclass in the compositional paradigm, the programmer first creates an empty superclass for an existing class C. Then, he renames the new class and moves members of C to the new superclass one by one. The compositional paradigm provides immediate feedback after each step and removes the upfront configuration cost.

This paradigm seeks to address a major issues with the current wizard-based paradigm of refactoring, i.e., lack of predictability. While others propose more automation for refactorings, we are taking the opposite direction by reducing the automation level of refactoring tools and putting the programmer in control.

The compositional paradigm is grounded in our empirical studies of refactoring. We applied a frequent pattern mining algorithm on a large corpus of refactoring usage data collected by the Eclipse foundation from hundreds of thousands of programmers over the years. We found that programmers frequently invoke a variety of automated refactorings in a short time span (Section 5.1).

The Eclipse foundation data set does not include enough context about each refactoring event. This makes it impossible to tell why a programmer might have invoked one automated refactoring after another. To gain insight into the rationales for invoking multiple automated refactorings in a short time span, we manually inspected a sample of the detailed data that was collected during our field study over several months. This analysis revealed several refactoring composition patterns. A refactoring composition pattern is a recurring set of automated refactorings that programmers compose for similar rationales. These refactoring composition patterns show that the refactorings invoked in a short time span are often semantically related and compose into a larger refactoring. This finding provides evidence for the naturalness of composing refactorings.

We have built an Eclipse plug-in [19] that supports refactorings such as Extract Superclass, Extract Interface, Pull Up, and Push Down in the compositional paradigm. We have conducted survey and controlled studies to evaluate the effectiveness of the compositional paradigm using Extract Superclass as an example. The results of the survey study show that programmers are receptive to the compositional paradigm. Similarly, the results of the controlled study show that the compositional paradigm is more efficient and reliable than the wizard-based paradigm.
1.8 A Universal Type Qualifier Inference System

Our motivation for developing the compositional paradigm of automating refactorings was to assist programmers in performing complex refactorings. After implementing a few of the existing Eclipse automated refactorings in the compositional paradigm and evaluating them, our next question was whether the compositional paradigm can be used to automate even more complex refactorings. We chose type qualifier inference (TQI) as an example of a complex refactoring that today’s IDEs do not support. Type qualifiers \cite{54,72,73} augment an existing type system to check more properties statically. Type qualifiers make it possible to check properties such as safety against null dereferences, SQL injections, and unsafe locking.

A cost of using type qualifiers is the burden of annotating the source code. Realizing this annotation burden, researchers have proposed a variety of type qualifier inference systems \cite{40,55,64,67,68,81,88–91,105,124,137,149,150,153}. The existing TQI tools operate in batch mode. That is, they take a piece of code as input and insert all type qualifiers throughout the code base at once. Although these tools can infer many type qualifiers without getting any input from the programmer, they have several limitations. First, each TQI tool is limited to a specific set of type qualifiers. Building a new TQI tool for a new set of type qualifiers is a nontrivial task that can take weeks to months of development. Second, the batch TQI tools are rigid and support only one kind of code changes, i.e., adding type qualifiers. However, programmers often have to make other code changes to refactor the code to a form that the checker can express. Because the existing tools operate in batch mode, they do not give the programmer the opportunity to refactor the code during the inference process. Third, similar to the existing refactoring tools, existing TQI tools are unpredictable. They insert type qualifiers to the code without giving the programmer any justifications for the inserted type qualifiers.

Normally, TQI and refactoring are thought of as unrelated concepts. However, we consider TQI a refactoring. Because adding type qualifiers to code does not change the behavior of the program, it satisfies the traditional definition of a refactoring. With this insight, we developed CASCADE, a TQI tool that uses the compositional paradigm. The idea of compositional refactoring is to automate small, predictable steps of a large refactoring. Following the compositional paradigm, CASCADE decomposes TQI to code changes that resolve the error messages reported by the type qualifier checker. It parses the error messages and recommends code changes for the ones that report a type qualifier incompatibility.
between two program elements. The changes that CASCADE recommends resolve error messages by propagating the type qualifiers from one program element to the other. It extracts the information about the type qualifiers of the two program elements from the error messages of the checker. Building upon the existing analysis of the type qualifier checker makes CASCADE the first universal TQI system, a system that takes a checker for a set of type qualifiers and turns it into an interactive TQI system.

Speculative analysis [50, 122, 123] is a technique that makes it easier for programmers to see the consequences of their actions on the system. Rather than committing to a decision too early, running into problems later, and having to backtrack, speculative analysis makes it possible for the programmer to see the impacts of one or more actions ahead of time. This ahead-of-time insight can enable the programmer to pursue a better course of actions.

CASCADE uses speculative analysis to present the solution space to the programmer. Type qualifiers propagate. If a programmer annotates a variable with a type qualifier, the other variables that the annotated variable flows into need to have compatible type qualifiers. This property of type systems makes programmers propagate type qualifiers throughout the source code. To show the type qualifiers that the programmer has to propagate after adding a type qualifier, CASCADE adds the type qualifier to a copy of the code, runs the type qualifier checker on the copy, computes the set of new error messages, and finally suggests the code changes to resolve those error messages. CASCADE displays the error messages and suggested code changes in an interactive tree view. The programmer can navigate the tree to see the error messages that each type qualifier change introduces and the type qualifier changes that would fix each error message.

We implemented CASCADE as an Eclipse plug-in and evaluated it through a comparative lab study with 12 participants. The qualitative and quantitative results of the lab study indicate that CASCADE outperforms JULIA [149, 150], a state-of-the-art batch TQI tool, along dimensions such as task completion time, control, and willingness to use.

Once again the results suggest that reducing the level of automation can result in superior results. In this case, CASCADE automates small changes at a time while existing batch TQI tools introduce many type qualifiers at once.
1.9 Contributions

This dissertation makes several major research contributions to the field of refactoring.

**Reasons of underusing automated refactorings.** Through a rigorous field study of programmers refactoring in the real world, it identifies the common reasons that programmers underuse automated refactorings.

**A usability evaluation methodology for refactoring tools.** By adapting the Critical Incident Technique to refactoring tools, it shows that events such as cancellations, error messages, and repeated invocations of automated refactorings are likely indicators of the usability problems of automated refactorings.

**A compositional paradigm of automating refactorings.** By studying how programmers approach complex refactorings, it proposes the compositional paradigm of automating refactorings. In this paradigm, the tool designer decomposes a large refactoring into a set of smaller, more predictable automated program transformations and the programmer manually composes this set of primitive automated program transformations to make larger changes.

**A universal type qualifier inference system.** Through a combination of compositional refactoring and speculative analysis, it presents CASCADE, the first universal system for inferring type qualifiers. Our user study shows that CASCADE outperforms a state-of-the-art type qualifier inference tool that operates in batch mode.

1.10 Organization

Some of the chapters of this dissertation are derived from our prior publications.

Chapter 2 overviews the existing refactoring data sets and discusses their limits with respect to our research questions. This chapter is derived from our paper that appeared in the proceedings of the workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) [157].

Chapter 3 presents the factors that lead to use, underuse, and misuse of automated refactorings. The findings are based on the quantitative and qualitative analyses of the data that we collected during a field study. This chapter is based on a paper that we published in the proceedings of the International Conference on Software Engineering (ICSE) [156].
Chapter 4 shows how analyzing events such as error messages, cancellations, and repeated invocations can reveal usability problems of refactoring tools. A paper that presents this methodology and its evaluation appeared in the proceedings of the International Conference on Software Engineering (ICSE) [159].

Chapter 5 presents a new paradigm of automating refactorings, called compositional refactoring. This work has been published in the proceedings of the European Conference on Object-Oriented Programming (ECOOP) [155].

Chapter 6 uses the compositional paradigm to automate the inference of type qualifiers in a usable and universal way.
Chapter 2

The Need for Richer Refactoring Usage Data

More detailed data than what are currently available are required to answer important questions about the use of automated refactorings in practice. For example, why do programmers not use the automated refactorings more? Perhaps programmers just need more training in the tools, the tools implement the wrong refactorings, or most of the refactorings are not done frequently enough to justify their automation. Is the problem that the tools have poor user interfaces? Perhaps they are based on a flawed understanding of how programmers use these tools in their workflows.

Some of the questions cannot be fully answered by solely relying on the usage data. Therefore, we interview programmers to better understand their perceptions of the refactoring tools and the problems that they find with these tools. Richer refactoring usage data will also enable us to ask more specific questions during the interviews and get more accurate answers.

Almost all we know about the usability of refactoring tools comes from the studies based on a few coarse-grained, inconsistent, or incomplete sets of data (Section 2.1). There is a large amount of coarse-grained data, but, only a small amount of fine-grained data from a few programmers on the usage of refactoring tools. These data sets are inconsistent and difficult to correlate because they capture different data over different time intervals from different programmers. Others have reported some interesting statistics on the usage of refactoring tools based on these data [121]. However, we need to study more aspects of automated refactorings such as their configurations, previews, and failures to discover the major usability problems of automated refactorings.

The limitations of the existing data have motivated us to develop our own usage data collectors, CODINGSPECTATOR [17] and CODINGTRACKER [18], for capturing richer data about high-level refactorings and low-level code edits. We have combined the data of these two data collectors and our interviews with programmers to answer several open research questions about the use of refactoring tools (Section 2.2).
This chapter first gives an overview of the existing data on the usage of refactoring tools (Section 2.1). Then, it discusses some of the current findings about the usability of refactoring tools and presents several open research questions (Section 2.2). Answers to these questions will give us a better understanding of the major usability problems of mainstream refactoring tools. However, we show that the existing data sets are not sufficient to answer these research questions. So, we will present our technique for collecting richer and more consistent data about automated and manual refactorings (Section 2.3). Our richer usage data and interviews will make it possible to answer some of the open questions that are vital to designing the next generation of refactoring tools that better align with how programmers work (Chapter 8).

### 2.1 Limitations of Existing Data

Prior work [112, 118, 121] has analyzed multiple sets of data to study the usability of refactoring tools. The generalizability and number of conclusions that can be drawn from these studies depend on the quality and quantity of these data sets. This section discusses the strengths and weaknesses of each set of refactoring usage data that prior research has studied to understand the usability characteristics of refactoring tools.

#### 2.1.1 Mylyn Monitor

*Mylyn Monitor* was the first usage data collector tool for Eclipse [112]. It captured data about the use of various features and plug-ins of Eclipse. Mylyn monitor captured the interaction history of each user. The interaction history included the time of invoking commands and changes to views, perspectives, and editor selections. A subset of the interaction history captured information about the refactoring commands. *Mylyn Monitor* collected data from 41 programmers. Since its focus was not refactorings, it did not collect detailed information about how programmers interacted with automated refactorings, e.g., the configurations of automated refactorings.
2.1.2 Aggregated Eclipse Usage Data

From 2009 until 2012, official releases of Eclipse came with a plug-in called the Eclipse *Usage Data Collector (UDC)* [23]. UDC recorded various events including the usage of all perspectives, views, and commands in Eclipse. It also recorded invocations of refactoring as commands, and captured the ID and the time of invocation of every command locally. Then, it regularly uploaded the collected data to the Eclipse foundation servers, if the user agreed to share his or her data.

The Eclipse foundation aggregated and published the data every year. The aggregated data reports the total number of invocations of each command by all users during every month. UDC has been in operation since April 2008. So far, the Eclipse foundation has published the aggregated data until January 2010. Figure 2.1 shows a sample of the aggregated Eclipse usage data publicly available on the UDC website.

Since UDC is pre-installed in most releases of Eclipse and captures coarse-grained data that do not contain sensitive information, many users agree to submit their data. On average, UDC has received data from 168,100 users per month. Even though not all users of UDC use Eclipse for Java programming, a significant fraction of them invoke Java specific commands.

The UDC data record what automated refactorings get invoked. However, they do not contain more detailed information about how an automated refactoring is performed, e.g., how the user has configured the tool. In addition, there is no one-to-one mapping between the IDs of the commands of the UDC data and the refactoring IDs. For instance, the UDC data do not distinguish the Rename Local Variable refactoring from the Rename Method refactoring. The UDC data distinguish the six variants of the Extract refactoring but not the ten, four and three variants of the Rename, Move and Inline refactorings, respectively. The impacts of these refactorings depend on the program elements on which they are invoked. For example, the impact of renaming a local variable is limited to a method while renaming a method could affect multiple files. Since the impacts of these refactorings are different, the users might use them differently. Therefore, it is useful to differentiate the refactorings performed on different kinds of program elements to study how programmers use such refactorings.
2.1.3 Time-stamped Eclipse Usage Data

The Eclipse UDC plug-in captures the time of invocation of every command locally [23]. The Eclipse foundation has published the time-stamped usage data from January 2009 until August 2010 [24]. The UDC set of data with timestamps reports the exact time rather than the year and month of invocation of every command. The time-stamped usage data make it possible to study the refactoring activities in a small window of time. Figure 2.2 shows a sample of the time-stamped Eclipse usage data from our own local system.

2.1.4 Eclipse Refactoring Histories

Eclipse logs completed refactorings locally. The intent of this kind of log is to replay the refactorings. Eclipse represents every refactoring operation as a refactoring descriptor. Refactoring descriptors make it possible to decouple the back-end refactoring engine from the front-end user interface of refactoring tools. A refactoring descriptor contains enough information to replay the refactoring on the same source code. In addition to the ID and the time of invocation, a refactoring descriptor stores all the parameters and configurations provided on the user interface of the refactoring tool. Figure 2.3 shows a sample of the data collected in the Eclipse refactoring histories.

When the user performs an automated refactoring, Eclipse creates a refactoring descriptor and logs it in its local refactoring history. If the user undoes the automated refactoring later, Eclipse will remove the corresponding refactoring descriptor from the refactoring history.
Figure 2.3: Sample of the data in Eclipse refactoring histories

The Eclipse refactoring histories are more detailed than the UDC data, but these two data sets are not consistent (Sections 2.1.2, 2.1.3). The UDC data capture every refactoring command that is initiated, while refactoring histories only capture the refactorings that have been completed and have not been later undone.

Since the Eclipse refactoring histories have been designed to replay refactorings, they do not capture information about the failures of automated refactorings and the error messages that they report. However, a better understanding of the failures of refactoring tools might lead to improvements in the usability of such tools [115].

Although the Eclipse refactoring histories are richer than UDC data, there has been no systematic mechanism for collecting the Eclipse refactoring histories to a central repository, and few refactoring histories are available. Robbes has reported about the usage of refactoring tools by himself and another programmer [139]. Four programmers who primarily maintain the automated refactorings of Eclipse have shared their refactoring histories with several researchers. Additionally, eight programmers of the Eclipse Mylyn project used to check their refactoring histories in their CVS repository. Nonetheless, Eclipse started to capture some of the refactorings only in the middle of the data collection from these programmers [121]. As a result, the refactoring histories of these 12 programmers are incomplete, because the data about some of the early automated refactorings are missing.
2.1.5 Version Control Histories

Version control systems contain the source code of many open source projects. Several researchers have studied the histories of version control systems for refactoring activities [63, 96, 121, 169]. Finding the refactorings between two versions of the source code requires sampling, metrics, and heuristics. The following are some of the disadvantages of relying on the version control histories for studying refactorings:

1. Programmers may not commit all their refactorings to version control systems.

2. Sometimes, there are different ways to refactor one version of the code to another. In such cases, it is often challenging to determine which sequence of refactorings have led to a particular version of the source code.

3. Most of the time, it is almost impossible to tell whether a refactoring has been performed manually or through a tool by just comparing two snapshots of the source code. Therefore, some have tried to match the version control commits with Eclipse refactoring histories to distinguish the automated and manual refactorings [121]. However, few programmers have made their refactoring histories available (Section 2.1.4).

2.2 Research Questions

Recent work has posed several research questions about the use of refactoring tools [112, 121] and discussed different methods for studying the use of such tools [118]. This section categorizes and discusses these research questions and introduces a few new ones. We briefly discuss the existing work and possible implications of answers to each set of questions. These open questions require richer data about the usage of refactoring tools beyond what are currently available. However, even our richer usage data will not be sufficient to answer all of our questions. For example, we may not capture enough data about automated refactorings that programmers rarely use. Therefore, we will conduct interviews and combine quantitative and qualitative approaches to gain deeper insight [60]. Table 2.1 lists the different categories from this section and summarizes whether existing data sources can be used to answer the questions.
Table 2.1: Can we use the existing data sources to help answer our research questions? Y (Yes); P (Partially); N (No)

<table>
<thead>
<tr>
<th>Research Aspects</th>
<th>Mylyn Monitor</th>
<th>UDC Data</th>
<th>Refactoring Histories</th>
<th>Refactoring Version Histories</th>
<th>Refactoring and Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual and automated refactorings (Section 2.2.1)</td>
<td>N</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>Awareness (Section 2.2.2)</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Methods of invocation (Section 2.2.3)</td>
<td>P</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>Configuration of automated refactorings (Section 2.2.4)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>Previewing the outcomes of automated refactorings (Section 2.2.5)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Exceptional cases (Section 2.2.6)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Cancellations and undos (Section 2.2.7)</td>
<td>P</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

2.2.1 Manual and Automated Refactorings

Manual refactorings are believed to be error-prone and automated refactorings are supposed to help programmers perform refactorings quickly and correctly [165]. Nonetheless, a study of the Eclipse refactoring histories and version control histories of two Eclipse development teams suggested that automated refactorings are underused [121], that is, programmers opt to perform most refactorings manually despite the availability of automated support. Prior research [121] has also suggested that programmers intersperse refactorings with other edits.

Answers to the following questions will provide a better understanding of how programmers combine automated refactorings with manual changes and the characteristics of automated refactorings that are underused. This understanding may have implications for designing tools that are used more frequently and better match the workflow of programmers.

1. How do programmers intersperse refactorings with other kinds of program changes?

2. How common are simple and complex automated refactorings?

3. How frequent are refactorings?

4. Are automated refactorings underused?

5. How often are different refactorings performed with and without tools?
Eclipse refactoring and version control histories cannot be used to answer the above questions reliably (Sections 2.1.4 and 2.1.5). We need to collect data from more programmers about the fine-grained edits, and not just coarse-grained commits of version control systems. Capturing fine-grained edits also allows us to find out how the uses of automated refactorings differ for intermediate refactorings and the final refactorings found in consecutive snapshots of a version control system.

2.2.2 Awareness

A survey conducted on a few programmers suggested that programmers do not use some automated refactorings because they are unaware of the tools [121]. Our interviews with four students of the software engineering course at the University of Illinois corroborate this observation. Although the students had received training on several refactorings, they were unaware of the automated support for many others. More quantitative and qualitative data are required to answer the following questions:

1. What refactorings do programmers frequently underuse because of unawareness?

2. How do programmers become aware of certain automated refactorings?

Not all questions about the awareness of automated refactorings can be answered based on the usage data. We can capture data only about automated refactorings that programmers are aware of. More interviews with programmers are required to better understand the effect of awareness on the use of automated refactorings.

2.2.3 Methods of Invocation

There are different ways to invoke an automated refactoring. A programmer may invoke the refactoring by selecting a program element in the editor or a structured view. Also, the programmer could use the Quick Assist feature of Eclipse to get a list of applicable refactorings on the current selection. Quick Assist narrows down the decision space of programmers by proposing only a subset of refactorings applicable to the current context. Other methods of invoking refactorings have also been proposed [114]. However, little is known about the impact of each invocation method on the use of automated refactorings [112]. It is not clear if programmers prefer to invoke refactorings from within editors or graphical representations.
of code. Also, we do not know if light-weight methods of invocations such as Quick Assist encourage programmers to use automated refactorings more often.

Some automated refactorings operate on specific program elements and are unavailable on others. Prior research [115] suggests that it is sometimes tricky to select a valid piece of code for invoking the Extract Method refactoring. However, it is still unclear how prevalent the selection problems are for all automated refactorings. Based on these observations, we find it essential to seek answers to the following questions in order to invent better ways of invoking the automated refactorings:

1. What refactorings are more difficult to invoke? What refactorings do programmers fail to invoke because of wrong input selections?

2. How do programmers prefer to select the input program elements of automated refactorings? Do they prefer textual or structural selections?

3. How often do programmers use the Quick Assist feature of Eclipse to invoke automated refactorings?

2.2.4 Configuration of Automated Refactorings

Most automated refactorings are configurable. For instance, the wizard of the Extract Method refactoring in Eclipse provides options to configure the access modifier, declared exceptions, and comment of the new method. Knowing what configuration options programmers frequently use or ignore helps tool designers streamline the user interface.

Prior work [121] has analyzed the Eclipse refactoring histories (Section 2.1.4) of two development teams to understand how often these programmers have changed the default configurations of their tools. These results have raised the following new set of questions that cannot be answered using any of the existing sets of refactoring usage data (Section 2.1).

1. Do programmers manually change the outcomes of automated refactorings? Do these changes indicate any options that are missing from the user interfaces of automated refactorings?

2. Under what circumstances do programmers change the default options of their tools? Can the automated refactorings suggest better default options based on the recent activities of the programmer in the IDE?
3. How long do programmers spend on configuring various automated refactorings? Are long configuration times indicators of usability problems?

2.2.5 Previewing the Outcomes of Automated Refactorings

Most automated refactorings offer the option of previewing their proposed changes. Typically, for each file that will be affected by the refactoring, the preview dialog shows two versions: the one before applying the refactoring and the one after.

One programmer raised concerns about the usability of the preview dialog in a survey [121]. None of the existing data sets contain data about previews of refactorings and we do not know of any empirical research on this aspect of refactoring tools. We collect data about the preview actions of programmers to answer the following questions (Section 2.3):

1. How often do programmers preview automated refactorings?

2. What refactorings are programmers more likely to preview?

2.2.6 Exceptional Cases

Automated refactorings try to guarantee that their changes will be behavior preserving. If the tool detects that a particular use of it may change the observed behavior of the program, it will report some error message to the programmer. The automated refactorings of Eclipse rate the severity levels of their messages as information, warning, error, or fatal error. Eclipse prevents the programmer from continuing an automated refactoring only if it finds a fatal error.

Others have observed that the Extract Method refactoring of Eclipse communicates error messages to its user poorly [115]. By collecting more data about the error messages reported by automated refactorings (Section 2.3), we will be able to answer interesting questions such as the following about the usability of many refactorings.

1. What automated refactorings present error messages more frequently?

2. What are the most frequent error messages about?

3. How do error messages affect the programmer’s future uses of the tool?
2.2.7 Cancellations and Undos

Others have found that undo and erase events are indicators of usability problems in creation oriented applications such as SketchUp [29,35]. Similarly, we hypothesize that cancellations and undos of automated refactorings may be indicators of usability problems. However, none of the existing data sets capture such events and we are unaware of any research study on such refactoring events. Collecting data about the cancellations and undos of automated refactorings can shed light on the following questions:

1. What are the most highly canceled and undone automated refactorings?

2. Does the number of cancellations and undos correlate with other indicators of complexity such as complicated error messages, size of the refactoring, and the amount of time spent on configuring the automated refactoring?

3. Do the error messages of automated refactorings influence programmers to cancel or undo their refactorings?

2.3 CODINGSPECTATOR and CODINGTRACKER

To address the limitations of existing data sources (Section 2.1) and to answer the open research questions about the use of refactoring tools (Section 2.2), we have developed our own minimally intrusive usage data collection tools: CODINGSPECTATOR [17] and CODINGTRACKER [18]. Both tools modify Eclipse to capture more data. Over 25 programmers have used our data collectors in real-world settings. Based on the feedback from our participants, we have iteratively improved the user experience and data collection capabilities of CODINGSPECTATOR and CODINGTRACKER so that they do not slow down or crash Eclipse or interfere with programmers' work.

The following is a sample of the data that CODINGSPECTATOR collects about refactorings:

1. whether the refactoring was performed, canceled or unavailable

2. the kind of refactoring (Rename Local Variable, Move Static Method, Extract Method, etc.)

3. when the refactoring was invoked

4. the kind of selection that was made to invoke the refactoring (textual or structural)
5. the selected source code element

6. whether the refactoring was invoked using Quick Assist

7. how long the programmer spent on each step of the refactoring (configuration, preview, error message comprehension)

8. how the programmer has configured the refactoring

9. any problems reported by the refactoring tool

CODINGTracker captures fine-grained edit operations down to the level of an insertion or deletion of a character in the Java editors of Eclipse. It captures the edits so precisely that it can later replay them to show the code changes in action. In addition, CODINGTracker collects other data including the undone refactorings and commits to version control systems.

CODINGSpectator and CODINGTracker gather consistent data that we combine for our analyses. The design of CODINGTracker and the studies based on its data have been discussed elsewhere [125, 126]. The results presented in this dissertation rely mostly on the data collected by CODINGSpectator. Our rich sets of usage data and interviews allow us to answer all the open research questions we presented in Section 2.2. However, capturing rich usage data from real programming environments is challenging, because it raises privacy issues and makes it difficult to recruit participants.
Chapter 3

Use, Disuse, and Misuse of Automated Refactorings

To understand the factors that lead to the low adoption of refactoring tools, we conducted a study consisting of both quantitative and qualitative data collection. We studied 26 programmers working in their natural settings on their code for a total of 1268 programming hours over three months, and collected data about their interactions with automated refactorings. We observed patterns of interaction in our quantitative data and interviewed nine of our participants to take a more detailed qualitative look at the behavioral data. Then, we adapted a general framework of human-automation interaction [131] to frame the use, disuse, and misuse of automated refactorings. Use of automated refactorings refers to programmers applying automated refactorings to perform code changes they might otherwise do manually. Disuse of automated refactorings is programmers’ neglect or underuse of automated refactorings. Misuse of automated refactorings refers to programmers’ use of these tools in ways not recommended by the designers.

Our empirical study sheds light on how programmers interact with automated refactorings. First, the results show that a single context-aware and lightweight method of invoking refactorings accounts for a significant number of refactoring invocations (Section 3.2). Second, we have found several factors that lead to the underuse of automated refactorings such as need, awareness, naming, trust, predictability, and configuration (Section 3.3). Third, the data indicates that programmers usually continue an automated refactoring that has reported some error or warning. This finding casts doubt on the main property of automated refactorings, namely, behavior-preservation. In addition, we report our observations of unjustified uses of the refactoring tool (Section 3.4). Finally, we propose alternative ways of designing refactoring tools based on the findings of the study (Sections 3.2.2, 3.3.7, and 3.4.3).
3.1 Research Methodology

To understand why existing automated refactorings are underused, we analyzed a large corpus of interaction data gathered from 26 programmers over three months. In addition, we conducted a set of nine semi-structured interviews with programmers to understand the rationales of their refactoring practices.

3.1.1 Participants

We recruited 16 programmers working on research projects at the University of Illinois at Urbana-Champaign. Eleven of these internal programmers were enrolled in Computer Science graduate programs, and the remaining five were research interns. We also recruited 10 external programmers by sending more than 25 individual emails and posting recruitment messages to the mailing lists and IRC channels of over 40 open-source Java projects.

We asked every participant to fill out a brief survey that collected some demographic information including their years of programming experience and projects. We received the survey results of 24 participants. Based on the survey, 2, 4, 11, and 7 of our participants had 1–2, 2–5, 5–10, and more than 10 years of programming experience, respectively. Our participants reported that they had been working on a diverse range of projects such as banking, business process management, marketing, database management, and projects of six research labs at the university.

3.1.2 Data Collection

We gathered the interaction data using our two minimally intrusive data collectors for the Eclipse IDE: CODINGSPECTATOR and CODINGTRACKER (Section 2.3). Our participants used these tools for about 1268 hours (mean = 49, sd = 46).

Our data collectors were developed to capture data regarding the failures of automated refactorings, context of the failure, configuration overhead, and invocation methods.

CODINGSPECTATOR captures data about the use of automated refactorings while CODINGTRACKER collects all manual edits. CODINGSPECTATOR collects three kinds of events: canceled, performed, and unavailable. Canceled events are triggered when the programmer cancels an automated refactoring. Performed events occur when the programmer applies an automated refactoring. Unavailable events
are triggered when the programmer tries to invoke an inapplicable automated refactoring and Eclipse reports an error.

Eclipse creates *refactoring descriptor* objects for some invocations of automated refactorings and stores them in an XML format. CODINGTracker captures the descriptors of all refactorings created by Eclipse, and CODINGSpectator creates refactoring descriptors of its own, which capture more data than those of Eclipse. CODINGSpectator supported 23 of the 33 automated refactorings supported by Eclipse during the study.

CODINGSpectator records the following information in its refactoring descriptors:

1. the time of occurrence of every refactoring event
2. the identifier of the automated refactoring
3. configurations, e.g., input elements, project, and settings that affect the result of the tool
4. information about the selection used to invoke the automated refactoring and its context
5. whether the refactoring tool was invoked using Quick Assist
6. the problems reported by each invocation of an automated refactoring
7. the time spent on each page of the refactoring wizards

Figure 3.1 illustrates the refactoring descriptor that CODINGSpectator captures for an application of the Extract Method refactoring. Due to privacy issues, we use our own examples instead of our participants’ data.

CODINGTracker records the edits made inside the Java editors of Eclipse so precisely that it can later replay them to show the code evolution in action. We replayed some of the code edits to obtain more context about some of the refactoring events and estimate the number of lines and files affected by each automated refactoring. We also used CODINGTracker to estimate the number of programming hours of each participants. We computed the number of programming hours by adding up the time intervals between consecutive CODINGTracker events that were at most half an hour long. Both CODINGSpectator [17] and CODINGTracker [18] are open source and publicly available.
void printInfo(double amount) {
    printBanner();
    System.out.println("Amount: " + amount);
}

<refactoring
    stamp="1317326947775"
    id="org.eclipse.jdt.ui.extract.method"
    comment="Extract method 'private void
    printDetails(double amount)' from 'Printer.printInfo()'
    to 'Printer'" exceptions="false" input="/src/<Printer.java"
    name="printDetails"
    code-snippet="
    void printInfo(double amount) {
        printBanner();
        System.out.println("Amount: " + amount);
    }"
    selection-text="
    System.out.println("Amount: " + amount);"
    invoked-by-quickassist="false"
    status="<OK>
    navigation-history="
        {[ExtractMethod,BEGIN_REFACTORING,1317326935617],
        [ExtractMethodInputPage,Preview>,1317326940477],
        [PreviewPage,OK,1317326947379],}
    />

Figure 3.1: The descriptor captured by CODINGSPECTATOR for an Extract Method refactoring invoked on the highlighted statements of method printInfo() in the top box. See the numbered list of items in Section 3.1.2 for a description of each group of attributes.

