Improving the Usability of Refactoring Tools for Software Change Tasks

Anna Maria Eilertsen

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2021

UNIVERSITY OF BERGEN

Improving the Usability of Refactoring Tools for Software Change Tasks

Anna Maria Eilertsen

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 22.12.2021

Year: 2021

Title: Improving the Usability of Refactoring Tools for Software Change Tasks

Name: Anna Maria Eilertsen

Print: Skipnes Kommunikasjon / University of Bergen

Acknowledgements

A doctoral dissertation is a large undertaking and one I could never have accomplished without the support of numerous individuals.

I thank my PhD advisors, Assoc. Prof. Anya Helene Bagge and Prof. Gail C. Murphy, for their time and guidance. I have tremendous gratitude for the opportunity to work with both of them, and for their endless encouragement and patience.

A special thanks goes to the participants in the studies that I conducted, for their willingness to share their time and experiences. This research would have been impossible without their help. I also thank the reviewers of the published and unpublished manuscripts for their insightful comments. Thank you to the members of my thesis committee: Massimiliano Di Penta, Olga Baysal, and Mikhail Barash.

I would like to thank my colleagues and friends who supported me through this journey, in particular Bertrand Ong, Tetiana Yarygina, Laura Garrison, Håkon Gylterud, Sofija Ivanova, May-Lill Bagge, and Giovanni Viviani, for their helpful feedback, advice, and discussions.

My gratitude also goes to the administrative staff that helped me navigate the various challenges that arose during these years, during my travels abroad, and most recently, from the COVID-19 epidemic.

Anna Maria Eilertsen
Bergen, Norway, September 27th, 2021

It's like a schoolhouse
of little words,
thousands of words.
First you figure out what each one means by
itself,
the jingle, the periwinkle, the scallop
  full of moonlight.
Then you begin, slowly, to read the whole story.

— *Mary Oliver, Breakage.*

In a word, the computer scientist is a toolsmith
— no more, but no less.

— *Frederick P. Brooks, Jr.*

Abstract

All successful software gets changed. Developers undertake *software change tasks* by editing the software's source code to meet new demands. Often, the edits they make are aligned with *refactorings* and can be performed automatically with *refactoring tools* that are found in the developers programming environments.

Refactorings are time-consuming and error-prone to apply manually. Consequently, automated refactoring tools hold the promise of many benefits, such as the ability to change software faster and with fewer errors. It is therefore surprising that developers largely eschew these tools in favor of applying refactorings manually. Despite the extensive refactoring support of mainstream development environments and despite several decades of research into how these tools can be improved, this *disuse* still persists.

In this dissertation, I address the usability of refactoring tools in the context of software change tasks. In doing so, I aim to identify, understand, and address, barriers that developers face when attempting to use these tools to change software.

I present a study of how developers approach software change tasks that are amenable to automation with refactoring tools. I found that developers approach such tasks with one out of three strategies and that the different strategies give rise to different usability problems that prevent them from integrating refactoring tools into their workflow. I also present a theory of refactoring tool usability, based on the ISO 9241-11 definition of usability, and use this theory to identify four usability themes that developers experience as important in refactoring tool design.

Then, I conduct a survey of a larger group of developers to investigate how they experience different tools that can support refactoring operations, including the tools that developers use to refactor manually. By comparing their experiences, I explore the different tools' usability profiles, that is, the combinations of factors that make developers find that tool useful. These profiles indicate that developers seek different experiences from different tools. This information might help refactoring tool makers identify the needs of potential users or ways in which their tools can be improved.

Finally, I address multiple of the usability themes that I identified as problematic for developers with existing tools. I do this by proposing a stepwise refactoring tool, that is, a tool that allows the developer to initiate a refactoring and apply it step by step, interleaved with other changes, or even with other refactorings.

Taken together, the findings presented in this dissertation provide actionable insights for researchers and toolmakers as well as a foundation from which future researchers can continue the efforts of improving refactoring tools.

List of Publications

1. Eilertsen, Anna Maria, 2020, *Predictable, Flexible or Correct: Trading off Refactoring Design Choices*, in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, (*ICSEW, IWOR'20*) [36]

2. Eilertsen, Anna Maria and Murphy, Gail C., 2021, *A study of refactorings during software change tasks*, Journal of Software: Evolution and Process, John Wiley & Sons Ltd. [41]

3. Eilertsen, Anna Maria and Murphy, Gail C., 2021, *Replication Data for: A Study of Refactorings During Software Change Tasks*, DataverseNO [38]

4. Eilertsen, Anna Maria and Murphy, Gail C., 2021, *The Usability (or Not) of Refactoring Tools*, 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) [40]

5. Eilertsen, Anna Maria and Murphy, Gail C., 2021, *Stepwise Refactoring Tools*, 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) [39].

6. Eilertsen, Anna Maria and Murphy, Gail C., *Usability Profiles of Refactoring Tools*, (unpublished manuscript)

Contents

Acknowledgements **i**

Abstract **iii**

List of Publications **v**

1 Introduction **1**

1.1 Thesis Overview 3

1.2 Contributions 6

2 Background **7**

2.1 Refactoring: A Primer 7

2.2 Refactoring Tools: the Beginnings 14

2.3 Refactoring Tool Usability 17

2.4 Studying Developers' Refactoring Experiences 18

2.4.1 Asking Developers 19

2.4.2 Observing Developers 20

2.4.3 Mining Source Code Changes 21

2.5 Studies of how Developers Refactor 21

2.6 Factors that Impact the Use and Disuse of Refactoring Tools 24

2.7 Software Changes 29

3 A Study of Refactorings During Software Change Tasks 31

3.1 Introduction 31

3.2 Study Design 32

3.2.1 Method 33

3.2.2 Participants 34

3.2.3 Experimental System and Tasks 37

3.3 Data and Analysis 42

3.4 Results 46

3.4.1 RQ1. *What strategies do developers use to approach software change tasks which include refactorings?* 46

3.4.2 RQ2. *How often do developers use automated support for refactoring versus proceeding manually?* 51

3.4.3 RQ3. *How do developers experience refactoring tools?* 54

3.4.4 Results Summary 64

3.5 Implications 65

3.5.1 Improving Refactoring Tools 66

3.5.2 Investigating Refactoring Tool Disuse 67

3.5.3 Bridging the Gap Between Developer Workflow and Refactoring Tools 68

3.6 Threats to Validity 70

3.6.1 Construct validity 70

3.6.2 Internal validity 70

3.6.3 External validity 70

3.6.4 Conclusion Validity 71

3.7 Summary 71

4 Refactoring Tool Usability 73

4.1 A Theory of Usability 74

4.1.1 Application 74

4.1.2 Proposed Theory 75

4.2 Analysis 75

4.2.1 Coding of Transcripts 76

4.2.2 Analysis of Codes 77

4.2.3 Analysis of Comments 78

4.3 Results 80

4.3.1 Prevalence of Usability Factors 80

4.3.2 Usability Factor Themes 81

4.3.3 Usability Themes 82

4.4 Discussion 83

4.4.1 Theory of Refactoring Tool Usability: Revisited 83

4.4.2 Implications for Refactoring Tool Designers 84

4.4.3 Relationship to Earlier Work 87

4.5 Summary 88

5 Usability Profiles of Refactoring Tools 91

5.1 Introduction 91

5.2 Methodology 93

5.2.1 Scope 93

5.2.2 Design 94

5.2.3 Data Collection . . . 100
5.2.4 Analysis . . . 101
5.3 Results . . . 102
5.3.1 **RQ1.** *Which tools do developers find useful for software change tasks?* . . . 102
5.3.2 **RQ2.** *Which usability factors—efficiency, trust, effectiveness, predictability, satisfaction—do developers deem as necessary for a tool?* . . . 104
5.3.3 **RQ3.** *Do change tasks impact developers' experience of usability factors?* . . . 107
5.4 Discussion . . . 108
5.4.1 Usability Profiles of Tools . . . 108
5.4.2 Developer Strategies and Preferred Usability Profiles . . . 112
5.4.3 Threats to Validity . . . 113
5.5 Summary . . . 114

6 Stepwise Refactoring Tools 115

6.1 Introduction . . . 115
6.2 Motivating Example . . . 117
6.2.1 Bob and the "Break-and-Fix" Approach . . . 118
6.2.2 Alice and the Awkward Automated Approach . . . 119
6.2.3 Charlie and the Controlled Approach . . . 119
6.2.4 Relevance to Developer Participants . . . 121
6.2.5 Usability Problems . . . 124
6.3 A Stepwise Refactoring Tool . . . 125
6.3.1 Bob and the Stepwise Approach . . . 128

6.3.2 Alice and the Stepwise Approach 130
6.3.3 Charlie and the Stepwise Approach 133
6.4 Discussion and Future Work 134
6.4.1 Tool communicates capabilities 134
6.4.2 Tool communicates change 135
6.4.3 Developer guides tool 135
6.4.4 Developer cost-benefit analysis 137
6.4.5 Implementation 138
6.5 Summary 140

7 Discussion and Future Work 143

7.1 Studying Developers' Refactoring Experiences 143
7.2 Refactoring Tool Usability 144
7.3 Refactoring Tool Support 146

8 Conclusion 151

A Refactoring Patterns 153

A.1 Remove Parameter 153
A.1.1 Conditions 154
A.2 Move Method 155
A.3 Inline Method 156

B Survey Questions 157

Chapter 1

Introduction

> In the beginner's mind there are many possibilities, but in the expert's mind there are few.
>
> — *Shunryu Suzuki, Zen Mind, Beginner's Mind*

All successful software gets changed [25]. After its initial creation, software developers perform a multitude of changes to maintain and improve the software system, both manually and with the help of tools. These changes vary in size, in complexity, and in their motivating reasons [58, 124, 99]. For example, the goal of such changes can be to introduce new features, correct mistakes or misunderstandings, or to react to changes in the software's environment [99]. Despite this substantial variability, one similarity is that software change tasks almost always involve modification of code by a developer.