The analysis of the interaction data was complemented by conducting semi-structured interviews with nine of our internal participants. Each interview lasted about an hour. During the interviews, we asked questions about participants' awareness and use of automated refactorings. In addition, we prompted the participants with the detailed data that our data collectors had captured and asked them questions about their specific usage patterns such as the refactorings they had performed or canceled, the pieces of code they had refactored, the selections and methods they had used to invoke the refactorings, and the refactoring problems they had received from the tools. In this chapter, we refer to the i-th interviewee as I_i.
3.1.3 Data Analysis

We used theoretical sampling [59] to decide what data to collect and whom to interview. For instance, based on the results of our pilot study on 14 undergraduate students at the university, we decided to study more experienced programmers for a longer period of time. We used an inductive approach for analyzing the qualitative data to reliably decide whether two interviewees had provided equivalent responses. We coded the interview scripts to derive the common themes of the interview responses. We then listed all the responses belonging to each code, and constantly compared and revised the codes until they saturated. Sections 3.2, 3.3, and 3.4 present the core categories of our data, namely, use, disuse, and misuse, and their related categories in subsections.

3.2 Use of Automated Refactoring Tools

Decisions about the use of automated tools depend on a complex interaction of a variety of factors and are subject to personal differences. The human-automation interaction literature has studied the roles of personal attitudes, mental workload, cognitive overhead, trust, confidence, risk, and other factors on human use of automation [131]. This section discusses the impact of invocation method on the use of automated refactorings.

3.2.1 Invocation Method

Eclipse supports several ways of invoking refactorings. The programmer can go to the “Refactor” menu, right click, or use shortcut keys to invoke refactorings. Alternatively, the programmer may use Quick Fix or Quick Assist (CTRL+1) to invoke some of the automated refactorings. Quick Fix assists programmers in resolving compilation problems. Eclipse shows a small icon close to the location of each compilation problem if a Quick Fix is available for the problem. Quick Fix sometimes offers a refactoring to resolve compilation warnings. On the other hand, Quick Assist is not tied into compilation problems, and programmers can use it to perform some common changes such as Rename and Extract Method (See Figure 3.2 for an example of Quick Assist).

Our results suggest that programmers prefer to quickly apply an automated refactoring and tweak its outcome later rather than spend time configuring the tool up front. Based on the data in Table 3.1,
Table 3.1: Data about the usage of automated refactorings from 26 programmers for about 1268 hours over three months. The CS subscript indicates the data captured using CodingSpectator, and the CT subscript indicates the data captured using CodingTracker. In the complexity column, S = Simple, M = Moderate, and C = Complex. ConfigCS is the average configuration time (seconds) of 788 refactorings. LinesCT and FilesCT are the average numbers of affected lines and files computed using the data available for 93% of the performed refactorings captured by CodingTracker. Pr(P | W)CS = Pr(Performed | Warning) and Pr(P | E)CS = Pr(Performed | Error). The symbol “-” indicates an unknown or undefined value. PerformedCS is less than PerformedCT for Change Method Signature because CodingSpectator did not support this refactoring from the beginning of the study.

| Automated Refactoring                  | Complexity | PerformedCS | CanceledCS | WarningCS | ErrorCS | Failed/WarningCS | Quick/HandCS | ConfigCS (sec) | LinesCT | FilesCT | Pr(P | W)CS | Pr(P | E)CS |
|---------------------------------------|------------|-------------|------------|-----------|---------|-----------------|--------------|---------------|---------|---------|-----------|-----------|
| Change Method Signature               | C          | 45          | 49         | 8         | 1       | 9               | 0            | -             | 0.8     | 7.48    | 2.31      | 1.00      |
| Convert Anonymous Class to            |            | -           | 3          | -         | -       | -               | -            | -             | 35.00   | -       | -         | -         |
| Nested                                | S          | 97          | 97         | 1         | 0       | 0               | 1            | 83            | 0.56    | 3.65    | 1.00      | -         |
| Encapsulate Field                     | M          | -           | 225        | -         | -       | -               | -            | -             | -       | 10.44   | 1.10      | -         |
| Extract Class                         | C          | -           | 15         | -         | -       | -               | -            | -             | 120.75  | 2.25    | -         | -         |
| Extract Constant                      | S          | 29          | 29         | 0         | 0       | 1               | 0            | 26            | 0.39    | 3.72    | 1.00      | -         |
| Extract Interface                     | M          | 2           | 2          | 4         | 2       | 0               | 0            | -             | 25.67   | 36.00   | 1.50      | 0.00      |
| Extract Local Variable to Field       | S          | 606         | 606        | 14        | 6       | 8               | 475          | 0             | 3.30    | 5.10    | 1.00      | -         |
| Extract Method                        | M          | 186         | 186        | 30        | 0       | 13              | 0            | 62            | 5.50    | 12.21   | 1.00      | -         |
| Extract Superclass                    | C          | 0           | 0          | 3         | 1       | 1               | 0            | 0             | 0.30    | -       | -         | 0.00      |
| Generalize Declared Type              | M          | -           | 0          | -         | -       | -               | -            | -             | -       | -       | -         | -         |
| Infer Generic Type Arguments          | C          | -           | 7          | -         | -       | -               | -            | -             | 1.29    | 0.57    | -         | -         |
| Inline Constant                       | S          | 38          | 38         | 0         | 0       | 0               | 0            | 0             | 0.50    | 1.00    | 1.00      | -         |
| Inline Local Variable                 | S          | 182         | 182        | 1         | 0       | 0               | 0            | 73            | 0.40    | 3.26    | 1.00      | -         |
| Inline Method                         | S          | 63          | 63         | 8         | 0       | 3               | 1            | 0             | 1.50    | 9.97    | 1.13      | 0.67      |
| Introduce Factory                     | M          | -           | 0          | -         | -       | -               | -            | -             | -       | -       | -         | -         |
| Introduce Indirection                 | M          | -           | 2          | -         | -       | -               | -            | -             | -       | -       | -         | -         |
| Introduce Parameter                   | C          | -           | 46         | -         | -       | -               | -            | -             | 4.74    | 1.52    | -         | -         |
| Introduce Parameter Object            | C          | -           | 0          | -         | -       | -               | -            | -             | -       | -       | -         | -         |
| Move                                  | C          | 147         | 147        | 8         | 0       | 3               | 3            | 10            | 5.50    | 12.01   | 2.19      | -         |
| Move Method                           | C          | 0           | 0          | 10        | 0       | 3               | 0            | 0             | 16.50   | -       | -         | 0.00      |
| Move Static Member                    | M          | 6           | 6          | 1         | 1       | 0               | 0            | 0             | 13.00   | 45.20   | 3.20      | 1.00      |
| Move Type to New File                 | S          | 7           | -          | -         | -       | -               | -            | -             | 54.50   | -       | -         | -         |
| Pull Up                               | C          | 9           | 9          | 0         | 5       | 0               | 0            | -             | 8.90    | 11.78   | 2.89      | 1.00      |
| Push Down                             | C          | 8           | 8          | 3         | 2       | 3               | 0            | -             | 9.30    | 32.25   | 12.75     | 0.50      |
| Rename Class                          | M          | 276         | 276        | 37        | 41      | 16              | 5            | 20            | 8.90    | 12.13   | 3.06      | 0.93      |
| Rename Enumeration Constant           | S          | 3           | 3          | 0         | 2       | 0               | 0            | 3             | 0.28    | 4.52    | 1.47      | 0.94      |
| Rename Field                          | M          | 125         | 125        | 9         | 16      | 3               | 6            | 6             | 2.80    | 4.52    | 1.47      | 0.94      |
| Rename Local Variable                 | S          | 465         | 465        | 14        | 4       | 16              | 6            | 20            | 0.23    | 3.37    | 1.00      | 0.50      |
| Rename Method                         | M          | 260         | 260        | 17        | 33      | 16              | 9            | 15            | 0.74    | 3.45    | 2.20      | 0.94      |
| Rename Package                        | M          | 12          | 12         | 0         | 4       | 0               | 0            | 0             | 0.68    | 4.67    | 2.75      | 1.00      |
| Rename Type Parameter                 | S          | 6           | 6          | 0         | 0       | 0               | 0            | 0             | 0.27    | 2.17    | 1.00      | -         |
| Use Supertype Where Possible          | C          | 0           | 0          | 7         | 0       | 0               | 0            | -             | 1.56    | -       | -         | -         |

<table>
<thead>
<tr>
<th>Total Counts</th>
<th>Weighted Averages</th>
<th>Overall Pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2565</td>
<td>2874</td>
<td>0.88</td>
</tr>
<tr>
<td>175</td>
<td>121</td>
<td>0.68</td>
</tr>
<tr>
<td>91</td>
<td>31</td>
<td>-</td>
</tr>
<tr>
<td>783</td>
<td>37</td>
<td>-</td>
</tr>
<tr>
<td>6.30</td>
<td>6.71</td>
<td>1.47</td>
</tr>
<tr>
<td>0.88</td>
<td>0.68</td>
<td>-</td>
</tr>
</tbody>
</table>

our participants used Quick Assist to perform the refactorings that it supports 35% of the time. Our participants relied on Quick Assist to perform the Rename refactoring less than other refactorings perhaps because Rename is so frequent that they had learned its shortcut key. If we exclude the Rename
refactorings, our participants used Quick Assist to invoke 65% of the refactorings. This paper reports the first quantitative results on the use of Quick Assist for performing refactorings.

Of the six interviewees who were aware of Quick Assist, five used it as their primary method of invoking the refactorings supported by it. Quick Assist is a popular method of invoking automated refactorings, because it can be quickly invoked via keyboard, narrows the decision space by proposing only a handful of transformations that are applicable to the selected context, and makes it easier to configure refactorings by using some default settings and not opening a dialog. However, we noticed that at least two of our interviewees were not aware that Quick Assist had used some non-default settings from the last configuration of the refactoring.

All of our interviewees were aware of Quick Fix. This awareness could be a result of the visual element that indicates the availability of Quick Fix. Three of our interviewees did not know about the Quick Assist feature. Nevertheless, those who were aware of it heavily relied on it to both discover and invoke automated refactorings. I₁ told us:

Most of them [the automated refactorings] I know about by using Quick Assist. I very seldom go into the refactoring menu and only when there is a refactoring that I cannot reach through Quick Assist and I don’t know about […]. Quick Assist will tell me if they are applicable in a certain context. […]. It always annoys me when they [automated refactorings] are not available through Quick Assist like Change Method Signature. […] I really like Quick Assist.
As another example, when we introduced $I_2$ to the Introduce Parameter refactoring, he commented:

That's [the Introduce Parameter refactoring] actually pretty cool. I never knew about the existence of this. I've done this a few times manually, and I always wondered if it's possible to do this automatically. Yeah, I'll probably try it. Does this show up in Quick Assist?

3.2.2 Implications

We have found that programmers prefer lightweight methods of invoking refactorings such as Quick Assist. Therefore, we suggest that other IDEs such as IntelliJ IDEA [25] and NetBeans [27] support refactoring invocation methods similar to Quick Assist.

We noticed that Quick Assist was a somewhat hidden feature of Eclipse. Some programmers will not know about this feature until they somehow learn about the magic shortcut key. More programmers know about Quick Fix because it has a visual representation. This observation suggests that recommending refactorings similar to the way Quick Fix recommends fixes for compilation problems might promote the use of automated refactorings. While Quick Fix removes compilation problems, automated refactorings remove code smells. Code smells are common deficiencies of code that make it less readable and reusable [75, p. 75]. Several tools have been proposed for detecting code smells [109, 117, 132, 152]. If a code smell detector has a low rate of false alarm and suggests automated refactorings that remove the code smells [47, 131], it may encourage programmers to use the refactoring tool more often. However, metrics for detecting code smells do not rival informed human intuition in practice [75, p. 75]. Perhaps we need systems that facilitate programmers’ collaboration on detecting code smells.

3.3 Disuse of Automated Refactoring Tools

In the human-automation interaction literature, disuse refers to underutilization of automation [131]. Disuse of automated refactorings occurs when a programmer performs a refactoring manually even though the IDE supports it.

Murphy-Hill et al. inspected a sample of the version control and refactoring histories of Eclipse developers and found that the developers had performed about 90% of their refactorings manually instead of using the refactoring tool [121].
Our interviews provided qualitative evidence for disuse of automated refactorings. For each of the following 15 refactorings, more than half of our interviewees sometimes performed the refactoring manually: Extract Method, Extract Class, Extract Super Class, Extract Interface, Extract Constant, Change Method Signature, Infer Generic Type Arguments, Generalize Declared Type, Use Supertype Where Possible, Encapsulate Field, Introduce Factory, Introduce Parameter, Move Instance Method, Move Static Member, and Pull Up.

In the rest of this section, we will discuss the factors that we have found to influence the disuse of automated refactorings.

### 3.3.1 Need

Some automated refactorings are underused just because programmers rarely need them. For instance, Table 3.1 shows that Extract and Pull Up are performed more than Inline and Push Down. Five of our interviewees told us that this was because they usually started with a simple design and gradually made it more general and reusable.

Two interviewees said that it was not worth learning some automated refactorings because they rarely performed the refactorings. For example, I₃ said:

> I know that there are many refactorings. But, many times I think that it's easier to just do something manually than try to learn a very particular refactoring that does something that I don't do very often.

### 3.3.2 Awareness

Programmers must be aware of an automated refactoring to use it. Prior survey studies have reported the role of awareness in the use of refactoring tools [121]. Our interviews showed that even experienced programmers do not know about many of the automated refactorings supported by Eclipse. We asked our interviewees the following three questions about each automated refactoring of Eclipse.

- Did you know that Eclipse supported this refactoring?
- Do you know what this automated refactoring does?
- Do you ever perform this refactoring manually? Why?
On average, our interviewees were unaware of the existence of more than nine automated refactorings of Eclipse. For each of the following refactorings, more than half of our interviewees did not know that Eclipse had automated support for the refactoring: Generalize Declared Type, Use Supertype Where Possible, Introduce Factory, Introduce Indirection, Introduce Parameter, Introduce Parameter Object, Move Type to New File, Move Instance Method, and Move Static Member.

We asked our participants how they learned the automated refactorings in Eclipse and why they knew only a subset of the refactorings in Eclipse. Our interviewees told us that they learned automated refactorings by seeing other programmers using them, reading articles, or exploring the IDE. Our findings corroborate the results of prior studies that identified peer interaction as a mechanism of discovering new tools [58, 120].

The interviewees did not always use all the automated refactorings that they knew about. For each one of Extract Method, Extract Class, Change Method Signature, Infer Generic Type Arguments, and Pull Up, at least five of the interviewees said that they sometimes performed the refactoring manually even though they were aware of its automated support in the IDE. In the rest of this section, we will discuss other reasons of disuse.

### 3.3.3 Naming

It has been assumed that recalling the names of automated refactorings is a barrier to using refactoring tools [114]. Our study provided more evidence that automated refactorings whose names are hard to understand, too technical, or confusing are more likely to get underused. I₄ told us:

> Generally, I don’t try them if I don’t know what they do. I might occasionally try them if I can kind of guess what they do even though I’m not sure, but I don’t do that very often.

The interviewees did not know the goals of more than eight automated refactorings on average. That is, our interviewees did not know what each of these automated refactorings did and were not able to correctly guess what the tool was supposed to do based on its name. For each of the following seven refactorings, more than half of the interviewees could not describe the transformation automated by the refactoring: Extract Class, Generalize Declared Type, Introduce Factory, Introduce Indirection, Introduce Parameter, Introduce Parameter Object, and Move Instance Method.
In particular, the majority of the interviewees confused the three automated refactorings: Infer Generic Type Arguments, Generalize Declared Type, and Use Supertype Where Possible.

### 3.3.4 Trust

Trust influences the use of automation, and reliability is a factor in the development of trust. If the automation is not reliable, the operators are more likely to lose their trust in the automation and stop using it, especially when the automation fails to perform simple tasks. However, if the automation is highly reliable, operators seem to tolerate its occasional failures and continue to use it [102, 106, 131].

We found usability to be a more important factor than reliability on programmers’ trust in a mature refactoring tool such as that of Eclipse. Even though others have found subtle errors in the refactoring tools of mainstream IDEs [61, 142], and there are many open issues about the refactoring tools in the bug tracking systems, none of the interviewees mentioned the existence of bugs in automated refactorings as a reason for not using these tools. Nonetheless, I₂ said that he would be more cautious while changing critical code:

> Most of the time, I don’t do [an automated] refactoring if it involves very critical codes. I’d rather do it manually. Only things that are so easy that they cannot possibly break, I would not expect them to break.

On the other hand, four of the interviewees did not use some of the automated refactorings because of their usability problems. I₃ said:

> There is also a notion of not trusting the [refactoring] tool. If the interface of the tool is not good enough, how do I know that the implementation is not sketchy?

The interviewees did not use automated refactorings that they had found to have complex user interfaces and unclear benefits. In general, if the benefits of automation are not readily apparent, humans are less likely to use the automation because of the cognitive overhead involved in evaluating and using the automation [131].

### 3.3.5 Predictability

We define the predictability of an automated refactoring as the programmer’s ability to tell how the tool is going to affect the code. We have found that the predictability of outcome is an important factor.
the use of automated refactorings. Three interviewees did not use some automated refactorings because of their unpredictability. For example, I_3 said:

[... ] If it affects only one file, then I kind of know exactly what the refactoring does and I can look at the result instantly afterwards. So, I don’t like refactorings that are ambiguous enough that I am not able to guess the final result. [...] If I cannot guess, I don’t use the refactoring. I consider it not worth the trouble. [...] If the thing that the [refactoring] tool does is so complicated that it isn’t easy to figure out things are alright, I’m kind of discouraged to use the tool.

In the following, we discuss how the complexity and preview of a refactoring affect its predictability.

3.3.5.1 Complexity

It was a challenge to determine the complexity levels of refactorings. Therefore, we used two approaches to estimate the complexities of refactorings. In the first approach, each of the first three authors individually assessed the complexities of manually performing the Eclipse refactorings in the IDE. Then, they compared their results and worked together to resolve the disagreements between assignments. Finally, they categorized the refactorings as simple, moderate, and complex. Table 3.1 illustrates the complexity level of each refactoring. We have found that our participants tended to perform simpler refactorings more frequently (Figure 3.3).

Our second approach for studying the effect of complexity on the use of automated refactorings was quantitative. We used the number of files and lines affected by an automated refactoring as an indicator...
of its complexity. Large refactorings can potentially alter many lines of code across many files. Such large refactorings are tedious and error-prone to do manually. Therefore, one might expect programmers to use automated refactorings for performing larger changes. However, our data show that 82% of the automated refactorings that the participants performed affected at most six lines and 84% of the performed automated refactorings affected only one file.

There could be various reasons for the low use of automated refactorings that make complex changes. First, there might just be less opportunity for performing large refactorings (Section 3.3.1). Second, we found that the current design of refactoring tools are not suitable for automating large refactorings. Two of our interviewees mentioned the problem with large refactorings. I1 said:

If it does too much, then it will overwhelm me. I will get too many changes at once. I don't like looking at diffs if I don't have to. And, if it does too much for me, I feel like I'm pushed out of the loop. Suddenly, it changes my program in a lot of ways. I will have to go and read these things while I prefer it to do a little for me.

Since the tools for small refactorings affect a narrow piece of the code, it is easier to understand the changes and verify their correctness. In contrast, programmers may worry that the tools for performing large changes may transform their code in unpredictable ways. Disuse of automated support for complex refactorings is consistent with findings from the human-automation interaction literature that imply humans prefer to take ownership of complex tasks and delegate simple ones to the machine [131]. More studies are required to understand the variables that affect the trend in programmers' use of complex refactorings.

3.3.5.2 Preview

The Eclipse automated refactorings allow the programmer to preview the changes before applying them. Quick Assist highlights the changes in its preview window (Figure 3.2), and refactoring wizards show the code before and after the change side by side. Others identified a usability problem of preview windows based on a survey [121]. CODINGSPECTATOR's data provides more evidence for the underutilization of preview windows. Our interviews and quantitative results (Table 3.1) show that our participants rarely previewed their automated refactorings. The participants previewed only 1% of the automated refactorings. We asked the five interviewees who had used Quick Assist whether they had previewed the
refactorings in the Quick Assist menu. All of them told us that they had not previewed the changes using Quick Assist. I$_5$ told us:

The scope of the preview is quite small and there is also no highlighting or indentation. So, if the code is a bit more complex, it can get quite difficult to understand.

I$_1$ gave the following reason for not previewing refactorings.

[... ] Quick Assist actions are very quick to execute and I can just look at them in the browser [...]. I admit I don’t really enjoy looking at diffs and I prefer to get a sense of the change (if it is local) by undoing/redoing, often several times.

The interviewees mentioned several reasons for not previewing refactorings. First, since they usually used the refactoring tool to perform small changes that were localized to a single method or class (Section 3.3.5.1), they did not need to preview the change. Second, one interviewee said that the preview window was not very useful because it always showed a small portion of the code. Third, the overhead of inspecting the large changes presented in the previews is high. Finally, two of the interviewees said that they could better review and evaluate their refactorings as they performed them manually. For instance, I$_6$ said:

Doing it [a refactoring] manually gives me a sense of how things have changed as a design review so that I can see the different options and reevaluate my choices.

3.3.6 Configuration

Eclipse lets the programmer perform slight variations of every refactoring by providing a few options. For example, the Extract Method refactoring in Eclipse 3.7 lets the programmer control the access modifier or declared exceptions of the extracted method.

CODINGSPECTATOR recorded the time of opening and closing every refactoring wizard (Section 3.1.2). We use the amount of time a refactoring wizard is open to estimate the time needed for configuring an automated refactoring. Based on the data, the participants configured the refactoring tool in at most eight seconds in 82% of the time.

Three of our interviewees complained about the complexity of refactoring wizards. I$_4$ said:
To me, whenever you go into these refactorings you have some dialogs and you have to figure out what it's doing and if there's one or two call sites, you can still do it simply manually I think.

Configuration dialogs break the programming workflow and impose an overhead by requiring the programmer to understand the options. More configuration options may make the automated refactoring more powerful but also more complex and harder to understand. Our results provide more evidence for the disruptiveness of refactoring tools that others identified in a survey [121].

3.3.7 Implications

Better training on refactorings and their tools may persuade programmers to use automated refactorings more. However, there are other obstacles to the adoption of existing automated refactorings. For instance, designers should choose more intuitive and consistent names for automated refactorings.

Tools that facilitate the exchange of knowledge between programmers can raise the awareness of refactoring tools. For example, a tool that uses social indicators to notify members of a software development team about the refactoring activities of other members might encourage programmers to learn more tools from each other [145].

A main motivation of automated refactorings is to reduce the human burden and error in making complex changes to the source code. Researchers have been proposing automated support for complex refactorings [66, 99, 143, 158, 167]. However, our results suggest that programmers are reluctant to use automated refactorings whose outcomes are difficult to foresee. One may expect previews to help programmers predict the results of automated refactorings. In contrast, we have shown that the current previews of refactorings are not effective. Perhaps more radical ways of reviewing refactorings are needed. The challenge would be to present the changes distributed across the code base in a concise and precise manner. An alternative way of reviewing the changes of refactorings is to provide facilities to inspect the changes after they are performed rather than before. One way to present the changes after they are performed is to mark up the changes in the editor. It might also be useful to help the programmer navigate through each part of the code that is affected by the refactoring tool. Alternatively, a graphical representation of a refactoring may be more effective for understanding the impact of the refactoring.
A high cost of configuration diminishes the value of automated refactorings. Therefore, the designers should make the configuration of refactorings seamless.

### 3.4 Misuse of Automated Refactoring Tools

Parasuraman and Riley defined misuse of automation as user's overreliance on automation. According to their definition, misuse of automation occurs when the user relies on the automation even though it would have been better to perform the task manually [131]. We sometimes found it challenging to judge whether a use of an automated refactoring was an overuse or clever use. Therefore, we qualify the definition of misuse to better explain the phenomenon in the context of refactoring tools. We define the misuse of an automated refactoring as use of the automated refactoring in ways not recommended by the designers.

Refactoring tools are designed to preserve the behavior of the program as much as possible except when certain features of the language such as reflection or native code are involved. The Eclipse refactoring tool checks a few preconditions to ensure that it will not introduce compilation problems or change the behavior of the program. If a precondition fails, the refactoring tool reports a problem with a severity level of *information*, *warning*, *error*, or *fatal error*. Warnings of automated refactorings attempt to predict compilation warnings. Errors of automated refactorings predict compilation errors and non-behavior-preserving changes. Thus, Eclipse does not recommend performing a refactoring with errors [21]. Fatal errors indicate that the refactoring tool is unable to carry out the transformation and prevent the programmer from continuing the refactoring. The rest of this section discusses some of the possible misuses of automated refactorings that we have identified.

#### 3.4.1 Unsafe Refactorings

When an automated refactoring reports a problem, it is no longer guaranteed to be behavior-preserving. Therefore, we call such a refactoring an *unsafe refactoring*. Traditionally, there has been an emphasis on the behavior-preservation property of refactorings [75, 127]. Our study provides the first quantitative and qualitative results about programmers’ use of unsafe refactorings.

A programmer can handle an unsafe refactoring in two ways. First, the programmer might cancel the refactoring, fix the code to satisfy the preconditions, and try the tool again. Second, the programmer
public int getNextNumber() {
    if (new Random().nextBoolean()) return 0;
    return 1;
}

Figure 3.4: If the Extract Method automated refactoring is invoked on the highlighted piece of code, the tool will report the error “Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed.”

can perform the refactoring and fix the problems afterwards. The former approach provides stronger behavior-preservation guarantees. However, we have found the latter to be the prevalent approach in dealing with unsafe refactoring. According to the data collected by CODINGSPECTATOR, the participants performed 79% of automated refactorings that had reported some error or warning. Table 3.1 illustrates the probability of our participants performing each kind of refactoring in spite of a reported problem.

Our participants received a total of 70 different messages from the Eclipse refactoring tool. The following are four of the 15 most frequent problems that the refactoring tool reported to the participants:

1. WARNING: Code modification may not be accurate as affected resource ‘resource name’ has compile errors.
2. ERROR: Found potential matches. Please review changes on the preview page.
3. ERROR: Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed.
4. WARNING: A variable with name ‘variable name’ is already defined in visible scope.

Thirteen participants received the first message for a total of 89 times and performed the refactoring 94% of the time. Twelve participants received the second message for a total of 31 times and performed the refactoring 77% of the time.

Figure 3.4 illustrates how to reproduce an error message that the Extract Method refactoring reported to four of the participants for a total of 13 times. In 54% of the cases, the participants chose to continue the refactoring and manually adjust the compilation problems of the resulting code.

Figure 3.5 illustrates an example where the Extract Local Variable refactoring warns the programmer about name shadowing. Two participants received this warning for a total of six times and performed the unsafe refactoring five times.
Figure 3.5: If the programmer uses the Extract Local Variable automated refactoring to extract the highlighted expression to a local variable named `i`, the tool will report the warning “A variable with name ‘i’ is already defined in visible scope.” If the user continues the refactoring, the compiler will not report any problems, but, the output of the program will change from 10 to 11.

One might argue that programmers perform unsafe refactoring because it is easier to interpret and resolve compilation problems than unfamiliar refactoring ones [113]. We asked the interviewees to explain the refactoring problems that they had received from the tool. The interviewees had understood almost all of the error messages of automated refactorings. Only one interviewee confused two error messages of the Extract Method refactoring, and at least two of our participants struggled with a strange error message of Extract Method. In every case, they were able to understand and resolve the problem eventually.

The interviewees mentioned that they relied on the compiler, visual inspection, and sometimes running their programs and tests to identify the possible problems of refactorings. However, visual inspection, compiler checks, runs of programs and tests, and code reviews may not catch the subtle errors of refactorings (Figure 3.5) [75, p. 391].

The interviewees gave us several reasons for performing unsafe refactorings. First, there is an overhead associated with canceling the refactoring tool and reconfiguring it. Second, the chance of introducing an error when the tool reports a warning is low. Third, our interviewees claimed that they were well aware of the limitations of the refactoring tool and could easily detect and fix the errors introduced by the tool. Fourth, our participants ignored non-descriptive messages. For example, none of our interviewees knew what “potential matches” meant in the aforementioned error message.

Five interviewees said that they sometimes manually performed a few steps of a refactoring and intentionally introduced some compilation problems to find all other places that needed to get updated.
Even though this way of performing a refactoring is slower than using the refactoring tool, it is more interactive and gives more control to the programmer.

3.4.2 Unjustified Uses

We suspect that at least two of our participants overused the refactoring tool, because they told us that they always used an automated refactoring if one was available for their desired task. However, it is not always optimal to use the automated refactorings. For example, one of our participants used the Change Method Signature refactoring to change the visibility of a method. Visibility changes could lead to subtle changes of a program’s behavior. Nevertheless, Eclipse does not currently perform the necessary checks to guarantee the behavior-preservation of such refactorings [151]. Thus, the use of the Eclipse refactoring tool to change the visibility of a method, especially in simple cases, is questionable. When we asked the participant about this particular use of the tool, he did not have any justifications.

A combination of excessive trust in the refactoring tool and low confidence in one’s coding abilities might lead to the misuse of the tool. For example, an interviewee told us that he always used the tool to perform refactorings, because he was afraid of making mistakes in performing the refactoring manually. While it is error-prone to perform some refactorings manually, programmers can perform the rest easily.

3.4.3 Implications

We suggest a few techniques for the designers of refactoring tools to avoid misuse.

A high rate of false alarms may lead to mistrust of the warnings [131]. Thus, reducing the number of false positives might mitigate the misuse of automated refactorings when they report warnings.

Although it is valuable to communicate the error messages better [115], our participants performed unsafe refactorings even though they had understood the messages. One way to mitigate the risk of unsafe refactorings is to provide specialized tools that verify the results and assist the programmers in completing the transformation.

Our participants’ use of unsafe refactorings and reliance on the incremental compiler to perform a refactoring in small steps suggest that predictability and interactivity may be more important factors in the design of refactoring tools than behavior-preservation. If automated refactorings present some of their intermediate results, become more interactive, and give more control to the programmer, they will
become more transparent and predictable. As a result, programmers will gain a better understanding of
the limitations of the automated refactorings and use them appropriately.

Another technique to reduce the misuse of automated refactorings is to make the tools more flexible. Such a flexible tool will attempt to change the code to satisfy the preconditions or propose possible fixes to the programmer instead of just reporting the problem [142].

Also, we suggest that trainers warn programmers about the possible excessive trust and misuses of refactoring tools. Trainers could make programmers aware of their excessive trust in certain automated refactorings and show them how to avoid or mitigate the consequences of their misuses.