Many of the code modifications that developers perform when changing software are aligned with refactoring operations, that is, a change to the program's structure that does not impact its observable behavior [47]. These modifications can be supported by automated refactoring tools [26, 47, 35, 91, 87] that developers have access to in their environments. For example, the standard installations of mainstream integrated development environments (IDEs) for the popular object-oriented languages Java (e.g., IntelliJ [3] and Eclipse[1]) and C# (e.g., Visual Studio[16]) each offer their users automated tools for over 40 refactoring operations. In theory, developers can reduce their manual workload by using refactoring tools to automatically perform subsets of their code edits. However, in practice, multiple investigations have shown that developers eschew refactoring tools in favor of editing the code manually [86, 67, 91, 112] and refactoring tools

are, in fact, *disused* [126].

Empirical studies indicate that developers avoid using refactoring tools that are present in their programming environments—even when they are aware of them—due to usability issues [67, 86, 84, 126, 112]. Researchers have proposed various approaches to the reported usability problems [90, 71, 85, 101, 127, 45, 50, 49], most of which are aimed at improving singular usability aspects such as speed [66], selecting the right code segments [84] or choosing the right refactoring operation [45, 49, 50].

This previous research largely considers refactoring tools in isolation and on subgoals that are taken out of context of an overall change goal (e.g., [84, 50]). Typically, developers decompose the overall goal of a change task into subgoals that are directly supported by tools [109]. While studies of tools in isolation provide valuable insights and enable improvements to individual tools, the improvements are not always compatible with the use of the tool within a larger change. For example, investigating how developers experience the `extract-method` tool when they are specifically instructed to perform `extract-method` refactorings [84] does not provide insight into barriers that prevent them from using the same tool to support a functional software change [118]. Moreover, despite the substantial research into developers' refactoring activities (e.g., [67, 86, 126, 91, 112, 72, 74, 95, 51] and [17] for a survey), it is difficult to merge disparate study findings since they focus on different tasks and experience measures as well as different definitions of refactoring.

Consequently, little is known about how developers experience the usability of refactoring tools as support for software changes. In fact, there is no comprehensive approach to refactoring tool usability. Moreover, despite some knowledge about which types of tools developers employ instead (e.g., compiler errors and search tools [50, 86, 126, 51]), we do not know which factors affect their choice of which tool to use and how they experience the different tools during software change tasks.

The goal of this dissertation is to bridge this knowledge gap. I take the perspective that refactoring tools can be be made more useful by addressing usability impediments that developers encounter when attempting to use them. To do so, I explore refactoring tool usability in the context of software change tasks, guided by the following thesis statement:

> Improving the usability of refactoring tools for use as part of functional software change tasks can enable developers to make code modifications more quickly and more correctly for these often occurring tasks.

This perspective on refactoring tool usability places refactoring tools in the context of

software change tasks. Consequently, the center of my research on refactoring tools is the developer's ability to progress on a software change task.

The focus on usability is in line with previous research that identified the usability of refactoring tools to be of high importance for developers (e.g., [76, 77, 84, 86, 126, 87]). The context of software change tasks is in line with early research into refactoring, which identified refactoring as code modification patterns that occur during software changes [94, 55, 63, 35]. This research also identified the potential of automated refactoring tools as developer support [55, 104, 63]. Moreover, this context aligns with recent research that identified functional changes and code reuse to be the context in which developers apply refactorings [74, 112, 95]. My perspective differs from the alternative perspective that refactoring tools automatically perform smell-removal [122, 23] or are applied for the purpose of automatically improving software quality [33, 29].

My perspective also identifies refactoring tool design as a *wicked problem* [34, p.84], that is, as an ill-defined problem with many interdependent factors where the effort to solve one aspect of the problem may reveal or create other problems. Similarly, I will describe how efforts at improving certain aspects of refactoring tools can worsen other aspect of the same tools. Consequently, a large part of this thesis is dedicated to understanding and defining the problem of refactoring tool *disuse* by before I attempt to propose a solution, as illustrated in Figure 1.1.

The remainder of this chapter contains an overview of this thesis, and a summary of its contributions.

1.1 Thesis Overview

Chapter 1 introduces this dissertation, describes the knowledge gap that I aim to bridge, and presents my thesis statement and the contributions of this dissertation. Each chapter is described in turn along with the associated publication(s).

Chapter 2 presents related work that contextualizes this research and contains a list of factors that previous works has suggested contribute to refactoring tool disuse. This chapter includes a brief primer on refactoring operations and refactoring tools. The primer focuses on the IntelliJ IDE that is used throughout this work to showcase state-of-the-practice refactoring tools.

A short paper that discussed the various factors that contribute to disuse has been published in workshop proceedings [36].

Chapter 3 presents the results of a novel study that investigates barriers developers face in using refactoring tools as part of functional software changes. I observed 17 individuals with software development experience attempt to solve three software change tasks that included functional changes and are amenable to refactoring tools, and conducted interviews with each participant immediately afterwards.

The data collected with this study constitute the largest dataset in this dissertation, consisting of 32 hours of screen recordings and transcripts of 130 000 words. By analyzing this dataset, I was able to identify three types of approaches, or *strategies*, that participants used to progress on the tasks. I was also able to identify use and disuse of refactoring tools that the developers had available to them during the task as well as collect information about experiences that they have had with such tools during their regular work.

The work presented in Chapter 3 has been published in a journal article [41]. A replication package that includes the dataset has been published in the DataverseNO system for storing scientific data [38].

Chapter 4 presents a *theory of usability for refactoring tools* and an analysis of usability themes that were experienced by the participants in the study that were identified through this theory. To address usability in a disciplined way, I operationalized the ISO 9241-11 definition of usability [5] to create a theory of usability for refactoring tools. This is a similar approach to the one taken by Mealy et al. [77], however, while their operationalization relied on the refactoring process laid out by Mens and Tourwe [78] and the perspective of refactoring as smell-removal tools, I take the perspective of refactoring tools as support for functional software change tasks and base my operationalization on recent research into the ways that refactoring tools are used. I also use the dataset described in Chapter 3 to refine the theory and show its utility in identifying usability themes in the transcripts from the developers.

The work presented in Chapter 4 has been published in conference proceedings [40]. The dataset that this chapter is based on is included in the DataverseNO repository [38].

Chapter 5 presents the subsequent development of a survey instrument to validate how a larger demographic of developers experience the usability of refactoring tools. The survey was developed using the usability factors identified in Chapter 4 and focuses on the tools and approaches that developers were observed to use in Chapter 3. In this survey, I also investigate how the usability of refactoring tools compare to that of other tools that developers use to approach refactorings during software change tasks. This includes both refactoring tools and other tools or information sources such as search-tools, navigational tools, and even compiler errors.

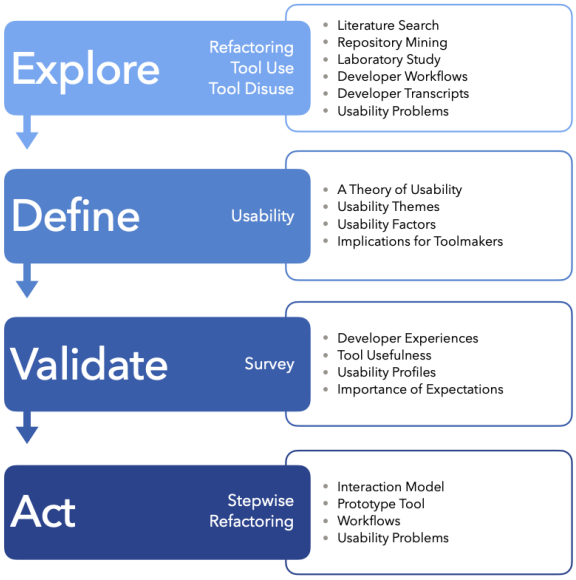


Figure 1.1: In this dissertation, I first **explore** the problem—*the disuse of refactoring tools due to usability problems*—by surveying literature (Chapter 2) and performing a laboratory study (Chapter 3). From this fundament, I **define** usability (Chapter 4) and **validate** the insights made so far by surveying additional developers (Chapter 5). Then, I **act** on these findings by proposing an alternative interaction model that addresses multiple of the usability problems that contribute to the disuse of refactoring tools (Chapter 6).

This survey uncovered variability in the needs that developers have from different tools and shows the possibility of creating usability profiles to capture this variability. This manuscript is unpublished. The full survey is included in Appendix 5.

Chapter 6 considers how refactoring tools can be altered to address the insights that presented in previous chapter. It presents a prototype of a stepwise refactoring that alleviates several of the usability problems presented in this thesis and in previous work. This chapter also discusses how the traditionally emphasized behavior-preservation can be relaxed in favor of increasing the user's control of the tool and access to information. The work presented in this chapter will appear as part of conference proceedings [39].

Chapter 7 contains a discussion of the work in this thesis and suggests future directions of research, and **Chapter 8** concludes this work.

1.2 Contributions

This dissertation makes several contributions in the areas of theory, empirical findings, engineering, and replicability of research. Here, the contributions are listed according to area.

Theory:

1. A compilation of factors that contribute to refactoring tool use created by surveying related works [36] (Chapter 2).
2. A theory of usability for refactoring tools [40] (Chapter 4).

Empirical Findings:

3. Three strategies that were observed to be used by developers to make progress in functional software change tasks [41] (Chapter 3).
4. Four usability themes that illustrate problem areas of refactoring tools in the context of software change tasks [40] (Chapter 4).
5. Usability profiles of nine types of tools that developers find useful for approaching software change tasks, including simple and complex refactoring tools (Chapter 5).

Engineering:

6. An interaction model for a stepwise refactoring tool, and a prototype of such a tool, that showcases ways to address multiple of the usability issues that were previously identified [39] (Chapter 6).

Replicability of research:

8. An experimental system that can be reused by other researchers to study realistic software change tasks that contain steps amenable to refactoring tools [38] (Chapter 3).
9. A dataset comprising 32 hours of screen recordings from 17 experienced practitioners with time stamped transcripts totaling 130k words and 100 refactoring invocations [38] (Chapter 3).