3.5 Limitations

Even though a study such as ours captures authentic data, it raises privacy issues and makes recruitment challenging. As a result, the majority of our participants and all of our interviewees were at the university (Section 3.1). The confidentiality issues might have also affected the projects our participants have enabled our data collectors on.

Because of the uncontrolled nature of our study, the numbers of programming hours of our participants vary a lot (Section 3.1), and the number of opportunities to perform refactorings on the projects might have been different.

CODING SPECTATOR did not capture data about ten automated refactorings in Eclipse. We prioritized which automated refactorings to study based on the usage statistics reported by others [112, 121].

We fixed some of the bugs of our data collectors during the study and we discovered some bugs in Eclipse that may have affected our data. However, to the best of our knowledge, these bugs introduce negligible noise in our data.

We collected data only from Java programmers who used Eclipse. However, we were able to generalize and suggest improvements to other IDEs based on our results.
Chapter 4

Alternate Refactoring Paths Reveal Usability Problems

Researchers have evaluated refactoring tools with usability testing methodologies such as lab studies and interviews (Chapters 3, 5) [74, 77, 103, 116, 119, 121]. These methodologies find real usability problems by recruiting only a small number of participants. However, these methodologies are known to be suitable for identifying only certain usability problems, e.g., those that can be exposed during a short lab study or the ones that a programmer can remember during an interview. This is because such methodologies can evaluate the tool only for a short period of time, limited kinds of tasks, and a small number of participants.

Humans learn from their past mistakes. Certain events severely affect our lives, e.g., accidents and injuries. Reflecting on such events, we try to improve our future strategies. If refactoring tools were human beings, how would they have learned from their past experiences?

The critical incident technique (CIT) [71, 146] is a general methodology for revealing problems by analyzing critical incidents. A critical incident is a breakdown of a user's interaction with the system that seriously affects the user's task. The Human-Computer Interaction (HCI) community has found that CIT is an effective, complementary methodology for discovering usability problems [35, 86, 87]. However, the notion of “critical incidents” is not well-understood or studied for interactive program transformation (IPT) tools, e.g., refactoring tools. This leaves three research questions open: What are the critical incidents for IPT tools? Are these incidents indicators of the usability problems of IPT tools? How can evaluators infer usability problems from critical incidents?

Our work bridges the gap between two lines of research (program transformation and CIT) by adapting CIT to refactoring tools. We show that alternate refactoring paths are indicators of usability problems. An alternate (refactoring) path is a sequence of user interactions with the refactoring tool that differs from the primary path. The primary path, also known as the happy path, is an ideal sequence of interactions that leads to a successful application of the automated refactoring. Refactoring tool users
(a) This selection results in an invocation of Move Static Members.

(b) This selection results in an invocation of Move Compilation Unit.

Figure 4.1: Selections that look equivalent to a programmer may be interpreted differently by the Eclipse refactoring tool. The Eclipse refactoring tool detects the type of the Move refactoring based on the code selection. The only difference between these selections is the two leading spaces on line 2 of Figure 4.1b. Although the code selections shown in this figure are very similar, they result in invocations of different refactorings: Move Static Members and Move Compilation Unit.

We hypothesize that events such as cancellations, repeated invocations, and reported messages of a refactoring tool, which correspond to alternate paths, are likely to indicate usability problems. To test this hypothesis, we analyzed the refactoring usage data that we collected during a field study (Chapter 3). We conducted the field study for three months and collected usage data for about a total of 2,320 programming hours of 36 programmers. By analyzing 145 alternate refactoring paths in this data set, we found 15 usability problems [1–15], 13 of which were previously unknown. We both reported the problems and proposed design improvements to the developers of Eclipse. Consequently, the developers acknowledged all the problems we reported and have already fixed four of them. These results suggest that analyzing alternate refactoring paths is effective at identifying the usability problems of refactoring tools.

The analysis of alternate refactoring paths revealed a variety of usability problems (Section 4.3). For example, Figure 4.1 shows how a minor change to a code selection results in the invocation of an unexpected Move refactoring. Reporting this problem [7] to the Eclipse developers led to the discovery of broader consequences of the problem (Section 4.3.6.3).
There are several advantages to our adaptation of CIT for finding the usability problems of IPT tools. First, our automatic data collection method can scale to many participants and collect data for a long time. This makes it possible to find usability problems from many user interaction paths. Second, our automatic data collectors are unobtrusive. This allows the evaluators to find usability problems without interfering with programmers’ work. Third, the large set of collected data can be mined to measure the frequency of usability problems empirically. Finally, analyzing the alternate paths reveals not only usability problems, but also design improvements.

In summary, this work contributes to the field of refactoring in several ways:

• We adapt (Section 4.2) and evaluate (Section 4.3) CIT for finding the usability problems of refactoring tools. This adaptation can be used to find the usability problems of other refactoring or IPT tools.

• We find real usability problems (Section 4.3) of a refactoring tool and propose design improvements to address them. These problems are encountered by programmers in the field and confirmed by the developers of the refactoring tool.

• We provide empirical evidence for the frequency of the usability problems revealed by our method.

4.1 The Origins of the Critical Incident Technique

The critical incident technique (CIT) is a general technique developed in its current form by Flanagan and published in Psychological Bulletin in 1954 [71]. The technique is believed to have been founded even earlier by Galton (circa 1930).

Flanagan defined CIT as a set of procedures for collecting and analyzing human behaviors that have critical significance (positive or negative). Typically, the respondents are asked to describe their significant experiences. Variations of this method have been widely used in Human Factors [146]. Despite all the variations of CIT, a common definition of the critical incident still holds. A critical incident is an event during a task that is a significant indicator of some aspect of the objective of the study.

Flanagan describes CIT as an outgrowth of the studies conducted as part of the Aviation Psychology Program of the United States Army Air Forces in World War II. One of the early studies in this program that employed CIT was analyzing the reasons of disorientation while flying. In this study, the pilots
were asked to think of occasions during their flights that they experienced disorientation, i.e., they felt uncertain about their spatial position. Then, they were asked to describe what they saw, heard, or felt that caused that experience. This study resulted in a number of recommendations for changing the cockpit design and training pilots to avoid disorientation.

del Galdo et al. adapted CIT to HCI in 1986 [62]. They conducted a study to evaluate the documentation of a conferencing system. The participants were asked to perform a task using the system. They were also asked to report any incident (success or failure) that they encountered while using the documentation. The experimenters observed the participants during the study. This variation of CIT in which the participants report the incidents as they are encountered is called the user-reported critical incident technique (UCIT). del Galdo et al. made recommendations for improving the documentation based on the reported incidents.

Hartson et al. adapted UCIT to remote usability evaluation, where the users and evaluators are in different physical locations [86, 87]. In this variant of CIT, the experimenters train the users to identify and report critical incidents. The users report the critical incidents as they are encountered during the task. Later, the evaluators analyze the reports and the accompanied contextual information (e.g., video recordings) of the incidents. Hartson et al. showed the effectiveness of CIT through several studies on web applications. They also reported a problem with UCIT—users tend to delay reporting the incidents. While Hartson et al. used CIT to enable remote usability evaluation, their studies were mostly in the lab.

4.2 Methodology

The critical incident technique (CIT) consists of two major phases: data collection and data analysis. The rest of this section describes how we adapted each of these phases for identifying the usability problems of IPT tools.

4.2.1 Data Collection

Evaluators can collect the data required by CIT in different ways. For instance, the evaluators can conduct surveys or interviews to ask the participants about the critical incidents that they have recently experienced. As another example, the evaluators can observe the participants while performing a task
and take note of the critical incidents. Alternatively, the evaluators can ask the participants to report the critical incidents as the incidents occur during a task.

These data collection techniques are not scalable to many users, are based on artificial tasks, or interfere with users’ work. So, we made our data collection automatic to collect a large set of data that covers many usage scenarios of the refactoring tool in a form that is amenable to automatic data analysis. We made the data collection unobtrusive to avoid altering programmers' behavior. Finally, instead of collecting the data from predefined tasks performed at the lab, we decided to collect the data from real tasks that are more representative of how the refactoring tool is used in practice.

We conducted a field study on 36 programmers for three months, collecting usage data for a total of 2,320 programming hours.

4.2.1.1 Participants

We recruited participants from the industry (N = 17) and academia (N = 19). We advertised the study to the open-source community via mailing lists, IRC, and e-mail. In addition, we invited researchers from six research labs at the computer science department of the University of Illinois at Urbana-Champaign. We did not remunerate the participants. Instead, we explained to them the potential contributions of our study. We asked the participants to fill out a demographic survey. Based on the survey results, 26 participants had at least five years of programming experience (Table 4.1). Moreover, the participants indicated that they worked on a variety of domains such as banking, database management systems, business process management, and marketing.

4.2.1.2 Automatic Data Collector

We developed CODINGSPECTATOR (Chapter 2), an unobtrusive tool for collecting the usage data of the Eclipse refactoring tool. The only interaction that the participants had with CODINGSPECTATOR was to

<table>
<thead>
<tr>
<th>Years</th>
<th>1–2</th>
<th>2–5</th>
<th>5–10</th>
<th>&gt; 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participants</td>
<td>2</td>
<td>6</td>
<td>17</td>
<td>9</td>
</tr>
</tbody>
</table>
install it like any Eclipse plug-in, and enter their username and password when prompted to submit their data to our central repository. We chose to make the data collection process unobtrusive to study software evolution practices in the wild (Chapter 3).

**CODING SPECTATOR** captures more data about the usage of the refactoring tool than what Eclipse already does. It captures invocations of 23 automated refactorings of the 33 supported by Eclipse. The Eclipse refactoring history captures the following information about only the automated refactorings that are *performed*:

- **timestamp** the invocation time of the automated refactoring
- **kind** the kind of the automated refactoring (e.g., Rename Local Variable, Extract Method)
- **selection offsets** the start and length of the code selection used to invoke the refactoring
- **configuration options** information about how the programmer configured the automated refactoring (e.g., the name and accessibility of the new method created by Extract Method)

Eclipse records data only about the *primary* paths that users take while using the refactoring tool. In other words, it only records the eventually performed automated refactorings. However, troubling interactions may make the user take *alternate* refactoring paths by *canceling* or *undoing* a refactoring that has reported a *message* or is hard to use.

Since our goal was to adapt and evaluate CIT for refactoring tools, we made **CODING SPECTATOR** augment the Eclipse refactoring history in two ways. First, **CODING SPECTATOR** records *canceled*, *undone*, and *redone* automated refactorings in addition to the *performed* ones along with any *messages* reported by the refactoring tool. Second, it captures contextual information about refactoring activities that an evaluator can use to derive usability problems. The contextual information consists of the following:

- **navigation history** a description of how and when the user navigated through the refactoring wizard
- **invocation method** whether the user has invoked the refactoring tool through the wizard
- **selection** the piece of code selected to invoke the refactoring, and a larger slice of code surrounding the selection

---

1We made **CODING SPECTATOR** capture this information because the selection offsets captured by Eclipse do not always reflect exactly the ones used by the programmer due to some normalization that Eclipse applies on the selections.
messages all problems that the refactoring tool has reported to the programmer (Section 4.2.1.3)

Since Eclipse does not provide a reusable API to capture the above information, we instrumented the Eclipse source code. During the installation process, CODINGSPECTATOR replaces several existing Eclipse plug-ins, such as LTK and JDT, by our instrumented versions of these plug-ins.

4.2.1.3 Refactoring Messages

A refactoring precondition is a property that the refactoring tool checks at various stages, e.g., selection, invocation, configuration, and commit, to guarantee that the change will preserve the behavior of the program. If a precondition fails, the refactoring reports a message whose type depends on the severity of the problem and the stage of refactoring. We refer to such a message as a refactoring message or just a message in this chapter.

The Eclipse refactoring tool may report any of about 640 messages of four types to its user [21]:

UNAVAILABLE The refactoring tool refuses to open the refactoring wizard due to the failure of a precondition.

WARNING This kind of message attempts to predict compilation problems and can often be ignored safely.

ERROR The Eclipse documentation recommends not to continue an automated refactoring that has reported a message of this kind, because it is very likely to break the code.

FATAL ERROR The refactoring tool refuses to perform the change on the code.

The refactoring tool may report an UNAVAILABLE message only before the refactoring wizard is open, while it may report the other types of messages only during the later stages of refactoring.

4.2.2 Data Analysis

The data analysis phase of CIT consists of two major phases: identifying critical incidents and inferring usability problems. We automated the identification of critical incidents and their accompanying contextual information. However, inferring the usability problems from critical incidents is a manual
process that an evaluator can do. The rest of this section describes how we adapted these two phases of CIT to IPT tools.

4.2.2.1 Identifying Critical Incidents

Collecting the critical incidents. We first collected a set \( I \) of critical incidents, i.e., refactoring invocations that reported a message (Section 4.2.1.3) as well as canceled, undone, and redone refactorings. Then, we add to \( I \) any refactoring that occurred within five minutes of a critical incident in \( I \).

Finding the most frequent refactoring messages. We computed the frequency of each refactoring message by counting the number of times it occurred in \( I \). We consider the frequency of a refactoring message a measure of the criticality of the message as an incident. We have released the data about the frequencies of the refactoring messages [30].

The number of all messages that the Eclipse refactoring tool may report is large (\( N = 640 \)). It is tedious to investigate all possible messages for usability problems. Besides, it is often impossible to infer a usability problem from a message without having other contextual information about the refactoring invocation. CODINGSPECTATOR’s data indicates that the Eclipse refactoring tool reported only 83 different kinds of messages to the participants during the study. Therefore, we focus on these messages that the refactoring tool reported in practice. In addition, we analyze the most frequent messages in the contexts that they appeared. The contextual information captured by CODINGSPECTATOR allows us to identify the conditions under which the message is reported and how the programmers react to the message.

Extracting refactoring batches. The refactoring batch of a critical incident is a subset of \( I \) containing the events that are semantically related to the critical incident. Examples of semantically related events are cancellations and invocations of the same refactoring or refactorings on related program entities. A refactoring batch provides the context necessary for inferring usability problems from a critical incident. We extracted the refactoring batches of the most frequent refactoring messages. To extract the refactoring batch of a critical incident, we manually inspected the events in \( I \) that occurred within 20 minutes of the critical incident and extracted those that were semantically related to the critical incident. We found that a window of 20 minutes was large enough to cover all events that were semantically related to a critical incident. For each event in the refactoring batch of a critical incident, we took a note of how the
event was related to the critical incident. We referred to these notes while inferring usability problems from refactoring batches.

### 4.2.2.2 Inferring Usability Problems

**Analyzing the refactoring batches.** We manually inspected 145 refactoring batches of the most frequent refactoring messages to infer usability problems. We used the contextual information in each refactoring batch (e.g., code snippet, selection, and invocation method) to reproduce the behavior of the refactoring tool. For each refactoring batch, we examined programmers' reactions to the reported messages. If programmers dismissed the message, we checked if the message was poor, e.g., vague, difficult to understand, or uninformative. We evaluated the effectiveness of the message ourselves based on general usability guidelines, and we did not contact the participants. If the programmer canceled the refactoring, we checked if the refactoring was later repeated. If the refactoring was repeated, we checked for the possible changes to the configurations of the refactoring or any manual changes made to the code since the previous invocation. Such changes often revealed why the refactoring failed in the first attempt and how the refactoring tool could be improved to avoid the failure. If the refactoring was repeated multiple times unsuccessfully, we considered it a stronger indication of a usability problem.

**Confirming the usability problems.** Finally, we reported the usability problems that we inferred to Eclipse developers to confirm that they are indeed considered usability problems from the Eclipse developers' points of view. We included in our reports some empirical data about each usability problem, e.g., the number of refactoring batches with the same usability problem, the number of times programmers canceled or performed a refactoring as well as a summary of the strategies that the programmers employed to remedy the usability problem. We also made our reports actionable by making concrete suggestions on how to resolve the usability problems.

### 4.3 Usability Problems

We were able to infer usability problems by analyzing alternate refactoring paths. This shows that alternate refactoring paths are indicators of usability problems. This section presents some of the usability problems that we identified. For each usability problem, we present its frequency in our data
Table 4.2: The frequency of each kind of automated refactoring and refactoring message in the CODINGSPECTATOR data set.

<table>
<thead>
<tr>
<th>Event</th>
<th>Occ.</th>
<th>Performed</th>
<th>Canceled</th>
<th>Undone</th>
<th>Redone</th>
<th>WARNING</th>
<th>ERROR</th>
<th>FATAL</th>
<th>UNAVAILABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4,245</td>
<td>267</td>
<td>284</td>
<td>34</td>
<td>191</td>
<td>124</td>
<td>49</td>
<td>85</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Meanings of the symbols used to describe an alternate refactoring path quantitatively. Occ. can be greater than Per. + Can., because a refactoring with an UNAVAILABLE message is not counted as either performed or canceled.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occ.</td>
<td>the number of occurrences of the alternate path</td>
</tr>
<tr>
<td>Par.</td>
<td>the number of participants affected by the alternate path</td>
</tr>
<tr>
<td>Bat.</td>
<td>the number of refactoring batches containing the alternate path</td>
</tr>
<tr>
<td>Per.</td>
<td>the number of instances of the alternate path that the participants performed</td>
</tr>
<tr>
<td>Can.</td>
<td>the number of instances of the alternate path that the participants canceled</td>
</tr>
<tr>
<td>Rank</td>
<td>the index of a message in an array of all messages sorted by Occ. descendingly</td>
</tr>
</tbody>
</table>

set, how we identified the usability problem, and what suggestion we made to Eclipse developers to resolve the usability problem.

CODINGSPECTATOR recorded detailed information for 92% (4,245 of 4,611) of the refactorings that were performed and recorded by Eclipse. Table 4.2 illustrates the frequencies of automated refactoring events. Table 4.3 introduces several symbols that we use in the rest of the paper. Table 4.4 lists the most frequent messages that the Eclipse refactoring tool reported to our participants.

Due to privacy and confidentiality constraints of the field study, we cannot present the participants’ code. Instead, we demonstrate the usability problems using simplified versions of the participants’ pieces of code.

### 4.3.1 Vague Messages

We analyzed the refactoring batches of the two most frequent messages of the Eclipse refactoring tool (rank one and two in Table 4.4). When we reproduced these messages, we noticed that they are vague. That is, they do not clearly explain the problem, leaving the programmer confused about the risks of performing the refactoring and the actions required to mitigate the risks. For example, for the
Table 4.4: A list of the most frequent messages reported by the Eclipse refactoring tool. Column “Refactorings” lists the refactorings that reported each message. Refer to Table 4.3 for the meanings of the other column headers of this table.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>All</td>
<td>Rename Compilation Unit, Type, Enum Constant, Field, Method, Package</td>
<td>4200</td>
<td>29</td>
<td>4032</td>
<td>168</td>
</tr>
<tr>
<td>1</td>
<td>Code modification may not be accurate as affected resource * has compile errors</td>
<td>WARNING</td>
<td>Rename Compilation Unit, Type, Enum Constant, Field, Method, Package</td>
<td>113</td>
<td>17</td>
<td>108</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Found potential matches. Please review changes on the preview page.</td>
<td>ERROR</td>
<td>Change Method Signature, Move</td>
<td>44</td>
<td>14</td>
<td>35</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Type * contains a main method—some applications (such as scripts) may not work after refactoring.</td>
<td>WARNING</td>
<td>Rename Compilation Unit, Type, Package</td>
<td>41</td>
<td>7</td>
<td>39</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>This refactoring cannot be performed correctly due to syntax errors in the compilation unit. To perform this operation you will need to fix the errors.</td>
<td>FATAL ERROR</td>
<td>Rename Field, Method</td>
<td>22</td>
<td>10</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Ambiguous return value: Selected block contains more than one assignment to local variables.</td>
<td>UNAVAILABLE</td>
<td>Extract Method</td>
<td>22</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>This name is discouraged. According to convention, names of local variables should start with a lowercase letter.</td>
<td>WARNING</td>
<td>Change Method Signature, Extract Local Variable, Rename Local Variable</td>
<td>15</td>
<td>5</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed</td>
<td>ERROR</td>
<td>Extract Method</td>
<td>15</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

most frequent message (rank one in Table 4.4), neither the message nor any other information on the refactoring wizard indicates what part of the code modification may not be accurate. This Warning is too broad to be of any actionable use. Perhaps this is why the participants continued 96% (108 of 113) of the refactorings despite reporting this Warning. We reported several instances of this usability problem to the Eclipse developers [1, 11–13].

4.3.2 Overly Strong Preconditions

Precondition checking of refactoring tools is a delicate process. On one hand, the preconditions should be strong enough to prevent the refactoring from breaking the code or altering its behavior in unintended ways. On the other hand, the preconditions should not be overly strong and reject safe refactorings. Our prior study showed that programmers prefer flexible refactorings, often ignore precondition failures, and
manually fix the problems afterwards (Section 3.4.1). The following discusses two usability problems [2, 10] related to overly strong preconditions that we identified.

Realizing the disadvantages of overly strong preconditions, others have proposed a bounded-exhaustive testing approach [148]. Our approach of finding overly strong preconditions by analyzing critical incidents complements theirs in two ways. First, we find overly strong preconditions that programmers have encountered in real-world, while their approach finds overly strong preconditions in a large number of automatically generated programs. Second, their approach does not currently support refactorings below the method level, e.g., Extract Method.

4.3.2.1 Ambiguous Return Value

The Extract Method refactoring reported the following UNAVAILABLE message to our participants (Occ. = 22, Par. = 6).

Ambiguous return value: Selected block contains more than one assignment to local variables. Affected variables are: [...]  

We first reproduced the message using the information captured in its refactoring batches to understand the conditions under which the refactoring tool reports this message. These batches contained an invocation of Extract Method with a code selection similar to the one shown in Figure 4.3. The Eclipse refactoring tool refuses to continue with such a code selection and reports the message in Figure 4.2. By studying the message itself and reproducing the repeated refactorings in the refactoring batches of the message, we found why the refactoring tool refuses to continue: the selected code assigns to two local variables (a and b) that are used outside the selection (line 6). Since a Java method can return at most one value, the refactoring tool cannot infer the return value of the extracted method (a or b). There are several strategies for handling this message:

1. Abandon the refactoring.

2. Remove the uses of the local variables outside the selection and invoke the refactoring tool on the same code selection again.

3. Narrow the selection and invoke the refactoring again.
Figure 4.2: The Eclipse Extract Method refactoring reports the message “Ambiguous return value” for the code selection shown in Figure 4.3.

Figure 4.3: The Eclipse Extract Method refactoring reports the message “Ambiguous return value” (Figure 4.2) for the code selection shown in this figure.

4. Widen the selection and invoke the refactoring again.

5. Convert some of the local variables into fields and invoke the refactoring again.

6. Create a new class with fields corresponding to the local variables that are assigned to in the selected piece of code. Have the extracted method store the local variables in the fields of an instance of this class and return it.

The Eclipse documentation advises the third and fourth solutions [22]. The refactoring batches showed that our participants followed the first, second, and third strategies. That is, they either canceled the refactoring or repeated it with a different code selection. These refactoring batches contain all three types of critical incidents: cancellation, reported message, and repeated invocations. We identified two usability problems by analyzing these alternate refactoring paths.

First, while trying to reproduce the message, we found that the message is not descriptive enough. The message indicates that the refactoring tool is unable to proceed because the selected piece of code
assigns two local variables. However, this is not a sufficient condition. The refactoring tool refuses to continue only when more than one local variables are both assigned in the selected piece of code and used outside the selection. We suggested that the Eclipse developers make the message more descriptive. The Eclipse developers fixed this problem [1].

Second, this precondition check of Eclipse is overly strong. Refusing to continue the refactoring caused additional overhead to our participants as shown by their efforts to repeat the refactoring in the refactoring batches. We examined the cancellations and repeated invocations in refactoring batches to understand how programmers handled this message. We found that programmers used three solutions. They either narrowed the selection, widened it, or converted the local variables to fields and repeated the refactoring. Based on this observation, we suggest that the refactoring tool be more flexible and let the programmer continue the refactoring in the following ways. One option is to warn the programmers about the use of the local variables outside the selection but still allow them to continue the refactoring and make the new method have no return value. Alternatively, the tool could suggest that it automatically converts the local variables to fields and proceed. We reported this overly strong precondition to Eclipse developers. They acknowledged the value of flexibility saying [2]:

I have had instances when I have had to perform a refactoring manually because Eclipse would not proceed because of an error. On such occasions I do wish for things to be a bit more flexible.

4.3.2.2 Missing Return

The Eclipse refactoring tool reported the following Error to the participants (Occ. = 15, Par. = 6, Bat. = 11, Per. = 8, Can. = 7):

Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed

This Error message was the second most frequent message that the Eclipse Extract Method refactoring reported to the participants (Table 4.4). Seven of the 11 refactoring batches that contained this message contained at least one cancellation and one repeated invocation. In one refactoring batch, the participant invoked the refactoring four times each time changing a configuration option, selection, or code. These alternate paths indicate the difficulty of using the tool and the possibility of usability problems.
Table 4.5: The most frequent refactoring messages that were due to name conflicts. Column “Refactorings” lists the refactorings that reported each message. Refer to Table 4.3 for meanings of the other column headers of this table.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Message</th>
<th>Type</th>
<th>Refactorings</th>
<th>Occ.</th>
<th>Par.</th>
<th>Per.</th>
<th>Can.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Duplicate local variable</td>
<td>ERROR</td>
<td>Extract Local Variable, Rename Local Variable</td>
<td>14</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>A variable with name * is already defined in the visible scope</td>
<td>WARNING</td>
<td>Extract Local Variable</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>Package * already exists in this project in folder</td>
<td>WARNING</td>
<td>Rename Package</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>Type named * already exists in package</td>
<td>ERROR</td>
<td>Rename Compilation Unit, Type</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>Compilation unit * already exists</td>
<td>FATAL</td>
<td>Rename Compilation Unit, Type</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

By reproducing the message, we found the underlying reason of the message. Eclipse cannot extract a set of statements with some control flow paths ending in `return` and some others not (Figure 4.4). We examined the code selections in the refactoring batches of this message to see if this precondition check can be relaxed for certain common code selections. As a result, we found that the refactoring could have been more flexible and infer the missing `return` statements (Figure 4.4c) in seven of the 11 batches that contained this message. The IntelliJ refactoring can infer the missing `return` statement in some cases. We suggested this enhancement to Eclipse developers and they acknowledged its benefits [10].

4.3.3 Name Conflicts

Refactorings such as Rename and Extract Local Variable, which change the name of an existing program element or introduce a new named program element, may cause name conflicts. Table 4.5 lists the most frequent refactoring messages that Eclipse reported to the participants due to name conflicts (Occ. = 43, Par. = 14, Bat. = 36, Per. = 20, Can. = 23).

Our analysis of the refactoring batches showed that the participants either continued the refactoring despite the message and resolved the name conflict manually or canceled the refactoring and invoked it again to enter a different name. Having the programmers navigate back to the configuration page or repeat the invocation of the refactoring with a different name to refactor safely is an additional overhead, which we consider a usability problem.
Rename Package warns the user about conflicting names immediately on the first input page. However, other refactorings including Extract Local Variable, Rename Local Variable report the conflicting name later, requiring the user to navigate back to correct the name. We reported this source of additional overhead to Eclipse developers and suggested that the refactoring tool informs programmers about potential name conflicts earlier [9, 15].

4.3.4 Unintended Invocations

Our analysis of the refactoring batches revealed that Eclipse refactorings are prone to unintended invocations. That is, programmers sometimes invoke a refactoring other than the one they intend. The participants invoked Extract Local Variable while selecting the name of a variable in its declaration (Occ. = 10, Par. = 4). The refactoring tool reported an error saying that Extract Local Variable is not applicable to the selected code (Figure 4.5):

`An expression has to be selected to activate this refactoring. Names used in declarations are not expressions.`

Eventually, the programmers applied either Rename Local Variable or Convert Local Variable to Field on the same selection. These alternate refactoring paths suggest that the programmer intended to invoke a refactoring other than Extract Local Variable. The usability problem that we inferred from these alternate paths is that Eclipse refactorings are prone to unintended invocations.

One way to avoid unintended invocations of automated refactorings is to enable only the ones that are applicable to the selected piece of code. However, the refactoring tool should be ready to explain to the programmer why a certain refactoring is disabled.

4.3.5 Unintuitive Configuration Options

The goal of the Move Instance Method refactoring is to move the declaration of an instance method from its enclosing class to another class. Figure 4.6 shows the effect of applying Move Instance Method on an example piece of code.

Six of the participants invoked the Move Instance Method refactoring for a total of 16 times as parts of ten batches. However, none of the invocations were applied. Either the programmer canceled the
refactoring (11 times), or the refactoring tool refused to continue and reported an UNAVAILABLE message (5 times). Two refactoring batches indicated that the participants invoked the refactoring tool three times but did not succeed to perform the Move Instance Method refactoring. These critical incidents led us to identify two usability problems.

First, the configuration dialog provides options that programmers cannot easily interpret. The dialog asks for a pair of “name” and “type” (Figure 4.7), while the programmer would like to select the destination class. A refactoring batch indicated that a participant spent 27 seconds on the configuration dialog of Move Instance Method, which is higher than the average time our participants spent on this dialog (16.5 seconds). We asked the participant why he spent this time on the configuration dialog and eventually canceled the refactoring. The participant said that he expected the refactoring tool to ask him about the destination class not a pair of “name” and “type”. He canceled the refactoring because he could not interpret the required options. However, selecting a destination class is not sufficient in general, because the refactoring tool has to update the call sites as well. The refactoring asks the programmer to select a variable to determine the new receivers of the call sites. For the example shown in Figure 4.6, the refactoring tool changes the call `c.m(e1, e2)` (line 12, Figure 4.6a) to `e1.m(c, e2)` (line 8, Figure 4.6b). Nonetheless, since the configuration dialog does not communicate the necessity of these options well, the programmer gets confused.

Second, the configuration dialog requires more options than what is necessary for moving certain kinds of methods. We found that fewer, simpler options were sufficient to support a common class of attempted refactorings captured in refactoring batches. We say an instance method is effectively static if it can be made static without introducing any compilation problems. For example, in Figure 4.6, method `C.m2()` (lines 8–13, Figure 4.6a) is effectively static. However, method `C.m(E, E)` (lines 4–7, Figure 4.6a) is not because it depends on the instance field `C.d1`. In five batches, the participants tried to move effectively static methods. If the method is effectively static, it would be sufficient for the dialog to ask the destination class from the programmer. However, the Eclipse refactoring tool always requires the programmer to select a field or parameter as the new target of the method. This design is restrictive because it does not allow the programmer to move an effectively static method to a class other than those reachable from a field or parameter.
Table 4.6: The most frequent refactoring messages that are due to invalid code selections. Column “Refactorings” lists the refactorings that reported each message. Refer to Table 4.3 for the meanings of the other column headers of this table.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>The end of the selection contains characters that do not belong to a statement</td>
<td>UNAVAILABLE</td>
<td>Extract Method</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>An expression must be selected to activate this refactoring.</td>
<td>UNAVAILABLE</td>
<td>Extract Local Variable, Constant</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>Cannot extract the left-hand side of an assignment.</td>
<td>UNAVAILABLE</td>
<td>Extract Method</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We reported both of the above usability problems to Eclipse developers [3, 4]. The developers acknowledged these problems and made an improvement accordingly. To address the first usability problem, we suggested that the configuration dialog asks the destination class and new receiver separately and clarify why it requires a new receiver. We provided the developers with prototypes of alternative designs of the configuration dialog. To resolve the second usability problem, we suggested that the refactoring tool automatically detects effectively static methods and allows the programmer to move them to any writable class without requiring a new receiver.

4.3.6 Invalid Code Selections

Programmers have to select pieces of code to invoke most automated refactoring. For example, programmers have to select a set of statements to apply the Extract Method refactoring on them. However, selecting a valid piece of code can be error-prone, especially when the selection is long. Table 4.6 lists the most frequent refactoring messages reported because of invalid code selections. In total, 30 refactoring messages were reported due to invalid code selections. We identified several usability problems [5–8, 14] related to invalid selections.

4.3.6.1 Trailing Semicolon

The inclusion or exclusion of a semicolon can make a selection invalid for Extract Method (Figure 4.8). Figure 4.8a illustrates a selection that causes Extract Method to report the following UNAVAILABLE message:

```
Cannot extract the left-hand side of an assignment.
```
The Extract Method refactoring expects the trailing semicolon to be included in the selection because it expects a set of statements not expressions as its input. The participants that received this message \((Oc. = 5, Par. = 2, Bat. = 3)\) eventually extended the selection to include the trailing semicolon and performed the refactoring successfully. Nonetheless, in two batches, the participants repeated the refactoring with the incorrect selection until they noticed the trailing semicolon. These repeated invocations indicate the subtlety of this message. We examined the refactoring batches of this message to see if the refactoring tool could be improved to avoid the alternate refactoring paths. We found that if the refactoring tool had automatically expanded such selections to include the trailing semicolon, the cancellations and repeated invocations of all refactoring batches of the message would have been avoided. We reported a usability problem to Eclipse developers \([5]\) suggesting that the refactoring tool automatically expands such selections. Eclipse developers fixed this problem.

Figure 4.8b shows a selection that causes Extract Method to report the following UNAVAILABLE message:

The end of the selection contains characters that do not belong to a statement.

The participants that received the above message \((Oc. = 5, Par. = 3, Bat. = 3)\) eventually shrank the selection to exclude the trailing semicolon and performed the refactoring successfully. Nevertheless, a participant repeated the invocation of the refactoring two times before noticing the trailing semicolon. We identified a usability problem based on an analysis of these alternate refactoring paths. The usability problem was already reported and fixed by the Eclipse developers \([6]\).

### 4.3.6.2 Unbalanced Braces

The Extract Method refactoring takes a set of statements as its input. However, selecting a well-formed set of statements can be error-prone, especially when the selection is long. The Extract Method refactoring reports one of the following UNAVAILABLE messages when the selection is not well-formed due to its unbalanced braces (Figure 4.9):

The selection does not cover a set of statements or an expression. Extend selection to a valid range using the ‘Expand Selection To’ actions from the ‘Edit’ menu.

The beginning of the selection contains characters that do not belong to a statement.
The participants made selections that were ill-formed due to unbalanced braces \((Occ. = 4, Par. = 3, Bat. = 3)\). In each batch, the participants changed the selection and repeated the refactoring at least twice until they finally managed to perform the refactoring successfully. These critical incidents reveal two usability problems. First, selecting a well-formed set of statements is error-prone. This finding supports the need for tools that assist programmers in making well-formed selections like the ‘Expand Selection To’ actions of Eclipse or Selection Assist \([116]\). Second, the latter UNAVAILABLE message mentioned above is misleading because it gives a wrong cue for the cause of the problem. The Eclipse developers acknowledged this usability problem \([8]\).

### 4.3.6.3 Equivalent Selections

We inferred a usability problem of the Move refactoring by analyzing the refactoring batches containing the following FATAL ERROR message \((Occ. = 4, Par. = 3, Bat. = 4, Per. = 0, Can. = 4)\):

> A file or folder cannot be moved to its own parent.

The Move Compilation Unit refactoring reports the above message when a compilation unit is about to be moved to its enclosing package—the default destination.

The configuration options that CODINGSPECTATOR captured for these refactorings indicated that in three of the batches the participants selected fields or methods and invoked the Move refactoring but the tool incorrectly interpreted the refactoring as Move Compilation Unit. We reproduced the attempted refactorings on pieces of code similar to the ones recorded by CODINGSPECTATOR. Surprisingly, we found that slight changes to the code selection result in the invocation of an unexpected kind of Move refactoring.

Refactoring tools infer the program elements under refactoring from the selected piece of code. For instance, if a programmer selects a method name and invokes the Rename refactoring, the Rename Method refactoring will be invoked. Similarly, if the programmer selects a class name and invokes the Rename refactoring, the Rename Type refactoring will be invoked. However, this mechanism results in the invocation of an unexpected refactoring at times.

Figure 4.1 shows two very similar code selections that are interpreted differently by the Move refactoring. The selection in Figure 4.1a results in an invocation of Move Static Members, while the selection in Figure 4.1b results in an invocation of Move Compilation Unit. This is because the former
selection tightly covers a method, while the latter covers slightly more than a method (leading spaces on line 2, Figure 4.1b). The refactoring tool interprets a selection that covers more than a method as a selection of the enclosing compilation unit.

This high sensitivity to code selections is a usability problem. Reviewing our report of this usability problem [7] led the Eclipse developers to discover more problems. The root cause of the problem is that the Move refactoring only considers the start offset of the selection. This observation uncovered another problem: if a programmer selects two methods, the Move refactoring will ignore the second one.

4.4 Lessons Learned

Although we applied CIT to find the usability problems of an IPT tool, we learned some lessons along the way that are generalizable to other programming tools.

**Reported messages.** We found that the reported refactoring messages were more likely to indicate usability problems than other events. Nevertheless, we identified at least one usability problem by just examining canceled refactorings. Repeated invocations of an automated refactoring helped us infer how the participant overcame a usability problem. Unlike an adaptation of CIT to applications like Adobe Photoshop [35], we did not find any usability problems by just studying undone and redone refactorings. Nonetheless, undone and redone refactorings in a refactoring batch provided stronger evidence for the usability problems revealed by other events, e.g., reported messages. There are several possible explanations for why undo is an indicator of usability problems for an application like Adobe Photoshop but not a refactoring tool. First, the role of the undo operation may be different in the two applications. Programmers seem to use undo as a natural means of exploring a solution space. Second, inferring usability problems from an undone automated refactoring may require more contextual information that CODINGSPECTATOR does not capture.

**Reproducibility.** We found that reproducing the critical incidents was required in most cases to infer the usability problems. This suggests that the reproducibility of critical incidents is an important criteria in adapting CIT to IPT tools and other domains. The data collector has to capture enough details about the incidents so that the evaluator can later reproduce the critical incidents and examine them.
**User reports.** We contacted a participant to infer one usability problem (Section 4.3.5). We were able to refresh the participant’s memory by showing him CodiNgSpeCtator’s data of his refactoring. The lesson here is that it is sometimes necessary to get the participant’s report of the critical incident, e.g., what they were trying to achieve, what prevented them from achieving their goal, and how they overcame the problem. When employing CIT in a remote usability evaluation, there are two general ways of getting the participants' reports of the critical incidents: either ask the participants to report the incidents as they occur during the task, or prompt them by presenting enough data about their actions not too long after performing the task.

**Design improvements.** Although our goal was to infer usability problems from alternate paths, we learned that alternate paths can also suggest design improvements. We analyzed the repeated refactorings to see how the programmers overcame a usability problem manually. For some usability problems (Sections 4.3.2.1 and 4.3.6.1), we suggested design improvements that automated these manual strategies.

### 4.5 Limitations

This work has several limitations, which call for future work in this area.

Although we successfully identified usability problems, we do not have an estimate of what fraction of the usability problems we identified. Future work may answer this question by comparing the results of our method by conventional usability evaluation methods. Eclipse developers acknowledged all of the usability problems that we reported. Nonetheless, developers' judgments may not be perfect.

Although the contextual information that CodiNgSpeCtator collects allows us to identify usability problems, our identification process was mostly manual and tedious. Future research could explore automated techniques to reduce the burden on the evaluator. We used heuristics such as the number of occurrences of the events and affected participants to prioritize our evaluation efforts. Another possibility is to ask the users to report the critical incidents that they encounter. The study of the advantages and disadvantages of such an approach is left to future work.

We adapted and evaluated CIT for the Eclipse refactoring tool. Since we were able to extend some of the results to other refactoring tools, we expect our results to generalize. Nonetheless, we did not thoroughly investigate the generalizability of our method to other refactoring and IPT tools.
CODING SPECTATOR captures snippets of code close to the program elements under refactoring. We found that this information was crucial for deriving usability problems. Nonetheless, recording such sensitive pieces of information raises privacy and confidentiality issues. We faced difficulty in recruiting participants because of these issues. There are several challenges in scaling our method to a large number of programmers. One challenge is to make the data collection transparent. Another challenge is to design an incentive mechanism for programmers to share their data.
Figure 4.4: The Extract Method refactoring of Eclipse results in a compilation problem if it reports the ERROR message: “Selected statements contain a return statement but not all possible execution flows end in a return. . . . “. 

(a) The selection contains a return statement but not all possible execution flows end in a return.

(b) The method generated by Extract Method has compilation problems. The return type of method \( n \) is wrong and a return statement is missing before line 14.

(c) It is possible to automatically infer the missing return statement in some cases.
class C {
    int m() {
        int a = 0;
        return a + 1;
    }
}

Figure 4.5: The Eclipse Extract Local Variable refactoring reports a message for the code selection shown in this figure and refuses to proceed: “An expression has to be selected to activate this refactoring. Names used in declarations are not expressions.”

class C {
    D d1;
    D d2;
    void m(E e1, E e2) {
        d1.m();
    }
    void m2() {
        E e1 = new E();
        E e2 = new E();
        C c = new C();
        c.m(e1, e2);
    }
}
class D {
    void m() {
    }
}
class E {
    void m(C c, E e2) {
        c.d1.m();
    }
}

(a) Original code

class C {
    D d1;
    D d2;
    void m2() {
        E e1 = new E();
        E e2 = new E();
        C c = new C();
        e1.m(c, e2);
    }
}
class D {
    void m() {
    }
}
class E {
    void m(C c, E e2) {
        c.d1.m();
    }
}

(b) Refactored code

Figure 4.6: The programmer selects instance method m (line 4) to move it from class C to E.
Figure 4.7: The configuration options of the Move Instance Method refactoring when applied on the code example shown in Figure 4.6. The Move Instance Method refactoring requires the new “target” of the method as a pair of “name” and “type”. However, this requirement is neither easy to interpret by programmers nor always necessary.

Figure 4.8: Inclusion or exclusion of a semicolon can make a selection invalid for Extract Method.
(a) An ill-formed selection of statements that can be made well-formed if it is extended to include the opening brace (‘{’) on line 3

(b) An ill-formed selection of statements that can be made well-formed if it is extended to include the closing brace (‘}’) on line 8

Figure 4.9: The Extract Method refactoring requires a selection of a set of statements. This figure shows selections that are invalid due to missing braces.
Chapter 5

A Compositional Paradigm of Automating Refactorings

Mainstream refactoring tools follow a wizard-based paradigm. Typically, a programmer selects a piece of code in the editor and invokes an automated refactoring from a menu. The programmer may then change some of the configuration options on the wizard. These options control the outcome of the refactoring by specifying the entities that should be created, copied, or moved. The tool may also preview the change, e.g., by showing snapshots of the affected files before and after the refactoring side-by-side.

Prior studies have identified several problems with the wizard-based paradigm of refactoring (Chapter 3) [121]. For instance, the long list of automated refactorings in the menu leads to higher learning and invocation costs. Programmers tend to know only a few of the automated refactorings supported by the IDE. The context-switch from a code editor to a wizard disrupts the programming workflow. The wizard imposes an upfront configuration cost, making it difficult to control the outcome of the tool. The preview page of the wizard is often too cluttered, which makes the refactoring tool less predictable. That is, the programmer cannot easily predict how the tool is going to affect her code. Even if a programmer makes her way through all the steps of invocation, configuration, and preview, the wizard may still notify her at the end that the refactoring is impossible or unsafe to perform. These problems call for rethinking the design of refactoring tools.

A key contribution of this dissertation is proposing a new paradigm, called compositional refactoring, for automating complex refactorings. The idea is to have the tool automate small, predictable changes and let the programmer manually compose these changes into complex ones. For instance, rather than performing a large refactoring such as Extract Superclass in a single step, the compositional paradigm automates small steps, e.g., Create New Superclass, Rename Class, and Move Member to Immediate Superclass, leaving it to the programmer to compose these steps into Extract Superclass.

The goal of compositional refactoring is to address some of the shortcomings of the existing monolithic, wizard-based design of refactoring tools, e.g., low predictability and control and high configuration and
learning costs. The compositional paradigm achieves these goals by lowering the level of automation, i.e., automating small changes, and putting the programmers in control by letting them compose the small changes into larger ones. Although it may seem counterintuitive that reducing the level of automation improves an automated tool, this phenomenon is not new. Other fields such as aviation, health-care, and manufacturing have gone through a similar process. Motivated by the perceived benefits of automation in these fields, highly automated systems were invented, often neglecting the role of the human operators. Further studies showed that often a less automated system with a human-centered design performs better, concluding that less is (sometimes) more, when it comes to automation [41, 98, 161].

The idea of compositional refactoring is inspired by our studies of the refactoring practices of programmers in the wild. Even though expert practitioners recommend that programmers perform refactorings as a composition of smaller ones [75, 95], little is known about how programmers compose refactorings in practice. Therefore, we mined two refactoring data sets: the Eclipse foundation and Illinois data sets. The Eclipse foundation has collected data from hundreds of thousands of programmers over the years. Our data mining of this large corpus of data shows that programmers frequently invoke a variety of multiple automated refactorings within a short period of time. Nonetheless, refactorings invoked in close time proximity may be semantically unrelated. Therefore, we consulted the Illinois data set, which we collected during a prior field study from 30 programmers over eight months. The Illinois data set is smaller but contains more contextual information about refactoring invocations. This detailed data allowed us to determine the semantic relation between refactoring invocations. We manually inspected a sample of the refactorings of the Illinois data set that were invoked in a short time span. This analysis reveals some of the rationales for systematically composing automated refactorings, providing further evidence for the naturalness of the compositional paradigm to programmers.

We evaluated the idea of compositional refactoring in two ways. We first distributed an online survey to get early feedback from hundreds of programmers on our design (Section 5.4). The survey presented mockup screenshots of compositional and wizard-based refactorings and asked the participants to compare the two paradigms. The survey results showed that programmers are receptive to the idea of compositional paradigm and provided improvement suggestions. This positive response motivated us to implement the compositional paradigm.
We enhanced the design based on the feedback we received from the survey participants. Then, we implemented it as an Eclipse plug-in. Finally, we conducted a lab study with 13 professional programmers at a large software company (Section 5.5). We instructed the participants to perform a refactoring on an open-source project using the compositional and wizard-based refactoring tools in a random order. Like the survey participants, the majority (nine) of the lab study participants were more satisfied with the compositional paradigm than the wizard-based one. In addition, the participants were more likely to finish the task correctly and significantly faster in the compositional paradigm.

Overall, the participants of the survey and lab studies appreciated the compositional paradigm because of its perceived higher level of control, easier method of invocation, and interactivity. In addition, they suggested features like abstract views and multi-selections. These results suggest that compositional refactoring is a promising paradigm for assisting programmers in performing complex refactoring. Our work contributes to the refactoring practice in several ways:

1. We provide empirical evidence for the prevalence and nature of automated refactorings that are invoked in close time proximity (Section 5.1).

2. We discuss some of the rationales for composing automated refactorings based on our manual inspection of the Illinois data set (Section 5.2).

3. We propose a new paradigm for automating complex refactorings, namely compositional refactoring (Section 5.3).

4. We provide an implementation of compositional refactoring as an Eclipse plug-in (Section 5.5.1).

5. We show the effectiveness of compositional refactoring through a survey (Section 5.4) and a lab study (Section 5.5).

6. We draw implications from our analyses of refactoring usage data, survey study, and lab study for designing future tools that better support complex refactoring.

Our tool and study artifacts are publicly available [19].
5.1 Frequent Refactoring Sets

In this section, we answer the following research questions:

- How frequently do multiple kinds of refactorings occur in a short time span?
- How diverse are the refactorings frequently invoked in a short time span?

Answers to these questions provide a bird’s-eye view of the phenomenon of invoking several automated refactorings in a short time span.

5.1.1 Eclipse Foundation Data Set

Usage Data Collector (UDC) is a pre-installed plug-in in Eclipse, which records uses of Eclipse commands, views, and perspectives. UDC generates a fresh identifier for the user and persists it in the home folder of the user. For each event or action performed by the user, UDC captures the timestamp, the event identifier, the user identifier, and the bundle that generated the event. If a user agrees, UDC regularly sends the user’s data to the Eclipse foundation’s servers. We analyzed a subset of the UDC data that contained information about the invocations of the Eclipse refactoring tool for Java. The Eclipse foundation has released the data from a total of 195,105 programmers who used the Eclipse refactoring tool for Java during 20 months from January 2009 until August 2010.

5.1.2 Data Analysis

We used the large data set of the Eclipse foundation to infer the frequent refactoring sets, the sets of automated refactorings that are frequently invoked in a short time span. Since the refactorings invoked in temporal proximity may not be semantically related, this analysis only provides a bird’s-eye view of the frequency and variety of compositions of automated refactorings in the wild.

5.1.2.1 Refactoring Batches

Intuitively, a refactoring batch is a maximal set of automated refactorings, such that the consecutive refactorings are invoked within a close time proximity. A refactoring batch is a nonempty set of refactoring
kinds. For instance, the refactoring batch \{Move, Rename\} may stand for one or more invocations of Move and Rename in any order within a close time proximity.

We partitioned the refactoring events of the data set into refactoring batches using a heuristic. The heuristic uses the large gaps between the invocation times of consecutive refactorings as the partition boundaries. This heuristic is based on the assumption that refactorings invoked far apart in time are less likely to be semantically related. First, the partitioning algorithm sorts the refactoring events of every UDC user by invocation time. Next, the algorithm creates a new batch for each user and adds the kind of the first refactoring event of the user to the batch. If a refactoring event is invoked by the same user within $\delta$ minutes of the preceding event, the algorithm will add the kind (Rename, Move, etc.) to the batch of the preceding event. Otherwise, the algorithm will add the refactoring kind to a new batch. When the batch of every refactoring event is determined, the algorithm terminates and returns the set of refactoring batches.

5.1.2.2 Mining Frequent Refactoring Sets

A refactoring set is a nonempty subset of a refactoring batch. The support of a refactoring set $R$ is the fraction of refactoring batches that are supersets of $R$. We applied a frequent itemset mining algorithm [84, pp. 246–50] on the set of refactoring batches to infer the frequent refactoring sets—the refactoring sets with the highest supports.

We used an implementation of the frequent itemset mining algorithm in the statistical computing software R [31] to infer the frequent refactoring sets. We provided the algorithm with refactoring batches ($\delta = 10$) and set the parameter $\minsup$ to 0.001. The output of the algorithm is a list of refactoring sets with a support of at least $\minsup$.

We repeated the analysis for each $\delta \in \{5, 10, 20, 40\}$, and compared the resulting frequent refactoring sets. Due to the negligible effect of such changes of $\delta$ on the most frequent refactoring sets, we present the results for only $\delta = 10$.

5.1.3 Results

The data mining algorithm inferred 47 frequent refactoring sets for all UDC users. However, the vast majority of UDC users use automated refactorings rarely. As shown in Figures 5.1 and 5.2, most (98.6%)
users invoked the refactoring tool at most 50 times, and 98.9% invoked at most five kinds of automated refactorings. We consider users who invoked at least five kinds of automated refactorings for a total of at least 50 times active and the rest inactive. This leads to 1,188 active users with a total of 112,885 refactoring events.

We hypothesized that the data of inactive users conceals some of the frequent refactoring sets of the active ones. Thus, we repeated the data mining algorithm on the active users alone. This resulted in about three times more frequent refactoring sets ($N = 150$), 44 of which were also inferred for all users. This indicates that limiting the data set to active users uncovers more frequent refactoring sets. Table 5.1 lists the most frequent refactoring sets of active UDC users. This table shows the following:

1. Programmers invoke a variety of automated refactorings in a short time span.
2. Some refactoring sets with multiple refactoring kinds are more frequent than those with a single kind. For instance, the refactoring set \{Extract Local Variable, Extract Method\} is about 2.5 times more frequent than \{Pull Up\}. In other words, a refactoring batch is more likely to contain Extract Local Variable and Extract Method than at least one Pull Up. This result reveals a limitation of prior studies (Chapter 3) [112, 121], which focused only on individual refactorings.

5.2 Refactoring Composition Patterns

The frequency and variety of refactoring sets (Section 5.1.3) led us to the hypothesis that programmers *systematically compose* certain kinds of automated refactorings to apply larger changes. This section presents answers to the following reseach questions:
Table 5.1: The 20 most frequent refactoring sets of UDC active users. A refactoring batch is the kinds of a maximal subset of automated refactorings such that the consecutive refactorings are invoked within 10 minutes. A refactoring set is a subset of a refactoring batch. For instance, the refactoring set \{Pull Up\} stands for one or more invocations of Pull Up in a refactoring batch. The support of a refactoring set R is the fraction of batches that are supersets of the R.

<table>
<thead>
<tr>
<th>refactoring set</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Rename}</td>
<td>0.591</td>
</tr>
<tr>
<td>{Extract Local Variable}</td>
<td>0.270</td>
</tr>
<tr>
<td>{Extract Method}</td>
<td>0.154</td>
</tr>
<tr>
<td>{Inline}</td>
<td>0.090</td>
</tr>
<tr>
<td>{Extract Local Variable, Rename}</td>
<td>0.076</td>
</tr>
<tr>
<td>{Move}</td>
<td>0.058</td>
</tr>
<tr>
<td>{Extract Method, Rename}</td>
<td>0.057</td>
</tr>
<tr>
<td>{Change Method Signature}</td>
<td>0.055</td>
</tr>
<tr>
<td>{Extract Constant}</td>
<td>0.043</td>
</tr>
<tr>
<td>{Extract Local Variable, Extract Method}</td>
<td>0.042</td>
</tr>
<tr>
<td>{Inline, Rename}</td>
<td>0.033</td>
</tr>
<tr>
<td>{Extract Local Variable, Inline}</td>
<td>0.031</td>
</tr>
<tr>
<td>{Extract Method, Inline}</td>
<td>0.027</td>
</tr>
<tr>
<td>{Convert Local Variable to Field}</td>
<td>0.025</td>
</tr>
<tr>
<td>{Move, Rename}</td>
<td>0.024</td>
</tr>
<tr>
<td>{Change Method Signature, Rename}</td>
<td>0.022</td>
</tr>
<tr>
<td>{Extract Local Variable, Extract Method, Rename}</td>
<td>0.020</td>
</tr>
<tr>
<td>{Pull Up}</td>
<td>0.016</td>
</tr>
<tr>
<td>{Extract Local Variable, Inline, Rename}</td>
<td>0.015</td>
</tr>
<tr>
<td>{Extract Constant, Rename}</td>
<td>0.015</td>
</tr>
</tbody>
</table>

- Do programmers compose automated refactorings?
- What are some of the rationales for composing automated refactorings?

The analysis of the Eclipse foundation data set showed that certain kinds of automated refactorings (e.g., \{Extract Local Variable, Extract Method\}) are frequently invoked in a short time span. Although this data set is huge, it does not capture enough context about each event to infer the rationales for invoking several automated refactorings in a short time span. Therefore, we analyzed the smaller but more detailed Illinois data set.
5.2.1 Illinois Data Set

The Illinois data set comes from two of our Eclipse-based data collectors, namely CODINGSPECTATOR and CODINGTRACKER (Chapter 2). CODINGTRACKER records applications of all 33 automated refactorings of Eclipse, and CODINGSPECTATOR records more detailed data (e.g., the piece of code surrounding the refactored program element and error messages reported by the refactoring tool) for 23 automated refactorings.

The Illinois data set contains data from 30 programmers consisting of a total of 2,296 programming hours over eight months. Fourteen of our participants were external developers who we recruited by sending invitation messages to individual programmers, mailing lists, and IRC channels of open-source projects. We also recruited twelve graduate students and four interns from six research labs at the computer science department of the University of Illinois at Urbana-Champaign. Based on the results of our demographic survey that 28 participants took, 1, 5, 15, and 7 participants had 1–2, 2–5, 5–10, and more than 10 years of programming experience, respectively.

5.2.2 Data Analysis

The partitioning algorithm (Section 5.1.2.1) found 1,633 refactoring batches of 244 kinds in the Illinois data set. We selected 32 kinds of batches, which were frequent in the Illinois data set or contained the frequent refactoring sets of the Eclipse foundation data set. Then, we manually analyzed a random sample of at most ten batches of each kind, leading to a total of 139 batches.

We examined the information captured for each refactoring event in a batch, e.g., the kind, invocation time, error messages, and the piece of code surrounding the selection. Based on these data, we decided if the refactorings in the batch were semantically related, and inferred a rationale for the batch. Next, for each batch kind, we collected the rationales of the batches of that kind. Finally, we collected five refactoring composition patterns. A refactoring composition pattern is a recurring set of automated refactorings that programmers compose for similar rationales.

5.2.3 Results

We found that the majority (81%, i.e., 112 of 139) of the analyzed batches contained related refactorings. The following presents the refactoring composition patterns that we observed in our sample of refactoring
batches. Each pattern reveals some of the rationales for composing refactorings, providing evidence that programmers systematically compose automated refactorings. The value of \( n \) in parentheses shows the number of refactoring batches with a particular property.

5.2.3.1 Refactoring Closely Related Entities \((n = 47)\)

We found that programmers frequently compose refactorings to refactor closely related entities that are not related by name binding. For instance, the participants composed several Rename refactorings on program entities with similar names \((n = 8)\) or a method and the variable that gets the return value of the method \((n = 2)\). As another example, our participants performed the Rename refactoring to rename a field and the constructor parameter that initialized the field \((n = 2)\).

Refactoring tools only update the entities that are syntactically related. For instance, the Rename refactoring updates the declaration and all references of a name. One recommendation for future tools is to support this refactoring composition pattern by reliably detecting the names that are likely to co-evolve. For instance, the tool might be able to detect related entities based on some heuristics, e.g., the similarity of the names, the distance and the relation of the corresponding program entities, and the programmers’ past Rename refactorings. A tool that is aware of the co-evolution patterns of names could assist programmers in updating related names and warn about inconsistent naming schemes.

5.2.3.2 Adapting Extract Method \((n = 34)\)

We found that programmers compose three kinds of refactorings, Extract Local Variable, Extract Method, and Inline Local Variable, to adapt the outcome of Extract Method (Figure 5.3). This refactoring composition pattern consists of three steps: preparation, method extraction, and simplification. First, the programmer performs Extract Local Variable so that the upcoming Extract Method refactoring infers a method parameter corresponding to the extracted local variable. Second, she invokes Extract Method on a piece of code excluding the declarations of any variables added during the preparation step. Finally, the programmer invokes Inline Local Variable to simplify the code. A refactoring batch with the method extraction step may contain only the preparation step \((n = 11)\), only the simplification step \((n = 19)\), or both \((n = 4)\). It is impossible to configure the refactoring wizard of Extract Method to extract the same method in one step.
public static void main(String[] args) {
    int factorial = 1;
    for (int i = 1; i <= 10; ++i) {
        factorial *= i;
    }
    System.out.println(factorial);
}

public static void main(String[] args) {
    int n = 10;
    int factorial = 1;
    for (int i = 1; i <= n; ++i) {
        factorial *= i;
    }
    System.out.println(factorial);
}

public static void main(String[] args) {
    int n = 10;
    int factorial = getFactorial(n);
    System.out.println(factorial);
}

private static int getFactorial(int n) {
    int factorial = 1;
    for (int i = 1; i <= n; ++i) {
        factorial *= i;
    }
    return factorial;
}

(a) The initial version of the code. The programmer extracts 10 into a new local variable n.

(b) The programmer reorders the declarations of factorial and n to exclude n from his future code selection.

(c) The programmer extracts the piece of code that computes the factorial of n into a new method getFactorial(int).

(d) The programmer inlines local variable n for it is now used just once, and it was needed just to adjust the outcome of the Extract Method refactoring.

Figure 5.3: Participants composed Extract Local Variable, Extract Method, and Inline Local Variable to extract methods with their desired signatures.

Instead of composing three refactorings to include certain parameters in the signature of the extracted method, the programmer could compose just two refactorings, namely, Extract Method and Introduce Parameter (Figure 5.4). However, there were no instances of the latter in the Illinois data set. In general, the automated Introduce Parameter refactoring is used infrequently and fewer programmers know about it (Chapter 3) [121]. Nonetheless, a programmer can adapt Extract Method without the need to learn the Introduce Parameter refactoring.

The following are some of the rationales of this refactoring composition pattern:

- configuring a refactoring in ways not supported by a wizard
- avoiding the need to learn additional kinds of automated refactorings
public static void main(String[] args) {
    int factorial = 1;
    for (int i = 1; i <= 10; ++i) {  
        factorial *= i;
    }
    System.out.println(factorial);
}

Figure 5.4: Even though it is possible to adapt the Extract Method refactoring with Introduce Parameter, none of the participants exhibited this composition pattern.

If a refactoring tool is aware of the composition patterns for adapting a refactoring, it could offer the programmer the opportunity to perform the simplifications in one step.
extracted the same piece of code into a constant \((n = 2)\). This indicates that programmers experiment with refactorings or accidentally invoke the wrong refactoring.

5.2.3.4 Composition-over-configuration \((n = 8)\)

Composition-over-configuration is a composition pattern that we found programmers employ to avoid the upfront configuration cost of refactoring wizards. With this pattern, the programmer composes multiple automated refactorings to perform a refactoring that could have been done by configuring a refactoring wizard.