Chapter 2

Background

Mind the Gap

— *Transport for London*

Refactoring has an almost thirty year history of study. In this chapter, I provide an quick introduction to refactorings and refactoring tools before presenting an overview of early work in refactoring in broad strokes, emphasizing the ideas that shaped the later research. Then, I provide a more detailed description of work related to the usability of refactoring tools for software changes and studies of how developers refactor. I present an overview of the factors that have emerged from previous works as potentially impacting developers' choice of using or not using refactoring tools. And finally, I discuss studies of software change tasks as they relate to the topics in this dissertation.

An early version of the discussion in this chapter was published elsewhere [36].

2.1 Refactoring: A Primer

This thesis relies upon knowledge of refactoring operations and refactoring tools. Here, I provide a short primer on these topics. Additional relevant background can be found in other literature [94, 47].

To *refactor* a program is to change its internal structure without altering its observable behavior [94, 47]. Typically, refactorings are performed to make the program easier to understand or change [47, 64, 112]. For example, the meaning of a function may be clarified by updating its name. Developers often refactor programs in the context

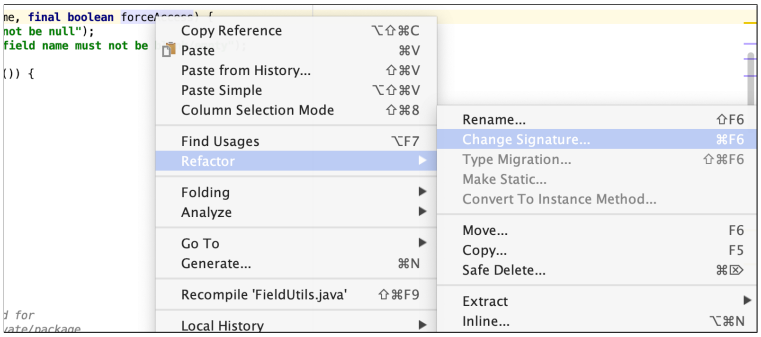


Figure 2.1: IDEs like IntelliJ give developers access to invoke a refactoring menu directly from the editor by right-clicking on a source code element. Rather than showing all the refactorings that the IDE implements, the menu is populated with the refactorings that are possible to invoke on the element that was clicked. Here, the menu was invoked by clicking on a parameter of a method.

of a functional change [86, 74, 95, 35]. Despite the refactoring itself being behavior-preserving, the overall goal of the change might be to implement new functionality or fix a bug [104]. For example, if the behavior of a function is changed, its name might need to be altered accordingly.

A *refactoring operation* is a specific change pattern that often occurs during changes to object-oriented[1] programs [94, 47]. For example, a developer can apply the `rename` operation to a method declaration to clarify its meaning. However, it is not enough to change the declaration: in order to retain the program's meaning, all callers must also be updated to use the new name. If the developer makes this change manually, they must locate and update all callers [47]. However, if the developer initiates the `rename` operation with a refactoring tool, then the tool can make the updates automatically.

Many refactoring operations are automated by *refactoring tools* found in modern mainstream IDEs such as IntelliJ [3] and Eclipse [1]. Figure 2.1 shows one refactoring menu from IntelliJ[2]. Refactoring tools support software change tasks by automatically performing steps of the refactoring process that developers would otherwise have to do manually. For example, the tool can automatically update all locations that are impacted by a refactoring, such as callers in the previous `rename` example. This is especially useful in cases where the initial change impacts many locations (e.g., in the case of applying `rename` to a method with many callers located in different files). In this case, a tool can save the

[1]When the program is not object-oriented, these changes are typically called *restructuring* rather than refactoring [55].

[2]Throughout this thesis, I will use examples from the JetBrains IntelliJ IDE because it is representative of the support in mainstream IDEs. JetBrains is known for producing high-quality IDE support for refactoring such as in the IntelliJ IDE for Java and the Visual Studio plugin ReSharper for C#.

developer from slow and error-prone navigation, editing, and context-switching.

While this depiction of refactorings might evoke associations to simple textual find-and-replace tools, implementing such tools are significantly more difficult. Consider a developer that wants to `rename` a variable named `i` to be called `foo`. A tool that replaces all `i`s in the entire program text, with the string foo would not be considered useful as, for example, every reference to the `int`-type would now be replace with the nonsensical `foont` type. To ensure that a refactoring tool changes the correct code, it needs to operate on a parsed version of the program. This typically entails operating an abstract syntax tree (AST) or a similar data structure, like the Program Source Interface (PSI) type used by IntelliJ [59]. The refactoring tool typically changes this AST or PSI representation of the program and the textual source code is updated accordingly.

To ensure that the changes are safe and correct, a refactoring tool typically performs *condition checks* [94, 22, 42]. Consider again the case of the `rename` operation on a method. If a developer attempts to give the method a name that already is defined in that scope, for example the name of a pre-existing field, then the resulting program will fail to compile. Whilst this might be difficult to identify in a manual process until the refactoring is already applied, due to mechanisms like inheritance, a tool can easily check this before the code is changed. If the name is already there, the tool can stop the application and alert the programmer that the new method signature will collide with an existing element.

For a running example of a refactoring, I introduce the `remove-parameter` refactoring[3]. This refactoring removes a parameter from a method declaration and updates all callers correspondingly. Its conditions are that the parameter is not used in the method's body and that the new signature does not collide with any other method signatures. Listing 2.1 shows a simple example of such a program and Listing 2.2 shows the result of applying `remove-parameter` to that program.