For example, it is possible to configure the Pull Up refactoring wizard to move one or more members (fields or methods) of a class to a superclass in one step. However, the participants sometimes composed two Pull Up refactorings to pull up two members of a class one at a time \((n = 2)\).

As another example, the Extract Local Variable refactoring wizard allows the programmer to set the name of the new variable. Nevertheless, a participant composed the Extract Local Variable refactoring by a Rename refactoring \((n = 1)\), even though he could have achieved the same outcome by configuring the Extract Local Variable refactoring wizard.

Although configuration options make the refactoring tool more powerful, they also increase the cost of using the tool and make its benefits less immediately apparent (Section 3.3.6) \([121]\).

Observing the composition-over-configuration pattern, we propose the compositional paradigm of automating refactorings (Section 5.3). We implemented the compositional paradigm using a feature of Eclipse called Quick Assist (Figure 5.5). Quick Assist is a popular method of invoking refactorings that supports composition-over-configuration (Section 3.2.1). For example, if a programmer invokes Extract Method through Quick Assist, it would apply Extract Method with a default name and then initiate a composition with Rename on the method name.

5.2.3.5 Multiple Refactorings on an Entity \((n = 6)\)

A program entity may undergo multiple refactorings. For instance, the participants composed Extract Method with Pull Up on the same method to refactor to the Template Method design pattern \((n = 2)\) \([76, p. 325]\). As another example, a participant composed Push Down with Encapsulate Field on the same field \((n = 1)\).
5.3 Design of a Compositional Refactoring Tool

Based on the lessons learned from our data analysis, literature review, and our prior research studies, we compiled a list of design goals for a new refactoring tool. These goals inspired the design and implementation of a tool for compositional refactoring. The tool currently supports several refactorings such as Extract Superclass, Extract Interface, Pull Up, and Push Down, and can be extended to support other refactorings. In this section, we first discuss our design goals and then explain the steps involved in performing the Extract Superclass refactoring using our tool.

5.3.1 Design Goals

5.3.1.1 Predictability

Our prior study showed that programmers rarely use the automated refactorings whose outcomes are not easily predictable (Section 3.3.5), e.g., the refactorings that affect several files. Our goal is to make such refactorings more predictable. One strategy to achieve predictability, employed by the wizard-based refactorings, is to assist the programmer in reviewing the changes. Another strategy, which we have explored in our compositional paradigm, is to divide a large refactoring into smaller, predictable refactorings.

5.3.1.2 Control

Programmers prefer to maintain control over the evolution of code during refactoring (Chapter 3). By maintaining control over the evolution of the code, the programmer is able to better evaluate, verify,
and customize the refactoring. One way to control a refactoring is to allow the programmer to configure it upfront. However, configuration dialogs increase the cost of using the tool (Section 3.3.6) [121], and programmers rarely configure the refactoring tool [121]. An alternative paradigm is to put the programmers in control of the refactoring by assisting them in performing the refactorings in smaller steps. In this paradigm, the programmers can evaluate the refactorings at each step and intersperse them with manual edits.

5.3.1.3 Discoverability

Researchers and tool vendors continue to automate more recurring code transformations [66, 99, 143, 158, 167]. However, programmers discover only a subset of the automated refactorings in their IDEs (Section 3.3). Quick Assist makes the refactorings more discoverable by proposing them based on the current context. It is a feature of Eclipse that proposes only the refactorings that are applicable to the current context. Programmers frequently use Quick Assist to discover and invoke refactorings (Section 3.2.1). Thus, our tool relies on Quick Assist as its method of invocation. There are various other complementary techniques for raising programmers’ awareness about automated refactorings. For example, one could provide social indicators to facilitate the transfer of knowledge about automated refactorings between programmers. Another technique is to detect manual refactorings online and notify the programmer about the availability of automated support for the ongoing refactoring [74, 77].

5.3.1.4 Learnability

The list of automated refactorings in modern IDEs is long. The cost of learning so many tools is a barrier to their adoption. Our goal is to solve this problem by allowing programmers learn a small number of reusable automated refactorings and compose them in a variety of ways to perform many kinds of larger refactorings.

5.3.1.5 Low Disruptiveness

Although configuration dialogs make the refactoring tool more powerful and customizable, they distract the programmers from the code and disrupt their flows of programming (Section 3.3.6) [121]. We aim to design refactoring tools that are highly interactive and allow the programmer to focus on the code.
5.3.1.6 Correctness

Wizard-based refactorings guarantee correctness by checking a set of preconditions. Similarly, compositional refactorings check the preconditions of the individual steps. Because the steps are small, we expect that programmers can verify them more easily. Moreover, the programmer can run tests after each step. This allows the programmer to identify the exact step that led to a problem.

5.3.2 Compositional Extract Superclass Refactoring

Our goals informed the design of a compositional refactoring tool. We use the Extract Superclass refactoring as an example to demonstrate the compositional paradigm. This refactoring lets the programmer create a superclass for one or more classes and move some of the members of the subclasses to the superclass. We chose Extract Superclass because it is one of the more complex and less frequently used automated refactorings of Eclipse (Chapter 3) [121]. Figure 5.6 shows our mockup of compositional Extract Superclass. We later improved the mockup and implemented it as an Eclipse plug-in (Section 5.5.1). In the following, we briefly describe how the tool works.

1. The programmer selects the class (Daisy) to extract a superclass from.

2. The programmer invokes the Quick Assist menu and selects the “Create new superclass in file” action.

3. This creates a new superclass in the current file and prompts the programmer for a new name (Flower) for the superclass.

4. The programmer selects the water method, invokes the Quick Assist menu and selects “Move to immediate superclass”.

5. This moves method water from Daisy to Flower.

6. Once the programmer is done moving all the desired fields and methods, the programmer selects Flower, invokes the Quick Assist menu, and selects “Move type to new file”.

7. This moves class Flower to a new file and completes the Extract Superclass refactoring.
The outcome of each of the above steps is immediately visible to the programmers in the code editor. At each step, the Quick Assist menu suggests a set of actions that are applicable to the selected element. The steps are independent of each other. That is, Quick Assist suggests the steps regardless of what step was previously performed. The programmers do not have to perform every step using our Quick Assist actions. Rather, the programmers can switch between the two modes: manual and automated. That is, they can complete some steps manually and leverage the Quick Actions for performing the rest.
5.3.3 Decomposition Process

The following describes the process that we followed to decompose refactorings into primitive changes. A refactoring tool designer can follow this process to design a compositional refactoring tool. The decomposition process is iterative. It is expected that the designer revisits any of the previous steps of the process, if needed.

5.3.3.1 Enumerate the steps of the refactorings

The first question that a designer should answer when using the compositional paradigm is what the refactoring does and how. There are two general ways of identifying the steps of a refactoring. If the refactoring has been documented in refactoring catalogs, the designer can consult these sources to identify the steps of the refactoring. Another approach is to observe programmers while performing the refactoring to understand the steps that they take to perform the refactoring.

5.3.3.2 Refine the steps of the refactorings

Rather than decomposing each refactoring in isolation, the refactoring tool designer should consider the decompositions of other refactorings while decomposing a refactoring. In this holistic approach to refactoring decomposition, the designer should add, remove, split, or merge the steps of refactorings to make the decompositions more consistent and increase the opportunity to use the primitive changes.

5.3.3.3 Identify the steps to automate

Not all steps of a refactoring may be amenable to automation. The designer should identify the steps to automate based on the following criteria:

- The change performed in each step should be predictable. That is, the programmers should be able to easily tell how the change is going to affect the code.

- There should be a high opportunity to use the automated change. Automating a change that is rarely performed has little benefit and can make the tool less usable by making it cluttered.

- The automation of the change should require no configuration.

- The automated change should be easy to name.
5.3.3.4 Evaluate the refactoring decompositions

The process of decomposing refactorings is a creative design process. It is challenging to find the right granularities of the primitive changes the first time. Thus, the refactoring tool designer should iteratively evaluate the decompositions through a series of low and high fidelity prototypes.

5.4 Survey Study

We distributed a survey to assess our design goals and compare our compositional prototype of the Extract Superclass refactoring (Figure 5.6) with the existing wizard-based user interface of this refactoring in Eclipse. The goal of the survey study is to answer the following research questions:

- How do programmers compare the compositional and wizard-based paradigms?
- Are programmers likely to adopt the compositional paradigm?
- What are some opportunities for improving the compositional paradigm of refactoring?

Answering these questions shows how receptive programmers are to the new compositional paradigm.

5.4.1 Method

We recruited 100 programmers by announcing the survey\(^1\) on reddit.com\(^2\), twitter.com, and mailing lists of open source projects. The survey was estimated to take 20 minutes, and started with questions about the experience of the respondent with programming, IDEs, and refactoring. Then, it asked about the programmer’s strategy in performing the Extract Superclass refactoring. Finally, the survey presented screenshots of the two user interfaces of the Extract Superclass refactoring, and asked the respondent to evaluate and compare the two.

5.4.2 Thematic Coding

We employed thematic coding [162], a systematic qualitative method, to analyze the responses to open-ended questions. The coding was inductive (data-driven) as opposed to deductive (theory-driven). We

\(^1\)https://illinois.edu/fb/sec/8454746
\(^2\)http://www.reddit.com/r/programming
Table 5.2: The distribution of responses to the question “Consider the situations in which you need to refactor your code. What portion of the time, in these situations, do you use a refactoring tool?”

<table>
<thead>
<tr>
<th></th>
<th>Never</th>
<th>Rarely</th>
<th>Some of the time</th>
<th>Most of the time</th>
<th>Nearly every time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage</td>
<td>8%</td>
<td>10%</td>
<td>22%</td>
<td>32%</td>
<td>28%</td>
</tr>
</tbody>
</table>

extracted the opinions and ideas associated with each segment of the comments. Through an iterative process, we defined, merged, and split the themes to better identify the central ideas. The goal of such a coding is to identify the major ideas not to count the frequencies of keywords. This coding allowed us to reliably decide if two participants provided equivalent responses. For each statement that we report, we include the number of participants that agree with it as an indication of its overall support.

5.4.3 Participants

The participants were familiar enough with modern programming environments to evaluate the compositional paradigm. The majority (91%) of the survey respondents had more than five years of programming experience. Of all the participants, 76% considered themselves to be experts in at least one programming language (on a five-point Likert scale ranging from “Unfamiliar” to “Expert”), and 99% rated themselves as either very familiar with or expert at one or more languages. On average, the respondents rated themselves as familiar with Java, Javascript, and C and somewhat familiar with C++, C#, Python, Ruby, and PHP. The respondents indicated that they were familiar with Eclipse (81%), Visual Studio (58%), NetBeans (39%), IntelliJ (36%), and Xcode (28%).

5.4.4 Results

Of all the survey participants, 60% indicated that they used a refactoring tool when they needed to refactor their code most of the time or nearly every time (on a five-point Likert scale ranging from “Never” to “Nearly every time”) (Table 5.2).
Before presenting the two interfaces to the participants, we asked about their current strategy for performing an Extract Superclass refactoring. We defined an Extract Superclass refactoring task as creating a new superclass for an existing class and moving a method from the existing class to the new superclass. Of the 97 participants who responded to this question, 38% reported that they would use the Extract Superclass automated refactoring to complete the described task; 31% indicated that they sometimes would use the refactoring tool to complete the task and other times they would do it manually; 31% said that they would do the refactoring manually.

Of the 71 who explained the rationale behind their strategy for performing the Extract Superclass refactoring with or without using the tool, eight said that they rarely needed to perform the Extract Superclass refactoring; 11 said that it was easier or less work to perform the refactoring using the tool, while seven said that the refactoring tool had a high learning cost or mental overhead, was difficult to invoke, or it was easier to do the refactoring manually; 16 said that the refactoring tool was faster; 20 said that the refactoring tool was more reliable while five considered the refactoring tool as being unreliable; six said that doing the refactoring manually would give them more control over their code, allows them to tailor the refactoring to their needs, or makes it easier for them to evaluate the refactoring.

The survey presented screenshots of the steps to perform the Extract Superclass refactoring using both the compositional and wizard-based paradigm on the same page. The survey asked the participant how often he would use each interface on a five-point Likert scale ranging from “Never” to “Nearly every time”. The majority (66%) of respondents said that if both compositional and wizard-based paradigms are available, they would use the compositional paradigm at least as frequently as the wizard-based one. More interestingly, those who did not use an existing tool for Extract Superclass or used the tool some of the time were more likely to prefer the compositional paradigm (Table 5.3). This shows that the compositional paradigm is a promising technique for increasing the utilization of automated refactorings.

Finally, the survey asked the participants to compare and evaluate the paradigms. We applied a qualitative analysis method (Section 5.4.2) on the comments provided by 50 participants. The following discusses the result of this analysis, which reveals the strengths and weaknesses of each paradigm and highlights opportunities for improvement.
Table 5.3: Joint distribution of respondents’ frequency of using the Extract Superclass refactoring wizard and their preferred paradigm (compositional vs. wizard-based). Since three respondents did not indicate their frequency of using the wizard, the last row is slightly different from the sum of the other rows.

<table>
<thead>
<tr>
<th></th>
<th>Prefers composition</th>
<th>Has no preference</th>
<th>Prefers wizard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not use the wizard</td>
<td>21%</td>
<td>4%</td>
<td>6%</td>
</tr>
<tr>
<td>Sometimes uses the wizard</td>
<td>16%</td>
<td>5%</td>
<td>9%</td>
</tr>
<tr>
<td>Uses the wizard</td>
<td>15%</td>
<td>3%</td>
<td>20%</td>
</tr>
<tr>
<td>All respondents</td>
<td>52%</td>
<td>14%</td>
<td>34%</td>
</tr>
</tbody>
</table>

5.4.4.1 Control

Three survey respondents indicated that they would prefer the compositional paradigm, because it provides more control over the evolution of code. For instance, P₅ wrote:

I think the second [compositional] UI [...] gives me the idea of having the control over what’s happening, and how further can I go with it.

This result is consistent with the findings of a prior study, which showed that programmers prefer to maintain control over their code and use automated refactorings whose outcomes are predictable (Section 3.3.5).

5.4.4.2 Invocation Method

Two respondents reported that it was difficult to invoke the wizard-based refactorings from the menu mostly because the menu was too cluttered. Five said that they would prefer keyboard shortcuts. For instance, P₁₀ suggested the following as a way to improve the wizard-based interface:

Make refactoring initiated by keyboard short-cut and not buried so deeply in a menu.

However, as one respondent said, keyboard shortcuts are hard to remember. Quick Assist is a middle-ground, because it is keyboard navigable and proposes only the refactorings that are applicable to the current context. Two participants said that Quick Assist was an easier way of invoking refactorings.
5.4.4.3  Incrementality and Testability

Six respondents mentioned that they do not want refactoring tools with modal dialogs. Five said that modal dialogs are distracting, and three said that the dialogs are too complex. On the contrary, five people indicated that the compositional paradigm is more interactive and two indicated that it allows running tests after each step. For instance, P_7 said:

The second [compositional] one provides a more stepwise view, giving me more intermediate feedback, as well as an ability to run my tests at each step. This goes a long way to making sure the refactoring is the right decision.

Nonetheless, one respondent said that the compositional paradigm had too many steps, and two preferred to perform the refactoring in a single step. For example, P_8 said:

I don’t write Hello World examples. I need control over what gets moved up and what not, what’s made abstract and so forth. I want to do this in one pass, not six [four].

We made compositional paradigm incremental to achieve high control and testability. We decided that compositional refactoring fits programmers’ workflow, because it mimics manual refactorings and programmers already compose refactorings (Section 5.2).

5.4.4.4  Abstract View

Nine participants were concerned that the compositional paradigm may not be suitable for large refactorings. For example, P_2 said:

[I'm] Not sure I’d want to use that [wizard-based] UI for any refactoring work. However, [the wizard-based UI is] probably better for very large refactoring tasks than the second [compositional] UI—but if you’re doing that, you’re doing too much in one go.

Four respondents said that a high-level view of the code would be useful for performing large refactorings. For instance, P_3 said:

[...] However, specifying the methods to be moved one by one rather than selecting them from a list might cause methods that should be extracted into a superclass to be missed. In some sense, it is mostly about whether an abstract view of the methods is preferable to a code level view when
choosing whether to extract them. Sometimes I find myself leaving the extract superclass dialog to figure out what a method actually does and whether it should be extracted.

The wizard-based paradigm lets the programmer operate at the level of classes and methods, but, makes it difficult to switch between the code and its higher-level view. On the other hand, the compositional paradigm that we demonstrated on the survey was tightly coupled with the code, which makes it easy to intersperse low-level code changes with refactorings. To offer the benefits of both code-level and abstract views, the tool could make the switch between the two views seamless, e.g., by making the refactorings available both in the code editor and graphical views.

5.4.4.5 Multi-selections

Five respondents preferred to be able to select multiple program entities and manipulate them at the same time. For example, P_4 said:

Usually, I'm extracting a common superclass to remove duplication from more than one similar class, so I'd need to be able to select multiple classes.

The wizard-based interface allows the programmer to select multiple classes and methods and operate on them in a single step. In a pure compositional paradigm, the programmer would compose more refactorings to simulate multi-selections. However, extending the compositional paradigm to support an abstract view will make it possible to select multiple program elements from the abstract view in one step.

5.4.4.6 Coding Conventions

The mockups of the compositional paradigm showed how to first create an empty superclass in the same file and move it to a new file later (Figure 5.6). Although one could move the superclass to a new file right after creating the superclass, nine preferred the tool to adhere to the coding conventions strictly and never introduce two classes in the same file.

Three people suggested that the tool splits the screen to show the existing class and its new superclass at the same time. Splitting the screen would make it possible to observe the changes made to a large class and its new superclass at the same time with less scrolling. Nevertheless, splitting the screen might
disturb the layout of open editors, especially if the programmer has already split the screen. Alternatively, the tool could create the new superclass in a new file and leave it to the programmer to split the screen.

5.4.4.7 Efficiency

Efficiency was not a priority in our design. However, we expect the compositional paradigm of refactoring to be faster than performing the refactorings manually. Two respondents believed that the wizard-based paradigm would be faster than the compositional compositional. For example, P_1 said:

The second [compositional] UI gives me a feeling that I'm in more control [...]. I suppose the first [wizard-based] UI is quicker though.

However, P_6 said:

The second option [compositional] still gives me the correctness of the first option [wizard-based paradigm], but appears to do so in a smaller, faster, less-intrusive way.

Even though the refactoring wizard makes the whole change in one step, one has to configure its options and review the final change. Our lab study showed that the compositional paradigm is more efficient than the wizard-based paradigm in practice (Section 5.5.5).

5.5 Lab Study

The survey study showed the overall preference of programmers towards compositional refactoring based on participants' evaluations of the mockup screenshots. The goal of the lab study was to answer the following research questions based on programmers' experience with real tools that support refactoring in the two compositional and wizard-based paradigms.

- Which paradigm of refactoring do programmers prefer: compositional or wizard-based?
- Do programmers perform refactorings faster using compositional or wizard-based refactoring?
- Which refactoring paradigm is less error-prone: compositional or wizard-based?
5.5.1 Tool

We implemented an Eclipse plug-in to support Extract Superclass, Extract Interface, Pull Up, and Push Down in the compositional paradigm. Based on the survey study, we improved the design of our tool in several ways. First, we replaced the “Create new superclass in file” action by “Create a new superclass for \( T \) in a new file”, where \( T \) stands for a type name. We made this change to adhere to the coding conventions of Java more strictly. Second, we added an action to the menu called “Create New Superclass” to support multi-selections. When the user selects multiple classes, e.g., in the Package Explorer view, this action would create an empty superclass for the selected classes. Finally, we implemented additional actions such as “Move type \( T \) to a new file”, where \( T \) is a type name, and “Add parameter to method \( m \) for expression”, where \( m \) is a method name (Introduce Parameter in Quick Assist). However, the participants did not use these two actions as they were not applicable to the refactoring task of the lab study.

5.5.2 Participants

All of our participants were experienced programmers who used Eclipse for Java development at a large software company. We first ran a pilot study on three programmers. Then, we conducted the main study on 13 programmers. Of all the participants, two reported having five to ten years of programming experience and 11 reported more than ten years. One participant reported that he was familiar with Java, nine participants considered themselves as being very familiar with Java, and three indicated that they were experts. One participant reported that he was somewhat familiar with Eclipse, four participants rated themselves as familiar, seven said they were very familiar, and one rated himself as expert.

5.5.3 Study Design

We instructed each participant to finish the task in both compositional and wizard-based paradigms (within-subject). Each participant tried the paradigms in a random order (counterbalancing) to overcome the potential carryover effect. We did not ask each participant to try only one paradigm (between-subject) for several reasons. First, individual differences would affect the results of such a study. Second, a between-subject study requires more participants to draw meaningful conclusions. Third, such a study would allow only a quantitative comparison (e.g., efficiency and correctness) of the two paradigms.
However, we felt that such measures were not enough to reliably compare the two paradigms. A participant can offer a qualitative comparison only if he tries both paradigms.

At the beginning of the study, we asked the participants to complete a prequestionnaire. Then, we asked them to perform some introductory tasks to familiarize themselves with the code base. We then instructed each participant to perform the task twice in a random order, once using the compositional paradigm and another time using the wizard-based paradigm. Finally, we asked the participants to rate the two paradigms of refactoring in a postquestionnaire and participate in a semi-structured interview with us. The study took about an hour for each participant, and we offered a $25 gift card to each participant.

5.5.3.1 Refactoring Task

We used a refactoring that had occurred in the open-source project HTMLParser as our refactoring task. Kerievsky used this refactoring as an example of the Extract Composite refactoring [95, p. 214] in his book. Several classes of an old revision of the code base exhibit code duplication. These classes contain a list and a method that iterates over the list and computes its string representation. The fields had different names in different classes and the methods accessed the elements in slightly different ways. We asked our participants to remove this code duplication between two classes by extracting the common field and method into a new common superclass. We limited the refactoring to two classes to make it feasible to finish the refactoring in about 20 minutes.

5.5.3.2 Pilot Study

During the pilot study, we noticed that some participants accidentally introduced subtle errors while refactoring the code. So, we asked the participants of the main study to check that the unit tests passed at the beginning and end of the study. In addition, we instructed the participants to ensure that the new superclass is only referenced by its subclasses. We decided that the existing uses of the subclasses should not be replaced by the new superclass, because of the existing dynamic type checks and casts.
5.5.4 Data Analysis

We measured the task completion time and checked the correctness of the performed task to compare the two paradigms quantitatively. If a participant finished the refactoring task, we compared his resulting code with the expected code in the instructions. If the participant missed some expected changes or introduced unexpected ones, we considered it an incorrect refactoring.

Similar to the analysis of survey comments (Section 5.4.2), we employed thematic coding [162] to systematically analyze the retrospective interviews.

5.5.5 Results

5.5.5.1 Task Completion Time

The medians of the task completion times using the wizard-based and compositional refactorings were 16.5 and 10.5 minutes, respectively. A Wilcoxon signed-rank test shows that there is a significant effect of refactoring tool on the task completion time ($W = 41, Z = 2.25, p = 0.02 < 0.05, r = 0.50$, two-tailed).

Two participants did not finish the refactoring task using either tools during the allotted time. Another participant could not finish the task using the wizards. We excluded these three participants from our significance test. One participant finished the refactoring two minutes faster using the wizards. The other nine participants finished the task faster using the compositional paradigm.

5.5.5.2 Correctness

Participants were more likely to complete the task correctly in the compositional paradigm than the wizard-based one. Seven participants introduced accidental changes to the code base using the wizard-based refactorings, while only one participant left the refactoring incomplete using the compositional paradigm.

The Extract Superclass refactoring wizard has an option called “Use the extracted class where possible”, which is checked by default. This option causes the tool to replace all occurrences of the selected classes by the superclass whenever this replacement does not introduce any compilation problems. The Pull Up refactoring wizard has a similar option. Only three participants unchecked this option on the wizard and only one participant noticed the unexpected changes in the preview and deselected them. The other
Table 5.4: Number of participants of the lab study who preferred each paradigm of refactoring (the first two rows) with respect to each quality (columns). The last row lists the number of participants with no preference.

<table>
<thead>
<tr>
<th>paradigm</th>
<th>control</th>
<th>correctness</th>
<th>ease of learning</th>
<th>ease of use</th>
<th>opportunity to use</th>
<th>satisfaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>compositional</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>wizard-based</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>no preference</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

seven participants were surprised when they discovered unexpected references to the new superclass at the end. At that point, it was difficult to revert the unexpected changes because the participants had already changed the code too much since the application of the wizard-based refactoring. Two participants failed to finish the task using wizards because reverting the unwanted changes was too time-consuming for them.

5.5.5.3 Qualitative

The majority of our participants were more satisfied with the compositional paradigm, found it easier to learn and use, felt more control and confidence over the refactoring, and expected more opportunities for using it in their code (Table 5.4).

During the interviews, we asked about the advantages and disadvantages of the two paradigms of refactoring. The following presents the themes that we extracted from the participants’ responses.

Control. Participants felt they had more control in the compositional paradigm, because the steps are small, predictable, and mimic their manual refactorings. One participant said:

The wizard gives this illusion of just doing everything for you. […] The downside is that there were a number of options that I read and didn’t quite make sense of, and said I guess I don’t have to care about that. And, of course, I found my sorrow that that wasn’t true. It did things that I completely didn’t expect. […] And, it doesn’t give control.

On the other hand, one participant attributed his feeling of control in the wizard-based paradigm to the previews.

Correctness. A participant said:
The thing that I like about it [compositional paradigm] is that you're taking actions yourself. So, when you see an error, you usually have an idea of which action that you took caused the error.

Another participant explained why he did not trust the correctness of compositional refactorings as much as the wizard-based ones as follows:

I was not sure if it [the compositional refactoring tool] was seeing the full picture of the changes. Since it was stepwise [and] I'm doing [each step] one by one, I'm not sure if each of the steps is going to be integrated correctly.

In practice, the participants were more likely to refactor incorrectly using wizards.

**Change Review.** Seven participants preferred the tool to inform them about the effects of an automated refactoring on their code. However, most participants did not inspect the previews of wizard-based refactorings. As a result, seven participants did not catch unintended changes of the wizard-based refactorings in their previews. Four participants mentioned that the previews were too cluttered. One participant said that previews are good for beginners who want to learn a new refactoring wizard. Our prior study showed that programmers rarely preview their refactorings in practice (Section 3.3.5.2).

**Multi-selections.** Four liked the wizard’s ability to refactor multiple entities at the same time. During the study, five participants tried to use the Extract Superclass wizard to extract a common superclass from two classes at once. The rest either found it easier to refactor in smaller steps or did not notice the configuration option to extract from multiple classes.

**Configuration Options and Error Messages.** Although the wizard-based refactorings provide many options to customize the refactorings, the participants only used a subset of these options. Because one of the methods that the participants had to move to the superclass referenced other members of the subclass, the refactoring wizard reported an error message. The refactoring wizard has an “Add Required” button that when pressed selects all members that are referenced by the currently selected members. However, none of the participants used this configuration option of the wizard. Instead, they performed the refactoring and fixed the resulting compilation problem manually. One of the participants said that he ignored the error message of the refactoring wizard because it was not actionable:

It [The refactoring wizard] came up with something like: “Sorry, this method is referring to this other variable that we can't change”. I didn’t know what I could do about that in the window. I was like: “OK. Thanks for the information!” [laughter]
These observations are consistent with the results of prior studies that showed programmers rarely configure the refactoring wizards \cite{121} and usually apply an automated refactoring that has reported problems (Section 3.4.1).

**Composition Order.** Three participants mentioned that one has to be careful with the order in which he composes the refactorings. On the other hand, one participant indicated that sometimes significant work is required to transform the code to a state that is amenable to the application of a wizard-based refactoring.

### 5.5.6 Design Suggestions

The participants suggested improvements to the compositional and wizard-based paradigms.

For the compositional paradigm, two participants proposed that the tool suggests the related entities that the programmer might want to refactor next.

For the wizard-based paradigm, two participants suggested the ability to match up similar entities. For instance, the Extract Superclass refactoring could detect similar members in multiple classes, or let the programmer match up the related members and pull them up to the superclass in one step. In addition, one participant proposed that the refactoring wizard provides an *incremental preview*. An incremental preview gets updated as the programmer manipulates the configuration options. Finally, one participant suggested that the tool presents the previews graphically.

### 5.6 Limitations

Like any study, each of our prior field study (Chapter 3), inference of refactoring sets (Section 5.1) and composition patterns (Section 5.2), survey (Section 5.4) and lab study (Section 5.5) has its own limitations. However, their results with respect to the effectiveness of compositional refactoring corroborate one another. The rest of this section discusses some of the limitations of our work.

#### 5.6.1 Eclipse Foundation Data Set

The Eclipse foundation data (Section 5.1.1), while huge, lacks precision. For instance, it does not differentiate certain refactorings, e.g., Inline Local Variable and Inline Method. In addition, this data
set does not include the project and workspace in which the refactoring is invoked. Moreover, it misses refactorings invoked through Quick Assist. This data set does not distinguish the different variants of certain refactorings that may have different composability patterns. For instance, it records Inline Local Variable and Inline Method as Inline. The data set also does not differentiate between different projects or workspaces of the same user, which makes it hard to decide whether refactorings invoked within close temporal proximities are indeed from the same project. Another shortcoming of this data set is that it does not capture all refactorings invoked through Quick Assist. Despite these limitations, the Eclipse foundation data serves as a good starting point to quantify the prevalence of frequent refactoring sets (Section 5.1). We address some of these limitations through the richer Illinois data. The Illinois data set contains contextual information, e.g., code snippets, configuration options, etc. for each refactoring allowing us to draw more meaningful conclusions about the composition of various refactorings.

5.6.2 Participants

The Illinois data set, while more precise, comes from a smaller pool of participants. We found it challenging to recruit a larger group of experienced programmers due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. Nonetheless, our demographic survey shows that our participants come from diverse backgrounds, have various levels of experiences, and work on a variety of nontrivial projects. Thus, we believe that our participants are representative of real-world programmers and their refactoring behaviors.

The majority of the lab study participants were very familiar with Eclipse. It is possible that novices who are not ready to take much control prefer the wizard-based design. Further studies are needed to understand the effect of experience on the preferred paradigm of refactoring.

5.6.3 Manual Refactorings

The refactoring data that we analyzed did not contain manual refactorings. Thus, we only inferred refactorings sets (Section 5.1) and composition patterns (Section 5.2) of automated refactorings. However, our analysis of these data sets provides enough evidence for the frequency and rationales of composing automated refactorings. Including manual refactorings will only provide additional support for the naturalness of the practice of composing refactorings.
Moreover, Negara et al. showed that over 50% of the common refactorings, e.g., Rename, Extract Method, Inline, from which we draw our conclusions are indeed performed using automated tools [125].

5.6.4 Generalizability

Due the constraints of the survey and lab studies, we evaluated the compositional paradigm using two refactorings, i.e., Extract Superclass and Extract Composite. We have implemented refactorings such as Pull Up, Push Down, and Extract Interface in the compositional paradigm. More evaluation is left to future work.

Our data sets were limited to the use of the Eclipse refactoring tool for Java. However, we expect our results to hold for similar refactoring tools, because they follow a similar user interaction model, i.e., wizard-based refactoring.

5.6.5 Survey Study Refactoring Task

The survey participants did not try the tools and their evaluations were merely based on the screenshots of the wizard-based and compositional refactorings screenshots. However, the wizard-based and compositional refactorings are based on familiar features of Eclipse, i.e., wizards and Quick Assist. The insightful comments of the survey participants indicate that they understood the two paradigms well.

The survey study provided us with early feedback on our design. We improved and implemented the tool based on the survey results and got similar feedback from the participants of the lab study who tried both tools for a real refactoring task.

The survey study demonstrated the two paradigms of refactoring using a small piece of code. Several features of the wizard-based refactoring such as extracting from multiple classes, computing the required dependencies, and using the new superclass wherever possible were not applicable to that refactoring task. Our goal was to keep the survey simple and make it accessible to programmers who may not be familiar with all configuration options of the wizard. Therefore, we used a small code snippet that still enabled us to compare the overall interaction models of the two interfaces. We intentionally kept the survey simple to make it understandable for programmers who may not be familiar with the intricacies of the refactoring wizards. Moreover, most configuration options of the wizard could be simulated by refactoring compositions. For example, a programmer could compose his Extract Superclass refactoring
with the Use Supertype Where Possible refactoring to simulate the “Use the extracted class where possible” option of the wizard. We consider some loss of functionality in the compositional design an acceptable trade-off for gaining simplicity and naturalness.

5.6.6 Lab Study Refactoring Task

For the lab study, we selected a single realistic refactoring task that Kerievsky used in his book to introduce the Extract Composite refactoring [95, p. 214]. As some of our participants speculated, the wizard-based refactoring might be more appropriate for refactorings that affect hundreds of files. Further studies are needed to compare the two paradigms of refactoring on a variety of refactoring tasks.

5.6.7 Carryover Effect

Each participant of the lab study performed the refactoring task using both wizard-based and compositional refactorings. The order in which a participant tries the tools may affect his evaluation. However, we expected personal differences, e.g., experience and preferences, to impact the results. Thus, we decided that quantitative measures alone, e.g., task completion time and correctness, were not sufficient to reliably compare the two paradigms. Therefore, we instructed each participant to perform the refactoring task using both tools and compare the two methods qualitatively. To mitigate the learning effect we used the standard technique of randomizing the order of treatments.

5.6.8 Participant Response Bias

A common limitation of user studies is that participants may favor the interface that they think the researcher has developed. However, we think that our results are less affected by this bias, because most of our participants could not tell which interface was ours. At the end of the lab study, most participants asked us which interface was ours. This is because the Extract Superclass wizard is rarely used (Chapter 3) [121], and few programmers remember all Quick Assist actions to identify the actions contributed by our plug-in.
Chapter 6

A Compositional Type Qualifier Inference Tool

We showed how the compositional paradigm improves the usability of classic refactorings. We next use the compositional paradigm to automate even more complex program transformations, which are not supported by today’s program transformation tools. One class of such program transformations is type qualifier inference (TQI), which is inferring the annotations required by a pluggable type system. A pluggable type system augments an existing type system with type qualifiers to check more properties. Type qualifiers are the annotations that the users of a pluggable type system write for declarations and uses of types. A program can use multiple pluggable type systems to check different properties.

Type inference and refactoring are usually thought of as unrelated concepts. However, we consider TQI a refactoring, because inserting type qualifiers to a program does not alter the behavior of the program. In addition, TQI often goes beyond inserting type qualifiers and requires code refactorings. This insight led us to automate TQI in the compositional paradigm. We chose to automate TQI in the compositional paradigm for several reasons. First, the different nature of TQI from classic refactorings makes it a good candidate to show the generality of the compositional paradigm. Second, our analysis of the existing TQI systems shows that they suffer from problems similar to those of the wizard-based refactorings, suggesting that compositional refactoring is likely to yield a better automation for TQI. Third, the compositional paradigm enables us to achieve a universality that existing TQI systems lack. It enables us to build a TQI system out of a type qualifier checker. Finally, Java’s recent support of type qualifiers makes them accessible to mainstream programmers and the demand for a usable and universal TQI urgent.

Over the past few decades, researchers have developed many pluggable type systems to check properties such as safe use of locks and units and safety against null dereferences and SQL injection. Recently, Java 8 extended the language to support the syntactic features required by pluggable type systems [26].
The benefits of a pluggable type system come at the cost of adding the type qualifiers to the code. The burden of annotating code with type qualifiers is a major obstacle to the wide adoption of pluggable type systems. Realizing the burden of adding type qualifiers, researchers have developed a myriad of type qualifier inference (TQI) systems [40, 55, 64, 67, 68, 81, 88, 89–91, 105, 124, 137, 149, 150, 153]. These TQI systems employ static analysis, dynamic analysis, or a combination of the two.

Existing TQI systems operate in batch mode. That is, they take the source code as input, analyze it, and insert all the type qualifiers at once. While batch TQI systems can add type qualifiers to large pieces of code without involving the programmer, these systems have several weaknesses. First, they are imprecise, because inferring precise type qualifiers requires an accurate view of the runtime behavior of the program, e.g., invariants, aliasing, control and interprocedural data flow. Second, they are rigid. That is, they assume that the code does not have to be changed, while usually programmers have to refactor the code to a form that the pluggable type system can express. Third, they are unpredictable, because they make many code changes and it is difficult for programmers to tell where and why a code change was applied. Fourth, they are specific to a single pluggable type system. While frameworks such the Checker Framework [130] reduce the cost of developing a checker for a pluggable type system, developing a TQI system for a new pluggable type system is still a nontrivial programming task.

This chapter introduces CASCADE, a universal TQI system. CASCADE takes a checker for a pluggable type system as input and produces a TQI system. It achieves universality through a combination of two concepts: compositional refactoring and speculative analysis.

Following the compositional paradigm (Chapter 5), CASCADE decomposes TQI into primitive changes. In this case, each primitive change resolves one or more of the errors reported by the type qualifier checker.

Speculative analysis [50, 122, 123] is a technique that assists programmers in decision making by presenting the consequences of their actions ahead of time. Applications of speculative analysis suggest that it can improve programmers' productivity by showing the future states of the system and helping programmers avoid undesirable states and unnecessary backtracking.

CASCADE is an interactive system that assists programmers in adding type qualifiers by guiding them through a tree of changes and error messages. It runs the checker on a copy of the source code and turns the error messages into the root nodes of the tree. Next, it recursively computes the children of each
tree node as follows. For each error message, it computes a change that would fix that error message and makes the change a child of the tree node corresponding to the error message. For each change, it applies the change on a copy of the source code and runs the checker on the resulting code to get the set of new error messages. Then, it makes the new error messages children of the change tree node.

The compositional aspect of CASCADE is computing the change to fix each error message. The speculative aspect of CASCADE is computing the effect of each change on the error messages reported by the checker and the required follow-up changes.

We conducted a lab study with 12 programmers to compare CASCADE with JULIA. JULIA [149,150] is the state of the art static analysis tool for inferring nullness qualifiers. We asked each participant to use both CASCADE and JULIA to insert nullness type qualifiers into two programs. We balanced the ordering of the tools and programs across the participants. The quantitative results show that the participants finished the task faster with CASCADE and inserted better type qualifiers. The qualitative results show that the participants felt more in control with CASCADE. They also indicated that they are more likely to use CASCADE in the future.

This chapter makes the following research contributions:

- It introduces CASCADE, which is the first universal TQI system. CASCADE achieves universality by reusing an existing type qualifier checker. CASCADE is open source and publicly available [16].
- It compares CASCADE, an interactive TQI system, with JULIA, a state-of-the-art batch TQI system, through a user study. To the best of our knowledge, this is the first user study on type qualifier inference.

### 6.1 Type Qualifiers

Type qualifiers are light-weight specifications that augment an existing type system to enable static verification of desired properties of software.

Pierce [135] defines a type system as follows:

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
There is a long history of debate on static vs. dynamic typing [85]. Advocates of static typing argue that static typing facilitates early error detection, abstraction, documentation, and efficiency. On the other hand, proponents of dynamic typing prefer it for higher expressiveness and conciseness, arguing that the kind of errors found by static types are easy to catch by unit tests. Many have attempted to combine the benefits of static and dynamic typing [45,48,49,80,92,147].

Pluggable type systems [48] are a way of bringing some of the benefits of dynamic typing to static typing. Mainstream programming languages support mandatory type systems, requiring that a program fully typechecks by a type system. On the contrary, optional type systems relax the notion of a type system in two ways:

1. An optional type system has no effect on the behavior of the program.
2. An optional type system does not mandate syntactic type annotations.

Optional type systems enable pluggable type systems. Programmers can enable different sets of optional type systems during various phases of software development.

A pluggable type system can be represented as a set of type qualifiers. For example, one can add the type qualifier `@NonNull` to the type of a `String` variable `s` in Java—written as `@NonNull String s`—to indicate that the variable `s` cannot be null. The nullness checker would ensure that every reference is dereferenced only when it is provably `@NonNull`.

Limited forms of type qualifiers exist in most programming languages such as Scheme, C, and Java. The `const` (C) and `volatile` (Java) keywords are examples of type qualifiers. Type qualifiers have been proposed for verifying a wide range of properties including null safety, tainting, regular expressions, immutability, and typestates.

Researchers have proposed frameworks for building custom pluggable type systems, e.g., CQual [73], Clarity [55], JQual [81], JavaCOP [36], and the Checker Framework [130]. A wide variety of pluggable type systems have been developed in these frameworks for checking software properties such as null safety, tainting, internationalization, ownership, and immutability.

Java 8 is the first version of Java that supports type qualifiers [26]. Java 5 introduced type annotations for certain program locations such as class, method, and variable declarations. Java 8 permits the type annotations to appear in more locations, i.e., anywhere that a type may appear including generics and
Table 6.1: Existing type qualifier inference tools.

<table>
<thead>
<tr>
<th>Pluggable Type System</th>
<th>Prog. Languages</th>
<th>Type Qualifier Inference Tools</th>
<th>Theoretical Soundness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nullness</td>
<td>Java</td>
<td>JastAddJ Nullness [67]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nit [90, 91]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SALSA Nullness [105]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Xylem [124]</td>
<td>Unsound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Julia [149, 150]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>Clarity Nullness [55]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td>Java, C, Perl, C#, C++</td>
<td>Daikon Nullness [68]</td>
<td>Unsound</td>
</tr>
<tr>
<td>Immutability</td>
<td>Java</td>
<td>JQual Immutability [81]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Javarifier [137] for Javari [153]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pidasai [40]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelInfer [89] for RelIm [89]</td>
<td>Sound</td>
</tr>
<tr>
<td>Ownership</td>
<td>Java</td>
<td>Inferring Universe Types [64]</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inferring Ownership [88]</td>
<td>Sound</td>
</tr>
</tbody>
</table>

casts. The Checker Framework [130] provides definitions and verifiers for over a dozen sets of type qualifiers compatible with Java 8.

Despite the advances in formalizing, standardizing, and verifying custom type qualifiers, there remains a major barrier to their adoption: annotation burden. The annotation burden refers to the cost of adding type annotations to an existing piece of code so that it typechecks according to the rules of a given type system. Frameworks for developing pluggable type systems make it easy to build many pluggable type systems. However, the annotation burden is still a barrier to a wide adoption of these pluggable type systems. To mitigate the annotation burden, researchers have proposed tools for inferring the type qualifiers of a few pluggable type systems. Table 6.1 lists some of the proposed type qualifier inference (TQI) tools.

### 6.2 The Illusion of Fully Automatic Type Qualifier Inference

Developing type qualifier inference (TQI) tools is an active area of research (Table 6.1). Typically, these tools operate in _batch_ mode. That is, they take an unannotated piece of code as input and infer type qualifiers. The goal is to infer a set of type qualifiers that the checker accepts.
The research on TQI has improved the accuracy of TQI tools over time. Usually, multiple sets of type qualifiers are possible for a given piece of code. For instance, one can replace a more specific type qualifier (e.g., @NonNull) by a more general qualifier (e.g., @Nullable) and suppress the resulting errors. As another example, when the program does not type check due to some type qualifier incompatibility, there are usually different ways of resolving the incompatibility. Typically, TQI tools aim to infer the “best” type qualifiers according to some objective function. Such an objective function prefers for each qualifier location the more specific type qualifiers over the less specific, more conservative ones.

Through several code examples from a benchmark suite, we show that current TQI tools cannot infer type qualifiers as accurately as programmers can. Moreover, we argue that a fully automatic solution that reaches the accuracy of programmers is an elusive if not an impossible goal. This argument is grounded in our experiment with three recent TQI tools for nullness: JastAddJ Nullness [67] (Subversion revision 9127), Nit [90, 91] (v0.5d), and JULIA [149, 150] (March 2013 release). We ran each of these tools on the ten Java programs of the JOlden suite (Table 6.2). Next, we compared the type qualifiers that the three TQI tools inferred for each program and took the most accurate ones for the program. Finally, we manually changed some of the inferred type qualifiers to reach a higher level of accuracy. We found that JULIA infers more accurate type qualifiers than Nit and JastAddJ Nullness. Thus, in the rest of this section we discuss the inaccuracies of the type qualifiers that JULIA infers. JULIA achieves this higher accuracy partially due to its better flow and interprocedural analyses and analyses of array elements. In addition, we found JULIA to be more reliable and more actively maintained than the other two TQI tools. Thus, we chose to compare our tool, CASCADE, with JULIA.

Although the existing TQI tools are able to infer many type qualifiers accurately, we were often able to improve the accuracy of the inferred type qualifiers manually. However, we found this manual process to be tedious, which required understanding, changing, and annotating the code. This suggests that the inaccuracy of batch TQI tools leaves the programmers with significant annotation burden.

The rest of this section outlines some of the coding idioms that we found to contribute to the inaccuracy of the inferred type qualifiers. These idioms reveal either limitations in the TQI tools or the type systems. This implies that making the TQI tools more accurate can only bring about limited gains because of the limited expressiveness of type systems. Much of the research on type systems is devoted to devising more expressive type systems, which can support more coding idioms. However,
Table 6.2: The list of JOlden programs. SLOC stands for source lines of code.

<table>
<thead>
<tr>
<th>Program</th>
<th># of SLOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes-Hut</td>
<td>624</td>
<td>Barnes-Hut, a hierarchical force-calculation algorithm</td>
</tr>
<tr>
<td>BiSort</td>
<td>191</td>
<td>adaptive bitonic sorting</td>
</tr>
<tr>
<td>Em3d</td>
<td>228</td>
<td>modeling the propagation of electromagnetic waves through objects in 3 dimensions</td>
</tr>
<tr>
<td>Health</td>
<td>345</td>
<td>simulating the Columbian health-care system</td>
</tr>
<tr>
<td>MST</td>
<td>276</td>
<td>Bentley's algorithm for finding the minimum spanning tree of a graph</td>
</tr>
<tr>
<td>Perimeter</td>
<td>303</td>
<td>computation of the total perimeter of a region in a binary image represented by a quadtree</td>
</tr>
<tr>
<td>Power</td>
<td>383</td>
<td>decentralized optimal power pricing</td>
</tr>
<tr>
<td>TreeAdd</td>
<td>97</td>
<td>a recursive depth first traversal of a binary tree</td>
</tr>
<tr>
<td>TSP</td>
<td>323</td>
<td>the traveling salesman problem</td>
</tr>
<tr>
<td>Voronoi</td>
<td>620</td>
<td>computing a Voronoi diagram for a set of points</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,390</strong></td>
<td></td>
</tr>
</tbody>
</table>

more expressive type systems tend to be more sophisticated and difficult to build an accurate TQI tool for. In Section 6.3, we propose an approach to combine the efficiency of TQI tools with the insight of programmers.

### 6.2.1 Blame Assignment

Batch TQI tools tend to perform poorly on a program that is not type checkable as it is. Any set of type qualifiers that a TQI tool infers for such a program will cause the checker to report some error. In such cases, one has to identify parts of the code that need to be changed to make the code type check, a process that we refer to blame assignment. TQI tools are not good at blame assignment because it requires a deep insight about the program and changing the code. Figure 6.1 shows a slice of the source code of Barnes-Hut in the JOlden suite of programs. JULIA, a batch TQI tool, infers weaker type qualifiers than what a programmer can achieve by slightly changing the code. Although the body of method Node.oldSubindex expects that the parameter ic be @NonNull (line 44), the tool infers @Nullable as the type qualifier of ic (line 41). This is because the tool correctly infers that the return type of Tree.intcoord is @Nullable (line 21), and the return value of Tree.intcoord is passed to Node.oldSubindex as the value of parameter ic (lines 10, 14, 37). However, a better approach would be to make xpic (line 36) and ic (line 41) @NonNull and either guard against the nullness of xqic on line 10 or insert an assert statement to document the fact that the particular method invocation at line 10 never returns null.
class Tree {
  //...
  @Nullable Node root;

  void makeTree(int nstep) {
    for (Enumeration e = bodiesRev(); e.hasMoreElements();) {
      Body q = (Body) e.nextElement();
      if (q.mass != 0.0) {
        q.expandBox(this, nstep);
        MathVector xqic = intcoord(q.pos);
        if (root == null) {
          root = q;
        } else {
          root = root.loadTree(q, xqic, Node.IMAX >> 1, this);
        }
      }
    }
    root.hackcofm();
  }

  @Nullable MathVector intcoord(MathVector vp) {
    MathVector xp = new MathVector();
    //...
    xsc = (vp.value(2) - rmin.value(2)) / rsize;
    if (0.0 <= xsc && xsc < 1.0) {
      xp.value(2, Math.floor(Node.IMAX * xsc));
    } else {
      return null;
    }
    return xp;
  }

  class Node {
    //...
    Node loadTree(Body p, @Nullable MathVector xpic, int l, Tree tree) {
      int si = oldSubindex(xpic, l);
      //...
    }

    static int oldSubindex(@Nullable MathVector ic, int l) {
      int i = 0;
      for (int k = 0; k < MathVector.NDIM; k++) {
        if (((int) ic.value(k) & l) != 0) i += Cell.NSUB >> (k + 1);
      }
      return i;
    }
  }

  Figure 6.1: Batch TQI tools do not infer proper type qualifiers for a program that cannot be type checked in its current form.
```java
class Value {
    private @Nullable Value left;
    private @Nullable Value right;
    //...
    int bisort(int spr_val, boolean direction) {
        if (left == null) {
            //...
        } else {
            int val = value;
            value = left.bisort(val, direction);
            boolean ndir = !direction;
            spr_val = right.bisort(spr_val, ndir);
            spr_val = bimerge(spr_val, direction);
        }
        return spr_val;
    }
}
```

Figure 6.2: The type qualifiers that JULIA infers result in errors because it cannot infer that `left` and `right` have the same nullness properties.

### 6.2.2 Invariants

Batch TQI tools infer poor type qualifiers when they fail to infer some program invariants. Figure 6.2 shows a method, `Value.bisort`, from the BiSort program of the JOlden suite of programs. JULIA, which represents a batch TQI tool, does not infer any annotations for method `Value.bisort`. As a result, the nullness checker reports the following problem at line 12:

```java
dereference of possibly-null reference right
```

This is because JULIA cannot infer that the tree is balanced and `left` and `right` have the same nullness properties. Inserting the following statement before line 12 will resolve the above problem:

```java
assert right != null : "@SuppressWarnings(nullness)";
```

Some invariants are more complex and more difficult to detect automatically. Figure 6.3 shows one such invariant from the TSP program of the JOlden suite. JULIA infers the variable `a` as @Nullable on line 13. Thus, it infers @Nullable for the parameter of method `Tree.distance` (line 4). Had JULIA noticed that the for loop (lines 12–18) iterates over a cycle, it would have inferred `a` (line 13) as @NonNull.
class Tree {
  private @Nullable Tree next;
  //...
  double distance(@Nullable Tree b) {
    return (Math.sqrt((x - b.x) * (x - b.x) + (y - b.y) * (y - b.y)));
  }
  
  Tree merge(Tree a, Tree b) {
    Tree min = a;
    double minDist = distance(a);
    Tree tmp = a;
    for (a = a.next; a != tmp; a = a.next) {
      double test = distance(a);
      if (test < minDist) {
        minDist = test;
        min = a;
      }
    }
    //...
  }
}

Figure 6.3: This figure demonstrates how JULIA infers unnecessary @Nullable qualifiers because of not recognizing a cyclic linked list.

6.2.3 Full Arrays

TQI tools tend to be inaccurate in inferring the type qualifiers of array elements. Figure 6.4 shows a snippet of the Em3D program of the JOlden suite. JULIA infers two unnecessary @Nullable qualifiers (lines 2 and 15). It infers @Nullable for the return type of method Node.fillTable, because it cannot prove that all elements of the returned array are @NonNull. Since the return value of Node.fillTable (line 32) is passed to Node.makeUniqueNeighbors (line 36), JULIA infers @Nullable for the parameter of method Node.makeUniqueNeighbors. However, these two @Nullable qualifiers make the code fail to type check, because both Node.makeUniqueNeighbors (line 20) and BiGraph.create (line 34) expect a non-null array of non-null elements.

Figure 6.5 shows a piece of code from the MST program of the JOlden suite. This example shows that JULIA infers inaccurate qualifiers because it does not detect a full array. Although the constructor (lines 6–11) fully initializes the array Graph.nodes, JULIA infers the type of Graph.nodes as @Nullable Vertex[]. As a result, the return value of Graph.firstNode is also inferred as @Nullable. However, both

1@Nullable Node[] is equivalent to @Nullable Node @NonNull[], which indicates a non-null array of possibly-null elements.
null

Figure 6.4: JULIA infers unnecessary @Nullable qualifiers (lines 2 and 15) because it cannot detect full arrays accurately.

of these qualifiers should be removed because the constructor fully initializes the array Graph.nodes. The cascade effect of the inferred qualifiers affects Hashtable as well. Since Graph.addEdges passes (line 12) elements of nodes to Hashtable.put, JULIA infers @Nullable for the first parameter of Hashtable.put. Similarly, because Hashtable.put passes (line 32) its first parameter, key, to Hashtable.hashMap, JULIA
Figure 6.5: JULIA infers unnecessary Nullable qualifiers (lines 3 and 15) because it cannot detect the full array created in the constructor.

infers the type qualifier Nullable for the first parameter of Hashtable.hashMap. This example shows that the ripple effect of an inaccurate qualifier goes beyond a single class.
6.2.4 Optional Fields

We refer to those fields of a superclass that are possibly-null in the superclass and some of its subclasses as optional fields. JULIA infers @Nullable qualifiers for optional fields. However, a refactoring that moves the optional fields to the classes that require the fields avoids unnecessary @Nullable qualifiers.

The Perimeter program of the JOlden suite shows that moving @Nullable fields of a superclass to its subclasses in which the fields are @NonNull reduces the number of required @Nullable qualifiers and false alarms of the nullness checker. This is another example in which refactoring the code enables the nullness type system provide stronger guarantees about the null safety of the program.

Figure 6.6a shows part of the Perimeter program. QuadTreeNode is a superclass of three other classes: BlackNode, WhiteNode, and GreyNode. QuadTreeNode declares four fields nw, ne, sw, and se, which have to be annotated as @Nullable, because they are null in instances of BlackNode and WhiteNode. Consequently, JULIA infers @Nullable for the return types of the four getter methods of nw, ne, sw, and se in QuadTreeNode.

Figure 6.6b shows how moving the @Nullable fields of QuadTreeNode to GreyNode reduces the number of @Nullable qualifiers and false alarms of the nullness checker.

6.2.5 Flow Analysis

Inaccurate control and data flow analyses cause TQI tools to infer inaccurate type qualifiers. Figure 6.7a shows one of the two pieces of code in the Power program that exhibit this weakness of JULIA. JULIA infers variable a1 (line 14) as @Nullable, because it does not infer that a1 and next_lateral have the same nullness properties. As a result, JULIA infers @Nullable for Demand.add (line 24). This @Nullable qualifier makes the nullness checker report an error due to dereferences of a1 in Demand.add (lines 26 and 27).

Figure 6.7b illustrates our refactored version of the code in Figure 6.7a with a simpler flow. The nullness checker can successfully analyze this flow and prove that a1 is not null at line 10.
(a) `@Nullable` fields of a superclass make Julia infer unnecessary `@Nullable` qualifiers.

(b) Moving the optional fields to subclasses avoids unnecessary `@Nullable` qualifiers.

Figure 6.6: Moving the `@Nullable` fields of `QuadTreeNode` to its subclass in which these fields are `@NonNull`, reduces the number of `@Nullable` qualifiers and the false positives reported by the nullness checker.

6.3 Design Goals

Despite the excellent work on inferring type qualifiers in batch mode (Table 6.1), annotation burden still remains a major obstacle to the adoption of type qualifiers for four main reasons. First, although the inference algorithms infer the type qualifiers according to some objective measure of goodness, the results are often imprecise and likely to differ from what programmers would expect. Inferring precise type qualifiers is not merely a mechanical task. Often, the programmer has to reason about the code and refactor it to write precise type qualifiers. For instance, a program may not be type checkable as it is, and the programmer may have to rewrite a part of it in a form that is amenable to automatic type qualifier checking. Without such a reasoning and rewriting by the programmer, the inference tool may infer overly conservative type qualifiers (Section 6.2). Second, they are rigid. That is, they assume
(a) A complex program flow causes JULIA to infer `@Nullable` qualifiers unnecessarily.

(b) The code refactored to use a simpler flow. JULIA and the nullness checker can analyze this simpler flow accurately.

Figure 6.7: The impact of complex control and data flows on the accuracy of a batch TQI tool.

that the code does not have to be changed, while programmers usually have to refactor the code to a form that the pluggable type system can express. Third, the changes inferred by a batch TQI tool are scattered across many files and unpredictable, similar to automated refactorings that make complex changes (Section 3.3.5). Fourth, each existing batch TQI tool is specific to a single pluggable type system. The cost of developing a TQI tool for a new set of type qualifiers is high.

Without addressing the above problems, TQI is likely to become yet another automation technology that remains grossly underused by programmers. Based on our studies on compositional and wizard-based paradigms of refactoring, we hypothesize that the underlying problem is once again too much automation.
Our analyses of the weaknesses of existing batch TQI tools and wizard-based automated refactorings led us to establish the following goals for the design of CASCADE, our TQI tool.

**Universality.** We aim to build a universal TQI tool. Instead of building a TQI tool for each set of type qualifiers from scratch, such a system can use the existing type qualifier checker to assist programmers in inferring the type qualifiers.

**Predictability.** To increase the chances of adoption, a TQI tool has to be predictable. That is, it has to communicate where, how, and why it has changed each part of the code.

**Control.** Inferring type qualifiers such that they are accurate, pass the type qualifier checker, and also appeal to the subjective criteria of programmers is challenging. This problem is likely to remain open for a long time. Meanwhile, a usable TQI tool should allow the programmers to control the inference process and override the tool’s suggestions.

**Flexibility.** In practice, adding precise type qualifiers requires code refactorings. Assuming that no code refactoring is needed or all refactorings are performed before using the TQI tool makes the TQI tool rigid. A desirable TQI tool should be flexible and allow the programmers to refactor the code during the TQI process.

### 6.4 Design Process

We followed an iterative user-centered process for designing CASCADE, our TQI tool. We started by creating four different low-fidelity paper prototypes. Our goal was to explore a wide design space and get early feedback on them. Based on our design goals and the feedback that we received, we selected two of the low-fidelity prototypes and turned them into high-fidelity prototypes. We implemented the high-fidelity prototypes as functional and interactive Eclipse plug-ins. Both of these prototypes incorporate the ideas of compositional refactoring and speculative analysis. The differences are in the user interaction models. Overall, we sought feedback from nine participants on the prototypes. The following presents the two high-fidelity prototypes that we developed.
6.4.1 Change-centric View

Figure 6.8 shows a screenshot of the change-centric view, a prototype user interaction model that we developed for our TQI tool. The change-centric view presents an interactive tree of changes and errors. The changes are the tool’s suggestions for resolving the errors reported by the checker. Each change will insert some type qualifier to resolve a type qualifier incompatibility error. The tool allows the programmer to expand the changes to see the errors that the changes resolve (green check marks), the new errors that the changes introduce (light bulbs), and the changes that will resolve the new errors. In the change-centric view, errors cannot be expanded. The programmer can double-click on each change in the tree to apply it on the code and expand the tree to examine the consequences of the changes.

The pilot study of this prototype revealed three major weaknesses of this design. First, the tree is cluttered with no visual boundaries between changes and errors. Second, finding the relationship between errors and the changes that resolve the errors is not easy. For example, most errors are shown twice once as a child of the change that introduces the errors and another time as a child of the change that resolves the errors. This duplication makes the mapping between the errors and the changes that resolve the errors confusing. Finally, the participants indicated that an error-centric view would be more intuitive than a change-centric view. An error-centric view first shows the errors and then the changes that would resolve the errors. Showing the errors first makes it clear why the changes are needed.

6.4.2 Separated Views

Our second high-fidelity prototype of a user interaction model for TQI separated the errors and changes into separated views (Figure 6.9). Similar to the change-centric view, the changes resolve type incompatibility errors and the errors show the impact of each change on the set of errors that the checker reports. The goal was to mitigate some of the shortcomings of the change-centric view, while experimenting with a significantly different design. The separated view shows a tree of only changes and a list of only errors. The prototype links the tree and the list through several interactive features. For example, selecting a change will highlight the errors that are resolved (green errors) and introduced (read errors) by the change. On the other hand, selecting an error, will highlight the change that resolves the error.

Our pilot study on this prototype showed that the participants appreciated that this prototype reduces the clutter in the tree. However, participants raised two main issues with the prototype. First, the
prototype takes significant space on the screen, limiting the space that is left to the code editor and the rest of the IDE. Second, separating the changes and errors increases makes it more costly for the programmers to identify the related changes and errors.