[3] `remove-parameter` is sometimes considered part of `change-signature` [47, 3] and invoked through the `change-signature` menu option.

Listing 2.1: A method `bar` with two parameters, and one caller, `foo`. The parameter `Y y` is unused.

```
60 public boolean foo(A a) {
61    return bar(a.getX(), a.getY());
62 }
63 public boolean bar(X x, Y y) {
64    return x.isBar();
65 }
```

Listing 2.2: The unused parameter `y` was removed from `bar` and from its call in `foo`.

```
60 public boolean foo(A a) {
61    return bar(a.getX());
62 }
63 public boolean bar(X x) {
64    return x.isBar();
65 }
```

If a developer wants to invoke a refactoring tool from an IDE, they can access a refactoring menu like the one shown in Figure 2.1 and select the desired refactoring from the list. Next, a Configuration View is presented, as shown in Figure 2.2. This view lets the developer configure the `change-signature` refactoring as a `remove-parameter` refactoring. This view is typically shown as a popup over the code editor and blocks the developer from making any changes to the source code while they are interacting with the view.

Once the developer finishes the configuration, she can Preview the refactoring to see a list of locations that will be changed, as shown in Figure 2.3. The developer may also click Refactor to immediately apply the entire change without previewing it. In both cases, the tool will check that refactoring's conditions and present a problem view if it detects any violations. Figure 2.4 shows such a view for the `remove-parameter` refactoring. Also in this case, the developer can apply the refactoring by clicking Continue in Figure 2.4.

In the Preview View, it is often possible to navigate to the identified locations and to exclude locations from being updated, as shown in Figure 2.3. When the developer is satisfied with the Preview, she may click Do Refactor to apply the refactoring. While some implementations of Preview View (like Eclipse) prevent the developer from interacting with the code, the one shown here, from IntelliJ, affords the developer the opportunity

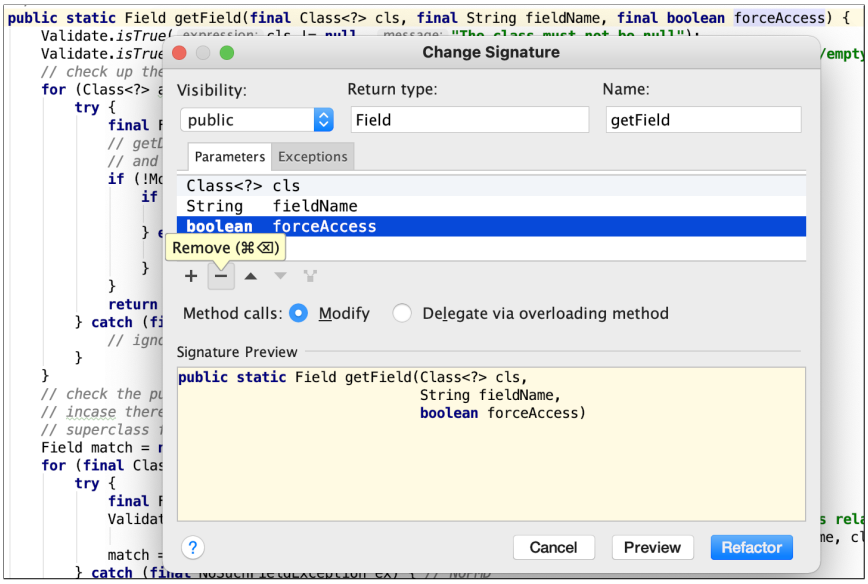


Figure 2.2: The configuration view of `change-signature` in IntelliJ. Here, the developer can configure the refactoring tool to apply `remove-parameter`. Once the developer has finished the tool's configuration, she can click Refactor to make the tool attempt to apply the refactoring. She can also choose to Preview the configured refactoring operation or use Cancel to discard the refactoring. While this view is shown, it is not possible to edit the source code shown in the editor behind the popup.

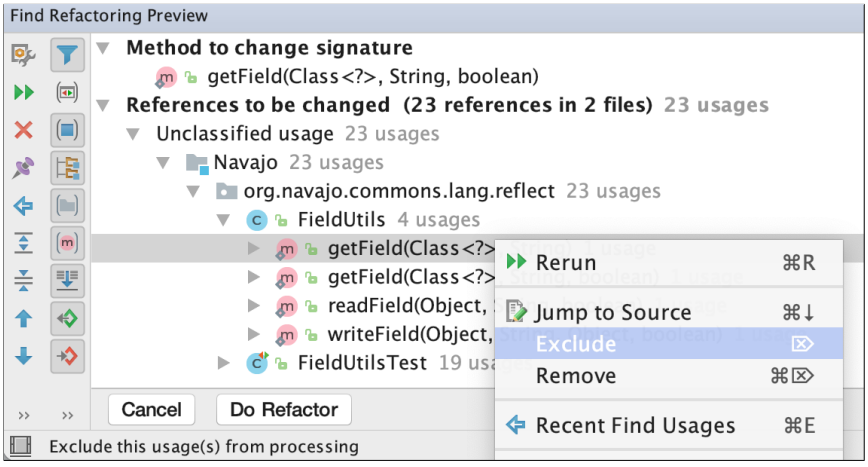


Figure 2.3: IntelliJ's Preview view for an application of `change-signature` to the method `getField`. The tool is configured to remove a parameter from the method. This Preview View shows all the 23 references that needs to be updated for the refactoring to preserve behavior. For each reference, a user may choose to "exclude this usage from processing". The Preview View is shown in a paneø that allows editing the source code; however, doing so will render the refactoring invalid and raise an error.