6.5 The Design of CASCADE

Our design goals (Section 6.3) and prototyping influenced the design of CASCADE. We use a combination of two concepts, *compositional refactoring* and *speculative analysis*, to build the first universal TQI tool and meet the other design goals that we established for a desirable TQI tool in the previous section.
Figure 6.9: A screenshot of the separated view, a high-fidelity and interactive prototype TQI tool that presents the errors and changes in separated views.

The compositional paradigm is to decompose a complex refactoring into small steps and to allow the programmer to compose them. We decompose TQI into small steps in which each step resolves an error reported by the type qualifier checker. Of course, each step can create new errors, and a step that resolves one error can resolve others, as well. The most common step in TQI is to propagate the type qualifier of one expression to another to resolve an incompatibility between the type qualifiers of the two sides, e.g., the two sides of an assignment. CASCADE automates only the type qualifier propagation steps. The more difficult steps, such as inserting assert statements, adding annotations other than type qualifiers that the checker supports, and refactoring code are left for the programmer to perform.
manually. CASCADE leaves the programmer in control of the entire inference process, so it is easy for the programmer to insert manual edits. CASCADE uses the type qualifier checker to find places where there are type qualifier incompatibility errors and develops a plan for fixing each error. However, it does not guarantee that these plans will work, and the programmer does not expect them all to work. The programmer will accept a plan when it seems reasonable, and make manual edits when it does not.

CASCADE forces the programmer to look at the entire program whose type qualifiers are being inferred. It might seem that it would take longer to infer type qualifiers this way than it would with a batch TQI tool. However, the batch TQI tools rarely can infer type qualifiers perfectly, and it can take a programmer a long time to figure out why they failed. When a program needs some changes before it will be type correct, CASCADE directs the programmer to the parts that need changing, allowing the programmer to discover them and make the changes.

We use the speculative analysis to show the consequences of applying each change and propose a plan for composing the changes. As a problem solving task, software development involves making decisions. Speculative analysis can help programmers in making these decisions by showing the consequences of the decisions in advance. With speculative analysis, CASCADE makes the programmer aware of the new error messages that the type qualifier checker will report because of applying a change. In addition, CASCADE suggests changes for the new errors that may arise, effectively suggesting a plan for composing multiple changes to eradicate an error reported by the checker.

The CASCADE view consists of a tree (Figure 6.10). The root of the tree is the set of error messages that the nullness checker reports for the selected program.

The CASCADE tree has two types of nodes: error message nodes and change nodes. An error node shows an error message that the type qualifier checker reports. A change node offers the programmer an automated change to apply. An error message node is either a root node or a child of a change node. A change node is a child of an error message node.

CASCADE provides an affordance to expand the tree (Figure 6.11). Although CASCADE follows the conventions of the Eclipse user interface, its interaction semantics is different. Unlike standard Eclipse views that use trees to show hierarchies of program elements, CASCADE uses a tree to show future states of the source code. Expanding an error message shows any change that CASCADE proposes to resolve that error message. For example, if a programmer expands the first error message shown in Figure 6.11,
Figure 6.10: A screenshot of the initial state of the CASCADE tree for the JOlden TreeAdd project. The root nodes of the tree are the error messages that the nullness checker reports for the program. CASCADE alternates the background colors of the tree nodes between white and gray to make it easy to visually distinguish the adjacent tree nodes. The different background colors of the tree nodes do not have any semantic connotation.

The tree will expand to propose a change described as Change field left to @Nullable TreeNode. Similarly, expanding a change shows the new error messages that will appear if the change is applied.

Expanding the CASCADE tree does not change the source code. Rather, it enables the programmer to explore the future states of the source code, and see how a sequence of proposed changes will affect the source code.

Programmers can expand the CASCADE tree deeply. For instance, consider the error message node highlighted in the CASCADE tree shown in Figure 6.12. To resolve this error message, CASCADE proposes the change Change parameter l to @Nullable TreeNode. Besides, CASCADE allows the programmers to expand the proposed change node. Expanding a change node shows the new error messages that will appear if the change is applied. In this case, the error message node that CASCADE shows as a child of the expanded change involves the expression left = l (Figure 6.12). With further expansion of the
Figure 6.11: A screenshot of the CASCADE tree with an expanded error message node. Similar to standard Eclipse tree views, users can expand a node of the CASCADE tree by clicking on the triangle icon besides it. When the user clicks on the triangle icon, the tree will expand to show the children of the tree node. If the user clicks on the triangle icon besides an expanded tree node, the tree will collapse the children of that tree node. The tree shows the child node below the expanded node and indents the child to indicate the parent-child relationship.

above error message, CASCADE proposes the change **Change field left to Nullable TreeNode** to fix the error message.

CASCADE maintains the link between the source code and the nodes on the tree. If the user single-clicks on an error message or change, CASCADE will open and highlight the piece of code related to the selected node.

If the user double-clicks on a change tree node, CASCADE will apply the change on the code and open the affected code in the editor (Figure 6.13). In addition, CASCADE will show the applied change and the error messages that it resolves as disabled nodes. CASCADE provides an undo feature. Undoing a change will cause CASCADE to enable the change and the error messages that it resolves.
Figure 6.12: A screenshot of the CASCADE tree with expanded change nodes. Expanding a change node will show the new error messages that applying the change will introduce. If the change won’t introduce any new errors, the change node won’t expand.

The programmers are free to apply the proposed changes in any order and intersperse them with manual edits. However, these actions may make the tree inconsistent with the source code, in which case the programmers can press the refresh icon to recompute the tree.

6.6 Universality Assumptions

We define a universal TQI tool as a tool that can infer type qualifiers according to the rules of any pluggable type system. Our compositional approach to TQI is universal under two assumptions:

A1. A type qualifier checker for the type qualifiers is available.

A2. It is possible to automatically compute the code changes that would fix some of the problems reported by the type qualifier checker, e.g., by using the information included in the problems that the type qualifier checker reports.
Figure 6.13: CASCADE changes the text colors of the change tree node and the error message tree nodes that the change resolves to gray to indicate that the user applied the change and the error messages are resolved.

CASCADE makes the following assumptions to satisfy the above assumptions:

A1.1. The type qualifiers are compatible with Java 8.

A1.2. A checker for the given set of type qualifiers is implemented in the Checker Framework.

A1.3. The Checker Framework Eclipse plug-in is configured to run the type qualifier checker.

A2.1 Some of the problems that the type qualifier checker reports are type qualifier incompatibilities. Type qualifier incompatibility problems report that the actual type qualifier of an expression is different from the expected type qualifier for that expression. Incompatibilities between the type qualifiers of the expressions of two sides of an assignment, a method argument and method parameter, and a return statement and declared method return type are examples of type qualifier incompatibilities.
A2.2 For each type qualifier incompatibility, the Checker Framework reports the location (enclosing file, offset, and length) of the code snippet that causes the problem as well as the expected and actual type qualifiers.

The next section explains how CASCADE achieves universality under the above assumptions.

6.7 Implementation

Although we demonstrated CASCADE for inferring the nullness type qualifiers (Figures 6.10, 6.11, 6.12, and 6.13), CASCADE is a universal TQI tool and supports the inference of all type qualifiers that come with a checker developed on top of the Checker Framework. CASCADE achieves this universality by leveraging an existing type qualifier checker. Reusing the checker makes the implementation of CASCADE simpler than a typical batch TQI tool.

We implemented CASCADE as an Eclipse plug-in that depends on the Checker Framework Eclipse plug-in, which in turn depends on the Checker Framework. The simplicity of CASCADE makes it easy to port it to other programming environments.

6.7.1 Getting the Checker Error Messages

CASCADE relies on the error messages that the type qualifier checker reports on the given source code and the variants of the source code. To get the error message, CASCADE invokes the checker through the Checker Framework Eclipse plug-in. The Checker Framework runs the checker and populates the Eclipse problems view with markers that capture information about the error messages reported by the checker. Each marker contains information such as the error message, the offset and length of the piece of code that caused the error message, and the expected and actual type qualifiers.

6.7.2 Proposing Changes to Resolve the Error Messages

CASCADE proposes changes to resolve some of the error messages reported by the checker. The error messages that CASCADE proposes a change for are those that report incompatibilities between type qualifiers and contain both the actual and expected type qualifiers. If the type qualifier of the right-hand side of an assignment statement is not a subtype of that of the left-side hand side, the checker will
CASCADE provides quick fixes to resolve the error messages reported by the checker. In this example, the checker reports incompatible types between the method argument and parameter. CASCADE proposes a change to resolve this error message. The change propagates the @Nullable type qualifier from the actual type qualifier to the expected type qualifier.

report a type qualifier incompatibility error. Similarly, the type qualifiers of method parameters and corresponding method arguments must be compatible. As another example, the type qualifiers of method return expressions and the corresponding method return types must be compatible. These programming language constructs that require the type qualifiers of two program elements be compatible are referred to as pseudo-assignments. CASCADE resolves the type qualifier incompatibilities by propagating the type qualifier of the right-hand side of the pseudo-assignment to that of the left-hand side.

If programmers run the checker through the Checker Framework Eclipse plug-in, they can apply the changes that CASCADE proposes through Quick Fix (Figure 6.14). Quick Fix is an Eclipse mechanism for providing automated changes that fix compilation problems.

All the changes that CASCADE proposes resolve the incompatibilities between the type qualifiers of the two sides of pseudo-assignments. At the implementation level, these changes fall into the following three categories.
**Variable Declaration Fixer.** A variable declaration fixer changes the declared type qualifier of a variable such as a method parameter, local variable, or field. Figure 6.10 illustrates a variable declaration fixer. The error message reports a type qualifier incompatibility and is caused by the piece of code that is highlighted in the editor. **CASCADE** looks up the AST node that corresponds to the highlighted piece of code. Since the highlighted expression is a method argument, **CASCADE** concludes that the change is a variable declaration fixer. Then, it looks up the method parameter that the method argument is bound to. Finally, it suggests to change the type qualifier of the method parameter to match that of the method argument.

**CASCADE** separates change representation from change application. It first creates a reusable representation of the change such that it can be applied on slight variants of the code. Then, it applies the change when the user decides to. Figure 6.15 illustrates the creation of a reusable representation of a variable declaration fixer, and Figure 6.16 shows the application of the created change on a given piece of code.

**Method Return Type Fixer.** If the type qualifiers of a return expression and the declared return type of the enclosing method are incompatible, the checker reports an error message. In this case, **CASCADE** proposes to change the declared type qualifier of the method to match that of the return expression.

**Method Receiver Parameter Fixer.** Java 8 permits type qualifiers on the receiver object `this` in the method signatures. If the checker reports an error message regarding the declared type qualifier of the special method parameter `this`, **CASCADE** proposes a method receive parameter fixer to change the declared type qualifier of `this`.

### 6.7.3 Proposing a Change Composition Plan

**CASCADE** proposes a plan for composing the changes in the form of a tree. It computes the plan through a speculative analysis.

Usually, changing a type qualifier comes with a **cascade effect**. A cascade effect refers to all the type qualifier changes required by a given type qualifier change. For instance, inserting the type qualifier `@Nullable` for parameter `xpic` of method `Node.loadTree` requires that the type qualifier of parameter `ic` of method `Node.oldSubindex` be changed to `@Nullable` as well (Figure 6.1). To support these cascade effects, **CASCADE** applies the change on a copy of the code and reruns the checker in the background to
input : RHS, the code snippet corresponding to the right-hand side of a pseudo-assignment with incompatible type qualifiers.  
actualTypeString, the string representation of the actual type of RHS.  
output: a reusable representation of the primitive change that fixes the problem.

function createVariableDeclarationFixerDescriptor(RHS, actualTypeString)

// Get the innermost AST node that covers RHS.

1 selectedNode ← getCoveredNode(RHS)

2 parentNode ← getParent(selectNode)  // Get the parent AST node.

3 if parentNode is an assignment expression or a variable declaration with initializer then

4 // Get the main identifier in the left-hand side expression.

5 variable ← getIdentifier(getLeftHandSide(parentNode))

6 variableCUPath ← getCompilationUnitPath(getDeclaringCompilationUnit(variable))

7 variableBindingKey ← getBindingKey(variable)

8 return new VariableDeclarationFixerDescriptor(variableCUPath,variableBindingKey, actualTypeString)

9 if parentNode is a method invocation then

10 methodCUPath ← getCompilationUnitPath(getDeclaringCompilationUnit(parentNode))

11 // Get the binding key of the method parameter corresponding to the selected method argument.

12 variableBindingKey ← getBindingKey(getMethodParameter(selectedNode))

13 return new VariableDeclarationFixerDescriptor(methodCUPath,variableBindingKey, actualTypeString)

Figure 6.15: The pseudocode for creating the abstract representation of the primitive change “variable declaration fixer”.

input : project, the Java project on which the change has to be applied.  
fixerDescriptor, the reusable representation of the change to apply.

1 function applyVariableDeclarationFixerDescriptor(project, fixerDescriptor)

2 cu ← getCompilationUnit(project, getCompilationUnitPath(fixerDescriptor))

3 variableDeclaration ← getVariableDeclaration(cu, getBindingKey(fixerDescriptor))

4 rewriteType(variableDeclaration, getNewTypeString(fixerDescriptor))

Figure 6.16: The pseudocode for applying the primitive change “variable declaration fixer”.

see if the change causes new type qualifier incompatibilities. If new incompatibilities occur, CASCADe will propose changes for fixing them and recursively continue to compute the consequences of those changes. Figure 6.17 illustrates the pseudocode of the speculative analysis.

CASCADe uses the binding information computed by the Eclipse Java Development Tools (JDT) to reliably apply changes on variants of the code. Eclipse JDT generates a binding key for declarations such as method and variable declarations. The binding key is a string that encodes the path to the
input : C, a piece of code
R, the root node of the tree to compute

output: a tree of changes and errors for inferring the type qualifiers of C rooted at R

1 function computeTree(C, R)
   // Let P be the set of problems that the type qualifier checker reports for C.
   2 P ← check(C)
   3 foreach p ∈ P do
      4 pn ← createTreeNode(p) // Create a new tree node for problem p.
      5 makeNodeChildOf(pn, R) // Make pn a child of R.
      6 F ← suggestedFixes(p) // Let F be the set of code changes that fix p.
      7 foreach f ∈ F do
         8 fn ← createTreeNode(f) // Create a new tree node for change f.
         9 makeNodeChildOf(fn, pn) // Make fn a child of pn.
      10 C′ ← changedCode(C, f) // Let C′ be a copy of C with code change f.
      11 computeTree(C′, fn)

Figure 6.17: The speculative analysis that computes the tree of changes is a recursive computation. The main call makes C be the original version of the code and R be a tree node that will be the only invisible node of the tree. The result will be a tree of changes and errors rooted at R.

declaration through the AST. By storing the change category, necessary binding keys, and type qualifier change, CASCADE can reliably apply the change on variants of the code for which the binding keys are valid. If the code changes drastically, the binding keys may no longer identify the desired declaration. In practice, because the changes that CASCADE makes to the code during its speculative analysis are only type qualifier changes, they preserve the binding keys.

CASCADE neither represents the changes as textual changes nor AST changes. If it represented the changes as textual changes, a change computed during the speculative analysis against a variant of the source code would have been unlikely to be applicable to the original source code. Similarly, if CASCADE stored references to the AST nodes that it modified in a copy of the code, the changes would have not been applicable to the original source code, because the AST node objects of the original code and its copy have different identities.

6.7.4 Presenting the Composition Plan

CASCADE displays the changes and error messages in a tree. The root nodes of the tree are the error messages that the checker reports for the original source code. The relationship between error messages and changes determines the parent-child relationship between the tree nodes. A change node is a child of
an error message node if and only if the change resolves the error message. Similarly, an error message node is a child of a change node if and only if applying the change introduces the error message.

CASCADE presents its composition plan using the standard Eclipse tree view to achieve a tight integration with Eclipse. The CASCADE tree is interactive. Programmers can expand the nodes of the tree to get suggestions for fixing error messages and see the consequences of applying the changes. They can also use the tree to locate the pieces of code that correspond to changes and error messages. The location of the piece of code corresponding to an error message is often different in the original code and a copy. Because CASCADE does not change the original source code, the tree is expected to be consistent with the original source code that programmers have access to. CASCADE uses a heuristic to locate the same piece of code in the original version of the code. It first locates the piece of code that caused the error message in the copy of the source code. Then, it expands that piece of code to include a larger part of the code. Finally, it searches the same piece of code in the original code and prefers the match whose offset is closest to the offset of the piece of code in the copy of the source code.

6.8 Evaluation

We conducted a user study to provide qualitative and quantitative insight about the strengths and weaknesses of two paradigms of inferring type qualifiers: batch and compositional. In the batch paradigm, the TQI tool takes a piece of code as input and inserts all the remaining type qualifiers into the code. In the compositional paradigm, the programmer manually composes multiple refactorings each of which inserts type qualifiers to a narrow piece of code.

In this study, we used two tools: JULIA and CASCADE, which support TQI in the batch and compositional paradigms, respectively. JULIA is the state-of-the-art static analysis TQI tool for nullness in the batch paradigm. CASCADE is the TQI tool that we developed based on the concepts of compositional refactoring and speculative analysis.

6.8.1 Research Questions

The goal of the study is to answer the following research questions for automated type qualifier inference (TQI) tools.
RQ1 How do JULIA and CASCADE compare along the following dimensions?

RQ1a task completion time
RQ1b quality of results
RQ1c learnability
RQ1d control
RQ1e willingness to use
RQ1f predictability

RQ2 How useful is the speculative analysis of CASCADE?

RQ3 What strategies do programmers employ in inferring the type qualifiers using JULIA and CASCADE?

6.8.2 Methodology

To evaluate CASCADE and answer the aforementioned research questions, we conducted a comparative lab study. We evaluated CASCADE, a compositional TQI tool, by conducting a controlled lab study on programmers instructing them to infer type qualifiers in both the compositional and batch paradigms. Such a study sheds light on the advantages and disadvantages of each paradigm.

6.8.2.1 Recruitment

We invited the graduate students and researchers at the computer science department of the University of Illinois at Urbana-Champaign who were familiar with Eclipse and Java to participate in our study. We invited graduate students because we have found them representative of professional programmers in our prior studies [125]. We advertised the study by sending an email invitation to all graduate students of the department and visiting several research labs within the department in person. We recruited 12 participants, and we offered a $25 gift card to each participant.

6.8.2.2 Lab Setup

We set up the lab to study one participant at a time. We reserved two adjacent rooms in the department. We set up a PC with the TQI tools under study in one room. The participants used the PC to perform the
tasks that we assigned to them. We set up another PC in the adjacent room and used it to observe the participant’s actions remotely. This live observation allowed us to tailor our interview questions based on the participants’ actions and help the participants quickly when they reached out to us with questions. We chose to move to a room different from the participant’s room to avoid causing distraction or stress to the participant. We instructed the participants to reach out to us if they have any questions.

6.8.2.3 Training

We prepared written and multimedia tutorials to introduce the participants to the nullness checker, JULIA, and CASCADE. We have made these artifacts publicly available [32].

Nullness Checker. Given that Java supported type qualifiers just recently (March 2014), few programmers knew about this feature during the time frame of our study. Thus, we prepared a tutorial based on the manual of the Checker Framework to familiarize the participants with the features of the nullness checker that were required for the tasks. The tutorial described some of the annotations supported by the nullness checker. Since it is easy to abuse the annotations and suppress warnings unnecessarily, the tutorial distinguished justified and unjustified annotations and encouraged the reader to use justified annotations. We asked the participants to study our tutorial about the nullness checker before the study. The tutorial had a few exercises to make sure that the participants grasp the key concepts of the nullness checker. The tutorial did not require the participants to try their solutions to the exercises against the nullness checker. Instead, it instructed the participants to email us their solutions. We reviewed their solutions and corrected them if necessary.

JULIA and CASCADE. We prepared video and written tutorials for both CASCADE and JULIA. During the lab study, we played the video tutorials to the participants before they used each tool and pointed them to the written versions of the tutorials for their reference. The CASCADE tutorial explains how to run CASCADE and use its various features. The tutorial for JULIA explains how to run JULIA using the Ant script that we integrated in Eclipse and what the outcome of JULIA looks like. The Ant script runs JULIA and inserts the annotations into the source code automatically.
6.8.2.4 Prequestionnaire

Before starting the task, we asked the participants to fill out a prequestionnaire to collect their demographic information. The prequestionnaire include questions about gender, age, start date of the graduate program, field of research, years of programming experience, level of familiarity with Java, Eclipse, and type qualifiers, programming languages familiar to the participant, and the programming projects that the participants contribute to.

6.8.2.5 Task Design

We designed the study as a within-subject, counterbalanced one. In a within-subject study, each participant evaluates both tools under study. An advantage of a within-subject design is that it enables the participants to qualitatively compare the two tools. Another advantage is that it mitigates the variance in the results due to the difference in the expertise levels of the participants. A standard disadvantage of the within-subject design is the carryover effect from the first task to the second one. Two common carryover effects are the learning and fatigue effects. Learning refers to the experience that the participant gains in type qualifiers and TQI during the first task. Fatigue refers to participants getting tired after finishing the first task. We employed two strategies to mitigate the carryover effect. First, we used two programs to avoid the learning effect of annotating the same program twice. Another benefit of using two programs is that it avoids the limitation of the results to a single program. Second, we used a counterbalanced design to balance the order in which the participants annotate the two programs using the two tools (JULIA and CASCADE). A participant can annotate the two programs and use the two TQI tools in four different orders. To achieve a counterbalanced design, we randomly divided our participants into four groups of the same size. Then, we had each group use the two TQI tools on the two programs in a unique order.

We used two programs, Barnes-Hut (BH) and Minimum Spanning Tree (MST), from the JOlden benchmark suite for our user study. We had two criteria for the programs under study. First, it should be possible to insert the nullness annotations into the selected piece of code in 30–40 minutes. Second, the program should be representative of real code.

During our pilot studies, we found that that the participants could not annotate the selected programs within the allotted time. So, we removed parts of the code and simplified other parts of it to make it
possible to annotate it within the allotted time. We have made our simplified versions of the BH and MST programs, which we used during the study, publicly available as part of the artifacts of the study [32].

We asked the participants to insert the nullness annotations in each program (BH and MST) using the designated TQI tool (JULIA and CASCADE). We told the participants that they are allowed to refactor the code but not change the behavior of the program. We required the participants to make sure that their uses of the annotations are well justified. For example, we asked them to avoid using @SuppressWarnings annotations and assert statements where they believe they are inappropriate or not needed. We also asked the participants to avoid @Nullable annotations where @NonNull is appropriate and vice versa. Similarly, we asked them to avoid @NonNull annotations where no annotation is needed. We provided a test suite along the program and required that the test suite passes before and after the task. After the participant finished the task, we asked them to run the tests and verify their annotations.

6.8.2.6 Postquestionnaire

After the participants finished the task, we asked them to fill out a postquestionnaire that captured their relative preferences towards the two tools along various dimensions. The postquestionnaire asked the participants to rank the two tools along multiple dimensions including ease of use, transparency, control, and willingness to use. In addition, it asked the participants to elaborate on the strengths and weaknesses of each tool.

6.8.3 Interview

After filling out the postquestionnaires, we conducted brief semi-structured interviews with the participants. The goal of the interview was to solicit more insight about how the participants perceived each tool and what strategies they employed in using each tool. During the interview session, we asked questions including the ones listed below.

- How did each of JULIA and CASCADE affect your strategies for adding type qualifiers?
- How intuitive and useful was the tree of changes and errors presented by CASCADE? How useful did you find CASCADE’s suggestions of future errors and changes?
- What are your suggestions for improving JULIA and CASCADE?
6.8.4 Results

6.8.4.1 Participant Demographics

The participants came from nine different research labs at the computer science department of the University of Illinois at Urbana-Champaign. One participant was a post-doc, one was a master's student, and the rest were PhD students. The participants worked in a variety of areas such as high performance computing, natural language processing, security, mobile computing, compilers, and computer architecture. The prequestionnaire asked the participants to rate their familiarities with Java and Eclipse along a 5-point Likert scale. All participants considered themselves at least familiar with Java, and 11 considered themselves at least familiar with Eclipse.

6.8.4.2 Task Completion Time (RQ1a)

To compare the efficiency of programmers with each TQI tool, we measured the task completion times with each tool. With a Welch’s t test ($t(11) = 2.89$, $p = 0.01$, Cohen’s $d = 1.13$), we found that the participants were significantly faster using CASCADE (mean = 28 minutes) than JULIA (mean = 39 minutes).

6.8.4.3 Quality of Results (RQ1b)

We computed the distribution of the annotations that the participants inserted using each of CASCADE and JULIA as a proxy for the overall quality of the annotations. Table 6.3 shows the total number of each annotation for each program and TQI tool. Overall, the participants inserted fewer @Nullable annotations using CASCADE than JULIA. This indicates that the participants were able to avoid unnecessary @Nullable annotations by placing a combination of @SuppressWarnings and @NonNull annotations and assert statements at appropriate places.

The @SuppressWarnings annotation is used to suppress a false positive reported by the checker. The participants inserted fewer @SuppressWarnings annotations with CASCADE than JULIA. However, the @SuppressWarnings annotations that the participants inserted with CASCADE suppressed more statements. The reason was that one participant chose to annotate a whole method as @SuppressWarnings while all other @SuppressWarnings annotations suppressed the checker for a single variable declaration.
Table 6.3: The distribution of the different annotations that the participants added to the subject programs (BH and MST). Rows @SuppressWarnings, assert, @Nullable, @NonNull report the total number of each annotation for each program and TQI tool. “Suppressed Statements” is the number of statements suppressed by the @SuppressWarnings annotations, which we calculated by counting the number of semicolons in the suppressed piece of code. “Unresolved Errors” is the number of problems that the checker reported after the participants finished annotating the programs.

<table>
<thead>
<tr>
<th></th>
<th>BH</th>
<th></th>
<th></th>
<th>MST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JULIA</td>
<td>CASCADE</td>
<td>JULIA</td>
<td>CASCADE</td>
</tr>
<tr>
<td>@Nullable</td>
<td>71</td>
<td>61</td>
<td>80</td>
<td>55</td>
</tr>
<tr>
<td>@NonNull</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>@SuppressWarnings</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Suppressed Statements</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>assert</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Unresolved Errors</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.4: Number of participants of the lab study who preferred each TQI tool (the first two columns) with respect to each quality (rows). The last column lists the number of participants with no preference.

<table>
<thead>
<tr>
<th>T = CASCADE or JULIA</th>
<th>CASCADE</th>
<th>JULIA</th>
<th>no preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>I found T easy to learn.</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>I know why T inserted each annotation.</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Using T, I have control over the annotation process.</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>I am willing to use T in future.</td>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Programmers can write assert statements to make the checker aware of certain properties. Use of assert is justified for those properties that hold at runtime but the checker cannot verify statically. The participants inserted slightly fewer assert statements with CASCADE than JULIA.

One participant ran out of time while annotating MST with JULIA and left two errors of the checker unresolved.

6.8.4.4 Ease of Learning (RQ1c)

Despite the fact that JULIA is a single push-button tool and CASCADE offers several interactive features, the participants rated the two tools equally easy to learn (Table 6.4). For example, P_{12} said:

I just watched the tutorial once and found them easy to understand and learn.
6.8.4.5 Control (RQ1d)

The results of the postquestionnaire (Table 6.4) indicate that the participants felt more in control with CASCADE than JULIA. P5 mentioned the following on the postquestionnaire:

Even though I could normally go back and change things in JULIA, I had no control over the process (since it was a batch script)

With CASCADE I could choose whether I wanted to make each of the changes it suggested.

6.8.4.6 Willingness to Use (RQ1e)

According to the postquestionnaire results (Table 6.4), the participants are more willing to use CASCADE than JULIA, assuming that robust and efficient implementations of both tools are available.

P9 mentioned that she would be willing to use JULIA for legacy software.

I would use JULIA if I wanted to annotate a project that is not starting now, so there is a lot of code that needs to be annotated immediately. The feature of automatically inserting annotations would be very useful in this case, provided that the amount of annotations I do not understand is not excessive.

6.8.4.7 Predictability (RQ1f)

A predictable program transformation tool is one that makes it easy for the programmers to tell what code changes it made and why. Table 6.4 indicates that the participants knew better with CASCADE than JULIA why the tool inserted each annotation. Participants reported that CASCADE made it easier to find the reason \(N = 7\) and location \(N = 1\) of inserted type qualifiers. On the other hand, participants found it difficult to find the reason \(N = 6\) and location \(N = 1\) of the type qualifiers that JULIA inserts. Participants said that JULIA adds many annotations \(N = 2\) including unnecessary annotations \(N = 2\) and does not explain why it inserts each annotation \(N = 3\).

For example, P9 said:

For CASCADE, since there were no annotations in the program, I had to start adding annotations, which was good for me because I could see why I needed each annotation and I had control over this procedure. With JULIA, a lot of annotations were already added, so, I had to change them. For some reason, this was harder for me because changing something that is already there is harder. You need to consider why it was placed at this point without having the whole program in your head.
As another example, P_{12} said:

Since I'm not very familiar with the algorithm JULIA uses, it feels a bit like a black box to me.

6.8.4.8 Speculative Analysis (RQ2)

During the interviews, we asked the participants about the usefulness of the tree that CASCADE computes through a speculative analysis. Eight participants indicated that the speculative analysis of CASCADE is useful. For instance, P_8 said:

I really like that. I could see how the warnings propagated through things more easily as opposed to just running the checker, which is kinda like oh this doesn’t work, well, what if I do this, oh there is a new one.

The participants said that CASCADE (i) makes them more careful (N = 3), (ii) helps them understand and think about the code structure (N = 2), and (iii) helps them focus on one problem at a time (N = 1) by showing the consequences of applying each change.

For example, P_4 said:

It [The CASCADE tree] was very good. [...] To some extent, I got a feel about how deep the effects are, where actually the source of the error is, where you could follow multiple paths, either you could fix the source or fix the whole tree. [...] It made you think more about is the solution it gave you the right solution or you could change something at the source of the tree so that it would give you a different possible fix.

As another example, P_3 mentioned:

I looked at the suggestions by CASCADE but also it made me actually think about the code structure.

6.8.4.9 Strategies (RQ3)

We investigated how the participants used each TQI tool to identify the common strategies that they employed. By encouraging the good strategies and discouraging the bad ones, TQI tools can become more effective.
**Exploration Order.** CASCADE does not impose any restrictions on the order in which the programmers expand the tree and apply changes. Four participants first applied the changes on the shorter paths of the tree. This observation can guide the automatic ordering of the nodes of the tree.