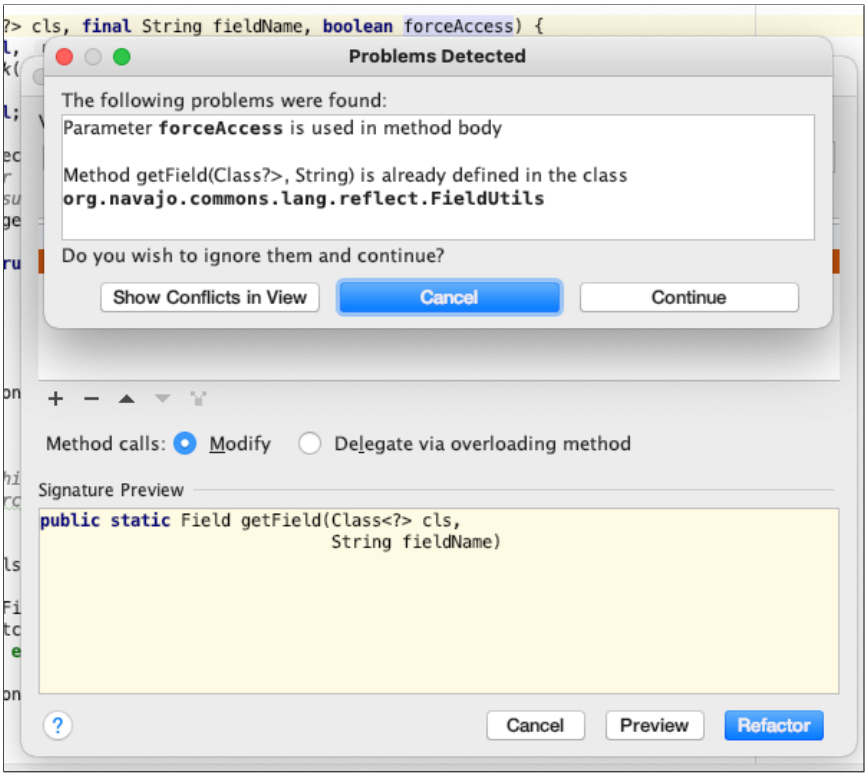


Figure 2.4: Problem view for the `change-signature` as a popup over the configuration view. The condition checker has found two problems, both of which are going to lead to compiler errors if the refactoring is applied. The user is given the option to learn more, to cancel, or to continue the application despite the problems.

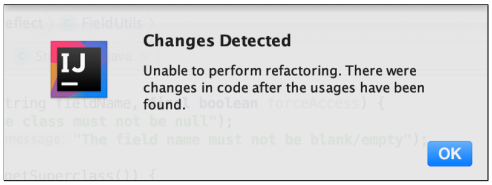


Figure 2.5: If a developer alters the source code during the preview step of a refactoring in IntelliJ, that refactoring can no longer be applied. Instead, this error is produced, and the refactoring must be re-invoked and re-configured before application.

to alter the source code before continuing the refactoring. However, if they choose to do so, the refactoring tool will prevent the application of the refactoring. In this case, the developer is forced to re-initiate and re-configure the refactoring. Figure 2.5 shows the error message that IntelliJ presents in such a situation. This is due to the tool's reliance of the code being unchanged to ensure that the refactoring and the conditions that has been checked are still valid.

Once the refactoring is applied, the tool changes all the identified locations in the source code according to its implementation, except any excluded locations.

Figure 2.6 illustrates the refactoring tool workflow supported by refactoring tools in IDEs. It shows that after a developer invokes and configures a refactoring tool, the tool transforms the source code according to that tool's implementation. As the exact transformation can vary [93], developers may be unable to correctly predict the outcome of a tool invocation [92] and risk applying transformations that they did not intend [86, 126]. It can even be difficult to inspect or verify the changes after they happen [40]. Regardless of how complex and distributed a change is, the tool performs the transformation in a single, atomic step. The resulting all-or-nothing situation forces the developer to choose between applying one atomic change for which they cannot predict the outcome, or editing the code manually.

As indicated in Figure 2.3 and Figure 2.4, the mainstream refactoring tool lets the user apply unsafe refactorings. In Figure 2.3, the user can exclude references from being updated. In this case, these references might change the program's behavior, either by preventing compilation or by altering references. For example, if a parameter is removed and one caller is excluded, as indicated in Figure 2.3, that caller will no longer bind to the right method and the program will not compile. If a removed parameter is referred to in the method body, as shown in Figure 2.4, a similar problem will arise. As another possibility, if there existed a field with the same name in that class, the program might still compile, but its behavior might have been changed.