Two participants did not expand any changes. They only expanded errors, applied the suggested changes, and recompute the tree. Effectively, these participants explored the tree in a breadth-first order. Since these participants did not use the speculative feature of CASCADE much, they had to frequently recompute the tree, which was slow. As a result, these two participants were less satisfied with CASCADE.

**Change Application Order.** Although CASCADE computes the changes from the root to the leaves of the tree, it does not impose an order for applying the changes. Two participants applied the changes on a path from the leaf to the root of the tree, and one participant applied the changes of a path out of order. The rest applied the changes of a path from the root to the leaf.

**Long Paths.** Long paths in the CASCADE tree indicate deep consequences of a type qualifier change. A good strategy that the participants employed when they encountered long paths was to examine the path, read the code, and refactor the code, if possible, to cut the path short. On the other hand, some participants ran into problems in handling long paths. Two participants applied the changes on the paths as they expanded the paths. Rather than first examining the whole path, they applied the changes prematurely. When these participants reached the leaves of the tree and noticed the unresolved errors, this strategy made the participants backtrack some of the changes they had applied. Similarly, one participant applied the last few changes on a path without examining them. Programmers should be more careful in handling the paths that leave unresolved errors. One way to improve CASCADE is to make it discourage careless change applications by warning the programmers about those paths of the tree that leave unresolved errors and require closer attention.

**6.8.4.10 Performance**

Participants said that the tree computation of CASCADE was slow ($N = 5$) and JULIA was faster ($N = 3$).

For instance, $P_7$ said:

*With CASCADE, one thing that annoyed me was that it's kinda slow, because I think it is meant to be interactive. [...] With JULIA, yes, I'm going to do it in the old “write code, compile, look at error”*
cycle. So, it was familiar. I didn’t expect any better of it. So, it didn’t matter that it was on the slower side. If CASCADE was faster, I’d probably be happier using it.

Four said that the manual work required to use CASCADE may make it unsuitable for large projects. On the other hand, they complained that the overhead of understanding and fixing the annotations inserted by JULIA is high ($N = 7$).

Although CASCADE’s tree computation was slow, the participants finished the tasks more quickly with it than JULIA (Section 6.8.4.2).

6.8.4.11 Suggestions for Improvements

Two participants suggested that CASCADE lets the programmer apply all the changes inside a subtree at once. For instance, $P_5$ said:

[Had the code been more complicated,] I would have liked the strategy to have been right-click, fix this tree for me rather than me visiting the entire tree. And, if there is some point in the leaf this sort of error, then tell me and I’ll deal with it manually. Basically, kinda what JULIA is doing but localized on the tree basis. So, I would’ve said run Julia on that tree kinda thing. [...] This is why I feel like a combination of the two would have been better.

Two participants expected that applying all the changes that CASCADE suggests would resolve all the problems. For instance $P_6$ said:

With CASCADE, when you apply a change, new errors will come and it was deceiving that you would apply a change and you’ll get new errors. [...] After I applied all changes on the tree, I still had errors.

Similarly, $P_7$ said:

With CASCADE, I got a little cocky because it seemed CASCADE would be very clever. And, so, I just kept accepting its annotations and suggestions and ended up annotating myself into a corner, because I ended up in a case where you had a $\texttt{Nullable}$ annotation and it was being dereferenced and wasn’t obvious how to fix it. So, I had to go back and again look at the code, understand it myself, and fix things. I think CASCADE was a false sense of security because of the way it was working.

CASCADE’s change composition plan is meant to be suggestive as opposed to definitive. While applying all changes on some paths of the tree leaves no problems behind, some other paths eventually leave
some problems. If the leaf node of a path is an error message node, applying all the changes on that path will result in the error message of the leaf node. One way to discourage programmers from taking CASCADE's suggestions definitive is to warn the programmers about the paths in the tree that leave some problems behind.

One participant found the overlaps between the subtrees of CASCADE confusing. Two participants suggested that CASCADE organizes the changes according to data structures instead of the call hierarchy.

Participants suggested that JULIA be made more interactive ($N = 3$) and explain what ($N = 1$) and why ($N = 3$) it changes.

One suggested that JULIA avoids making changes that cause the checker to report errors.

With JULIA, I'll say that the program should run automatically but not implement all the changes. For example, we ran JULIA and there was error given by the checker. Let's say it does not apply changes to that set of tree where after compilation it gives you an error. You just ask the programmer I cannot figure out for this tree. A mix of CASCADE and JULIA, that's what I'm talking about.

TQI tools are bound to be inaccurate (Section 6.2). The results of the study suggest that the TQI tools should avoid propagating inaccurate type qualifiers across the code base. Such a propagation often causes additional overhead as the programmers have to understand and revise the propagated type qualifiers. One possible way to improve the existing batch TQI tools is to report possible sources of inaccuracies and checker errors to the programmers and let the programmers decide how to handle such difficult cases. This will essentially bring some of the strengths of CASCADE into batch TQI tools and make them more iterative.

On the other hand, CASCADE requires the programmers to confirm the insertion of each type qualifier. While these confirmations put the programmers in control, they can also make the inference process tedious. One way to make the inference process of CASCADE more efficient is to incorporate some of the analysis that batch TQI tools perform in CASCADE. Such an analysis can be used to make CASCADE automatically insert some of the type qualifiers that are accurate and do not cause the checker to introduce new errors.
6.9 Limitations

Given the recent addition of type qualifiers to Java 8, few Java programmers were familiar with this new feature when we conducted the study. Although the participants were familiar with Java and Eclipse, they were new to type qualifiers. Thus, we trained the participants about the type qualifiers. Nonetheless, the level of familiarity with type qualifiers may affect the results of the study. As more programmers learn about type qualifiers, future studies can experiment with programmers that are more experienced with type qualifiers.

The time limits on lab studies constrain the choice of subject programs. Factors that may affect the performance of a TQI tool include the size of the subject programs, preexisting type qualifiers in the programs, and dependence of the subject programs on libraries. While one can speculate about the affect of these factors on the performance of a batch and compositional TQI tool, more research is required to evaluate such speculations.

Because of the limited duration of the lab study, the study compared only two tools. Future research can study other configurations such as inserting type qualifiers without a special TQI tool and just relying on the type qualifier checker or using a combination of a batch and compositional TQI tool.

During a lab study, the participants do not commit to long-term maintenance of the programs. The desired maintainability of the type qualifiers is another factor that may affect the desirability of a TQI tool. To avoid too much variability in the subject programs and mitigate unexpected bugs of the TQI tools in unknown code, we asked the participants to annotate two programs that we selected from a suite of benchmark programs.
Chapter 7

Related Work

Early research on refactoring go back to 1990. Opdyke and Johnson presented a catalog of refactorings [127, 128]. Griswold presented a model for behavior-preserving program restructuring and built a prototype tool for Scheme [82]. Roberts et al. developed a refactoring tool for Smalltalk [140]. The user interfaces of refactoring tools have not changed much since then. Today’s refactoring tools still follow a monolithic, wizard-based paradigm similar to the first refactoring tool for Smalltalk. In this paradigm, the programmer selects a piece of code, invokes an automated refactoring, sets the configuration options on the refactoring wizard, previews the change, and confirms the change. If the preconditions of the tool are met, the tool applies the confirmed changes.

Although refactoring is a widespread practice [112, 169], automated refactorings are greatly underused [121, 125]. We conducted a field study followed by interviews with programmers (Chapter 3). This study showed that programmers invoke the refactoring tool mostly for applying simple changes such as Rename and Extract Local Variable. In addition, this study revealed some of the major factors that lead to the underuse of modern refactoring tools, e.g., the rare opportunity to apply complex refactorings, programmers’ unawareness of automated refactorings, too technical and inconsistent naming of automated refactorings, unpredictability of the impact of an automated refactoring on the code, and the cost of configuring automated refactorings.

Realizing the shortcomings of the wizard-based paradigm of refactoring, researchers proposed alternative means of invoking the refactoring tool [103, 114], presenting refactoring error messages [116], and autocompleting refactorings [74, 77]. Nevertheless, this thread of work does not address the inherent problem of the refactoring wizards, namely unpredictability. On the other hand, the human-automation interaction community has extensively studied the design elements that lead to use and disuse of automation. However, this thread of work has focused on other fields such as aviation and health-care.
Our work bridges the gap between these two threads of work in the software engineering and human-automation interaction communities. We propose a novel paradigm that provides a more appropriate level of automation for program transformations and fits naturally into programmers’ coding practices.

7.1 Empirical Studies of Refactoring

Murphy et al.’s study of 41 developers using the Java tools in Eclipse [112] stimulated empirical studies on refactoring. Their study was the first to report the usage frequencies of automated refactorings. They reported the usage frequencies of Eclipse perspectives, views, and commands (refactorings being a subset). Their data provided a holistic view of how often various features of Eclipse were used and raised questions about how users were using the features of the IDEs. Our interaction data collection methods are similar because we both collect data from programmers in their natural settings. However, the focus of our work was on refactorings and we supplemented our quantitative data by qualitative ones. Our study extends their work by collecting more detailed data to explain the refactoring patterns. For example, we collect the selections used to invoke the automated refactorings in order to find the potential problems in invoking automated refactorings.

Murphy-Hill et al. then analyzed the existing data from Murphy et al., the Eclipse foundation [23] and two other data sources to study how developers refactor [121]. Their work was the first to suggest that programmers underuse automated refactorings. They also surveyed five programmers to get some insight about why programmers may not use automated refactorings. Our work builds on theirs and collects richer quantitative and qualitative data to better understand why programmers underuse automated refactorings. For example, we collect data about the failures of automated refactorings to study how they affect the user experience.

In another study, Murphy-Hill et al. examined barriers to using the Extract Method refactoring [115]. They instructed several participants to apply the Extract Method refactoring on several open source projects. They reported that programmers had trouble selecting the portion of code to extract and understanding the error messages of the Extract Method refactoring in Eclipse. Based on this observation, they proposed improvements to the user interface of automated refactorings. Specifically, they developed a prototype tool that visualizes code selections and error messages. We also observed that selection problems were common: at least ten of our participants encountered such problems. While some of
the results of the two studies overlap, the focus of our work was identifying the factors that deterred programmers from using automated refactorings rather than resolving a specific usability issue, e.g., poor messages of the Extract Method refactoring. Our study extends theirs by detecting problems in invoking refactorings and comprehending error messages in real-world environments for many more refactorings.

Mealy et al. listed 38 guidelines for refactoring tools by analyzing the literatures of industrial usability standards and human factors [110]. Our approaches are complementary because we proposed improvements to the design of refactoring tools based on the actual usability problems that our participants encountered.

Parasuraman and Riley discussed humans’ use, misuse, disuse, and abuse of automation [131]. We adapted their framework in three ways to the automation of refactorings. First, they defined abuse as enforcing automation by designers and managers without considering the consequences on the users. Since we did not find an evidence of abuse in the context of refactorings, we excluded it from our framework. Second, we used a slightly different definition of misuse. They defined misuse as overreliance on automation. We considered uses of automated refactorings in ways not recommended by designers as misuse. Finally, we have identified the factors that pertain to use, misuse, and disuse of automated refactorings specifically and not automation in general.

7.2 Finding the Usability Problems of Refactoring Tools

Researchers have proposed alternative designs to improve the usability of refactoring tools (Chapter 5) [74, 77, 103, 114, 116, 119]. These proposals seek to address specific usability problems such as programmers’ unawareness of automated refactorings [119], poor methods of invoking automated refactorings [74, 77, 103, 114], bad error messages of an automated refactoring [116], and low predictability of automated refactorings (Chapter 5). While this line of work has generated promising ideas for improving the usability of refactoring tools, they lack a general mechanism for finding usability problems encountered by programmers in the field.

Studies (Chapter 3) [103, 121] have uncovered some of the usability problems of refactoring tools through conventional usability evaluation methodologies such as lab studies, interviews, and surveys. Our adaptation of the critical incident technique to refactoring tools differs from these studies in two
ways. First, we derive a systematic usability evaluation method for refactoring tools. Second, our evaluation method relies mostly on automatically collected usage data rather than qualitative data.

Akers et al. devised a variant of CIT for evaluating the usability of applications like SketchUp and Adobe Photoshop [35]. They instrumented SketchUp [29], a 3-D modeling application, and recorded invocations of undo and erase events along with screen capture video [35]. They conducted a lab study, in which they instructed the participants to perform predefined tasks. To obtain more contextual information about these events, the participants were paired up to discuss their captured video episodes centered around the recorded events. They found that the participants sometimes failed to report problems because they forgot or blamed themselves rather than the application. They were able to identify several usability problems by analyzing the undo and erase events. Their work exemplifies the potential of monitoring for identifying usability problems. Inspired by their work, we collected detailed information about the usage of refactorings such as undoing, redoing, cancellation, reported messages, and repeated invocations to identify potential usability problems. An interesting difference in the results of our studies is that we did not find undo a good indicator of the usability problems of refactoring tools (Section 4.4).

Yoon et al. studied programmers' backtracking strategies (e.g., removing inserted code or restoring removed code) [170]. This study can be viewed as an application of CIT with implications for better tool support for backtracking strategies [171].

Studies have confirmed the effectiveness of remote usability evaluation in different settings [37,51,94]. Nevertheless, practitioners do not practice it much, although when they do, they appreciate its value [53].

While similar to this line of research we propose a variant of CIT, there are several major differences. First, we focus on IPT tools, for which the notion of critical incident is not well-understood. Second, we avoid interference with programmers' workflow. Thus, we capture more detailed information automatically to reduce the involvement of the users in identifying the usability problems. Finally, we evaluate our method by analyzing the authentic data collected from programmers performing real tasks in their normal working environments.
7.3 Compositional Refactorings

One way of automating composite refactorings is to build tools that execute a sequence of smaller refactorings atomically [104, 160]. Motivated by the perceived benefits of composite refactorings, researchers have focused on creating monolithic tools, e.g., wizard-based tools, that automate big transformations. To build trust in such monolithic tools, researchers rely on formalisms to guarantee behavior preservation. Several researchers [57, 100, 141] have proposed methods for checking the behavior-preservation of a composite refactoring based on the pre and post conditions of its individual refactorings. Roberts’s PhD thesis [141] provided a framework for behavior preservation by specifying each refactoring in terms of preconditions and postconditions. Preconditions specify the properties that must hold before the refactoring and postconditions specify the properties that must hold after applying the refactoring. As long as multiple refactorings do not violate each other’s pre and post conditions, they could be composed together. Extending this idea, Cinneide and Nixon [57] designed and implemented several composite refactorings for introducing design patterns in a behavior-preserving manner. Others [104, 160] have introduced scripting languages for automating composite refactorings. These researchers have concentrated on building tools for toolsmiths who are interested in building new monolithic tools from compositions of smaller refactorings in a behavior-preserving manner. Kniesel and Koch [100] created ConTraCt, a refactoring editor, to help toolsmiths build new refactoring tools from smaller refactorings. Verbaere et al. created JunGL [160], a domain-specific language for scripting new refactorings in .NET. Similarly, Li and Thompson [104] proposed a scripting languages for Erlang refactorings. We introduce a radically different paradigm for automating composite refactorings. Rather than building a monolithic tool from several refactorings, we propose that a large refactoring be decomposed into smaller ones. These two paradigms suit different needs. The monolithic paradigm is suited for toolsmiths who are in charge of applying a refactoring on a large code base in batch mode. The compositional paradigm is designed for interactive refactoring in an IDE. In addition, the monolithic paradigm aims to provide correctness guarantees by inferring preconditions [57, 100, 141]. The compositional paradigm makes it easy for the programmers to verify the correctness of the refactoring by making each step easy to predict and verify.

Compositional refactorings closely mimic the steps that a programmer takes while performing large refactorings (Figure 5.6). Each refactoring is small and its changes are predictable and understandable.
The changes are presented directly to the user in the code editor. Our field study (Chapter 3), our analysis of refactoring composition patterns, and our survey and lab studies all suggest that overall programmers prefer our proposed paradigm, because the outcomes are easier to predict. In our proposed paradigm, programmers are also free to intersperse manual edits in between refactorings, giving them greater flexibility in composing refactorings. Note that our compositional refactorings are not meant to replace, but rather to complement existing monolithic tools. As reported in Sections 5.4 and 5.5, some developers prefer the current monolithic paradigm over our proposed paradigm in some cases.

Murphy-Hill et al. [121] showed that developers frequently invoke the same kinds of automated refactorings in batches, i.e., within 60 seconds of one another. Negara et al. [125] analyzed the proportions of manual and automated applications of ten frequently used refactorings relative to all code edits and found that more than one third of these refactorings are performed in clusters. Our study goes beyond reporting the frequencies of refactoring sets (Section 5.1) and sheds light on the rationales of composing automated refactorings (Section 5.2).

Schäfer et al. [144] argued that a very fine-grained decomposition of a refactoring into a composition of micro-refactorings over an extended language makes the implementation of the refactoring tool more reliable and easier to understand. They used Extract Method as an example to demonstrate their technique. Their goal was to make the implementation of the refactorings more reliable. Our goal was to show that the compositional paradigm is an effective interaction model for automating large refactorings. While their focus was on reliability, ours is on usability. Generalizing their results, the compositional paradigm should lead to more reliable implementations of large refactorings.

Lee et al. [103] hypothesized that the invocation mechanism for each refactoring was too complicated and implemented a tool using drag-and-drop that streamlined the invocation process. They showed that invoking refactorings through drag-and-drop gestures is more intuitive than menus and wizards. Parnin et al. [133] proposed the use of multi-touch gestures for performing refactorings. Alternative methods of invocations are complementary to our work, because they can streamline the invocation of the individual steps of a composite refactoring.

Refactoring auto-completion systems [74, 77] prompt programmers to automatically complete a manual refactoring. Although both compositional refactoring and refactoring auto-completion systems aim to make automated refactorings more usable, they focus on different usability problems. Com-
positional refactoring and refactoring auto-completion systems focus on different usability problems. Auto-completion systems aim to make the invocation of refactoring tools seamless, while compositional refactoring addresses the unpredictability of tools for complex refactorings. Nonetheless, a combination of these two lines of research might lead to innovative solutions for auto-completing complex refactorings.

Researchers have proposed refactoring auto-completion systems [74,77], which prompt programmers to automatically complete a manual refactoring. Others have proposed alternative methods of invoking refactorings drag-and-drop [103] and multi-touch [133] gestures. While these systems aim to make the invocation of refactoring tools seamless, compositional refactoring makes automated refactoring more predictable. Nonetheless, alternative methods of invocations are complementary to our work, because they can streamline the invocation of the individual steps of a composite refactoring.

7.4 Type Qualifier Inference

Researchers have developed many TQI tools (Table 6.1). We discussed the differences between CASCADE and the existing batch TQI tools through our comparative lab study (Section 6.8). In the following, we discuss two tools that, similar to CASCADE, rely on an existing checker for inferring annotations. Houdini [69] is a tool for inferring the annotations required by ESC/Java [70], a static program checker, and CANAPA [56] is a tool for inferring nullness annotations for ESC/Java2, a static checker for the Java Modeling Language [52]. Houdini, CANAPA, and CASCADE are similar in that they use an existing checker for inferring annotations. This technique makes these three tools general. Unlike CASCADE, Houdini and CANAPA are not compositional, because they insert all the annotations that they infer at once, a change that tends to be large and unpredictable. The annotations inferred by Houdini and CANAPA may cause the checkers to report errors. Houdini generates an HTML report to help the programmer in finding the cause of each error. CASCADE assists the programmer to find the root causes of errors by computing an interactive tree of related changes and errors. Houdini and CANAPA are not speculative either, because they do not present the consequences of inserting the annotations in advance.
7.5 Speculative Analysis

Programming involves making many decisions, the consequences of which are sometimes unknown. Speculative analysis assists programmers in making decisions by precomputing the consequences of the decisions. It usually achieves this by eagerly committing to a decision and reporting its consequences to the programmer while keeping the programmer's view of the system intact.

Although speculative analysis, also known as speculative execution, is an old optimization technique used in domains such as computer architecture and database systems, it has been applied to software engineering only recently. The goal of the existing applications of speculative analysis to software engineering tasks is to improve the productivity of programmers not the performance of the system.

Quick Fix Scout [123] employs speculative analysis to make better suggestions for resolving compilation problems. Modern IDEs such as Eclipse suggest automated changes to fix compilation problems whenever possible. Eclipse refers to these automated changes as Quick Fixes. Quick Fix Scout applies each Quick Fix on a copy of the code and observes the effects of the Quick Fix on the compilation problems. Then, it computes a mapping between the Quick Fixes and the compilation problems that they resolve. Finally, it uses this mapping to augment the Eclipse Quick Fix in two ways. First, it presents the number of compilation problems that each Quick Fix resolves. Second, it presents each Quick Fix at all locations where it resolves a compilation problem.

Solstice [122] is a general framework that uses speculative analysis to turn an offline analysis into a continuous analysis. For instance, Solstice has been used to report up-to-date results of FindBugs [163] to the programmer without requiring the programmer to run FindBugs every time the code changes.

Crystal [50] informs the programmers ahead of time about the conflicts, build errors, and test failures that upcoming Version Control System (VCS) operations will introduce. It computes the consequences of merging the changes of the programmers into the VCS by speculatively merging the changes on copies of the code base.

CASCADE differs from existing applications of speculative analysis in two major ways. First, it brings speculative analysis to a new domain, namely, TQI. Second, it uses deep as opposed to shallow speculative analysis. A shallow speculative analysis reasons about the state of the system only one step away. However, a deep speculative analysis reasons about the state of the system several steps away. In addition to proposing an automated change to fix a type qualifier checker error, CASCADE proposes automated
changes to fix the new checker errors that the previously proposed change may introduce. CASCADE allows the programmer to navigate the future states of the code through an interactive tree visualization.
Chapter 8

Future Work

8.1 Appropriate Levels of Automation for Software Engineering Tasks

Researchers and practitioners invest in automating many software engineering tasks. Some of these efforts have led to technologies that programmers have adopted, e.g., IDEs, Version Control Systems, Continuous Integration Systems. However, some other automation efforts, such as automated refactorings, have not been widely adopted. This dissertation argues that a major cause of the low adoption of refactoring tools is their over-automation. Over-automation leads to inappropriate feedback and interaction, which discourages programmers from using the automation. Do the automation technologies for other software engineering tasks suffer from over-automation? Asking this question is an important first step in finding an appropriate level of automation and interaction model.

Consider software remodularization [166], which is an architectural program transformation that infers module boundaries from an almost monolithic, legacy piece of code. The tools that researchers have developed for remodularizing software employ clustering [39, 108, 154], search-based [111, 136], or information retrieval [107] techniques to automatically find a set of modules that optimize some metrics. These metrics are usually inspired by the desired properties of a modular system such as high cohesion and low coupling [34, 44]. However, studies [38, 168] suggest that the metrics do not produce modules close enough to the programmers’ mental models of the software. Is over-automation the root cause of the ineffectiveness of existing remodularization techniques? Perhaps we need to design interaction models that incorporate programmers’ intuitions [43].

Automated debugging is another automation of a software engineering task, which suffers from low adoption. Researchers have proposed many forms of automation to support the activities involved in debugging [42, 83, 93, 138, 172–174], i.e., fault localization, fault understanding, and fault repair. For example, fault localization techniques identify the statements that may be the root causes of a
failure. A recent study [134] suggests that automated debugging tools suffer from insufficient contextual information, lack of traceability between input and output, poor ranking of the results, and high configuration cost. Is the root cause of the problems of automated debugging tools over-automation? Will debugging tools become more effective if they reduce their levels of automation, engage the programmers in the debugging process, be more transparent, and show justifications for their results?

8.2 Finding the Usability Problems of Refactoring Tools

Our adaptation of the Critical Incident Technique (CIT) opens up future research in several directions. One direction is to adapt other variants of CIT, e.g., UCIT, to IPT tools. This would provide insight about the quantity and quality of user reports and their effect on the number and severity of inferred usability problems.

Another future line of research is to extend our method to other IPT tools, e.g., code generators, bug fixers, and other refactoring tools.

Finally, our vision is that programming environments adopt data collection frameworks like CODINGSPECTATOR to make remote asynchronous usability evaluation possible at a large scale. This large scale will raise new research challenges such as privacy assurance and automatic clustering of similar critical incidents.

8.3 Compositional Refactoring

We evaluated the compositional paradigm using survey and lab studies. Future research can evaluate this paradigm in the field for more refactorings and program transformations in general.

One area of future work would be to investigate other ways of assisting programmers in composing refactorings. For example, how accurately can a tool predict the next refactoring that a programmer may invoke in a compositional paradigm? Can a history of previously invoked refactorings and frequent refactoring sets be used to accurately make such a prediction?

Another line of future work would be to study the pedagogical aspects of the compositional paradigm. While our discussion of refactoring composition patterns (Section 5.2) could serve as a starting point for learning these patterns, more research is needed to deliver a more comprehensive catalog. One
technique to make people adopt new skills is to make it easy to learn from their peers. How can we facilitate the transfer of refactoring composition skills in a team?

8.4 Type Qualifier Inference

Our participants pointed out that CASCADe’s tree computation is slow. Given that CASCADe is an interactive TQI tool, it is important for it to be fast so that programmers can invoke it frequently. Our goal was to assess the user interaction model of CASCADe, and we left optimizing its performance to future work.

There are many ways to optimize the performance of CASCADe. First, disjoint subtrees of CASCADe can be computed in parallel. Second, the tree can be computed lazily as the programmer expands the tree. Finally, it is possible to use the copy-on-write strategy to avoid taking many copies of the whole code for speculative analysis.

Currently, the speculative analysis of CASCADe is unidirectional. That is, it propagates type qualifiers in one direction: right to left. For instance, if the checker reports a type mismatch in an assignment, CASCADe proposes a change to propagate the type qualifier from the right-hand side of the assignment to its left-hand side. However, the programmer may sometimes have to propagate type qualifiers from right to left, e.g., when the code is partially annotated or it depends on a library. Currently, CASCADe requires the programmers to do the left-to-right propagation manually and then refresh the CASCADe to see the consequences of the propagation. Future research can extend CASCADe’s speculative analysis to be bidirectional. Bidirectional speculative analysis shows the programmer a larger part of the solution space. The challenge is to present this larger solution space to the programmers concisely and intuitively.
Chapter 9

Conclusions

Interactive program transformation (IPT) tools, such as refactoring tools, aim to make the evolution of software more economical and reliable. Despite the automation of many recurring or sophisticated changes, refactoring tools are heavily underused (Section 3.3) [97, 121, 125]. We argue that usability problems are major obstacles to a widespread use of refactoring tools (Chapter 3).

Our quantitative data and interviews revealed many factors that affect the appropriate and inappropriate uses of automated refactorings. The results show that some automated refactorings are underused, because programmers are unaware of them, the overhead of learning and configuring some automated refactorings does not justify the few opportunities to use them, the names of some automated refactorings are confusing, and programmers cannot predict the outcomes of the tools that make large changes. On the other hand, programmers appreciate the tools that propose only the refactorings applicable to the current context, and are willing to use automated refactorings even when they may change the program's behavior. Our study shows that the major barrier to the adoption of refactoring tools is their usability problems not their rare bugs. These results suggest that designers should aim for flexible, predictable, and truly interactive automated refactorings in the design of the next generations of refactoring tools.

To improve automated refactorings, it is important to learn how they are used in practice by collecting usage data. This technique is used in other application domains, such as web applications. The data can be analyzed in various ways, including our adaptation of the critical incident technique. The software engineering community needs to collect data about the effects of tools on the productivity of programmers before it rushes to automate.

Rather than offering more automation, we took the opposite direction, and proposed the compositional paradigm for automating refactorings. In this paradigm, the tool automates primitive changes whose effects are easy to predict by the programmers. It puts the programmers in control by letting them manually compose the primitive changes into complex changes. The compositional paradigm was
inspired by our analysis of the refactoring practices of programmers in the wild. Our data mining and manual examination of two refactoring usage data sets provided evidence for the prevalence, diversity, rationales, and naturalness of composing automated refactorings. In addition, our survey and lab studies show that the compositional paradigm is more effective than the existing wizard-based paradigm of refactoring. In particular, these studies show that the compositional paradigm is more efficient and less error-prone than the wizard-based one, and programmers are more willing to use the compositional refactorings.

We used the compositional paradigm to automate type qualifier inference (TQI), a program transformation that is not supported by existing IDEs, but is needed to leverage a recent feature of Java. Following the compositional paradigm, we decomposed TQI into a set of primitive changes each of which adds type qualifiers to a single location. Our implementation of the compositional paradigm for the existing automated refactorings of Eclipse requires the programmers to devise the plans themselves to compose the primitive changes. However, CASCADE, our implementation of the compositional paradigm for TQI suggests a plan for composing the primitive changes. It presents this plan in the form of an interactive tree, which presents the primitive changes and their consequences on the errors reported by the type qualifier checker. CASCADE computes this tree through a speculative analysis. That is, it suggests primitive changes to fix the errors reported by the checker, speculatively applies the primitive changes on copies of the source code, and continues to run the checker and apply the changes recursively. A combination of compositional refactoring and speculative analysis makes CASCADE the first universal TQI tool. CASCADE is universal because it takes a checker for a set of type qualifiers as input and produces a TQI tool for those type qualifiers. We conducted a lab study to compare CASCADE with a typical batch TQI tool. The results show that CASCADE is easy to learn, gives more control, is faster, and programmers are more willing to use it.

The compositional paradigm outperforms the existing wizard-based and batch paradigms by reducing the automation level. Although this result may seem counterintuitive, it is not unique to software engineering. Designers of other fields, e.g., aviation, health-care, and manufacturing, struggle with similar problems. What is an appropriate level of automation? What should the role of the human operator be? Often, researchers find that less is more. That is, a modest design, which provides clear,
immediate feedback, outperforms a design with a high level of automation that does not integrate the human operator well [41, 98, 161].

This work can benefit the designers of automation for other software engineering tasks, e.g., fault localization, remodularization, and bug repair. An important step is to ask questions about the appropriate level of automation for the task. If the results of the automation need to be reviewed by programmers and the overhead of the review is high, it may be a sign of over-automation. One possible solution to this problem is to decompose the automation into smaller steps so that the programmers can review small pieces at a time. If the automation may produce solutions other than the ones that the programmers prefer, the designers may want to give more control to the programmers to steer the automation towards their desired solutions. If knowing why the automation produces a particular outcome helps programmers make better decisions, the designers can expose some of the intermediate results of the automation to the programmers to justify the produced outcome. Taking such a human-centric approach to software engineering paves the path to designing automation that programmers want to use.
References


166


[76] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


[163] FindBugs. findbugs.sf.net.