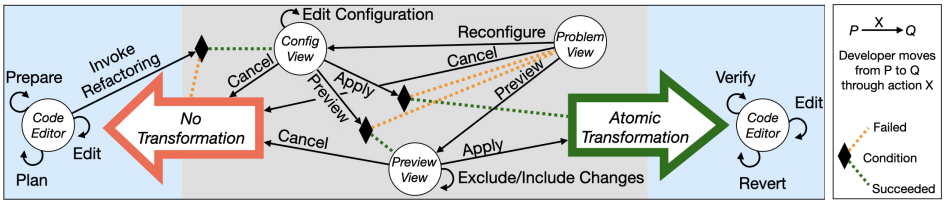


Figure 2.6: The design of most refactoring tools makes the developer "leave" the code editing context (shown in blue) when invoking a tool and "return" either by applying an entire transformation (right green arrow) to the source code or by discarding the change (left red arrow). While interacting with the tool (gray box), the developer can inspect and edit the upcoming refactoring (i.e., navigate the black lines). The Config(uration) View is used to edit the refactoring. The tool can check for problems (⧫) and inform the user about violated conditions (Problem View) and about the locations that will be impacted (Preview View). If the change is discarded, no configurations, transformations, or other information is retained.

2.2 Refactoring Tools: the Beginnings

The first organized examination of refactorings were done by Opdyke [94] and Griswold [55] in their dissertations in the early 1990s. Griswold described a number of restructurings that could ease program maintenance without changing a program's behavior, such as `rename`, `extract-method`, and `inline-method`. He also demonstrated that these types of transformations could be automated. Opdyke studied the evolution of object-oriented programs in C++ and created a catalogue of restructurings, or *refactorings*, that frequently occurred and were useful during the program's evolution. Included in the catalogue were the conditions under which each refactoring could be applied to a program written in C++ without altering its behavior. Shortly after, Roberts, Brant, and Johnson implemented a number of these refactorings and their conditions in the Smalltalk Refactoring Browser [104, 105].

These early works shaped much of the later research in two ways. Firstly, the specific refactorings that were described, catalogued, and automated, are still largely the same that are found in modern IDEs today [1, 3]. Secondly, the emphasis on preserving program behavior is still considered as essential to the refactoring discipline. Despite the incompleteness of Opdyke's conditions [120], the idea of combining code transformation with correctness checks has prevailed. In fact, the refactoring tools that are found in modern IDEs also use condition checks to guard against erroneous applications.

The interest in refactoring was fueled by its promise of a faster and cheaper means of evolving and improving object-oriented software. This was illustrated in Tokuda and Batory's study of refactoring-aided software evolution [120]. They selected two programs

that had been manually evolved by developers, and for each program, they selected two versions of their design, the source and target state. Then, they attempted to use refactoring operations to transform the source state into the target state. Based on this study, they estimated how much time could have been saved if the two programs had been evolved with the aid of tools that could automatically apply the refactorings rather than with a manual approach. For one program, they estimated that the time could be reduced from two days to two hours and for the other program, from two weeks to one day. The possibility of such drastic improvements on developer's productivity was an enticing prospect and motivated the development of automated refactoring tools.

There are substantial technical challenges associated with implementing refactoring tools [104]: they require non-trivial program analysis and transformation capabilities[59]. The details of correctness conditions and program transformations vary between languages which means that is difficult to reuse tool components written for one programming language in a tool targeting another. One example of this is the Refactoring Browser. This was an early example of automated refactoring support but only supported Smalltalk. This tool could not be used to refactor programs that were written in other languages, such as Java and C++.

In the late 1990s, there was consequently a sizable demographic of Java programmers in the industry who had little or no support for automated applications of refactorings. Instead, these practitioners were introduced to refactoring operations through Fowler's seminal book, Refactoring: Improving the Design of Existing Code [46]. This book described how and when to apply a catalogue of refactoring operations. While this catalogue largely contained the same ones as those described by Opdyke and Griswold and the ones implemented in the Smalltalk tool, the descriptions were aimed at practitioners without tool support. The book used a natural language aimed at practitioners to describe how to manually apply these operations. It also emphasized that the program's behavior should be preserved, but it recommends achieving this through small and controlled code changes and frequently running a test suite.

In 2004, Mens and Tourwé presented the first survey of software refactorings [78] and introduced a model of the refactoring process:

1. Identify where the software should be refactored.
2. Determine which refactoring(s) should be applied to the identified places.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.

5. Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).

6. Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

At this point, there was already substantial research into how to implement and improve refactoring tools. This research focused largely on technical improvements of the tool's ability to perform one or more of these steps. For example, researchers attempted to create refactoring recommendation tools that could support steps 1-2 [62], worked on implementing a wider variety of refactorings [107, 73, 104] (step 4) and implemented condition checks [30, 119, 79] (step 3). Condition checks are important because they help ensure that the program's behavior is preserved. However, Mens and Tourwé described that, in many cases, a formal guarantee for behavior preservation would cause tools to be overly restrictive; Brant later made a similar observation [24]. Indeed, subsequent research would develop to use refactoring either in the formal sense (e.g., [42, 107, 22, 116]) or in a more pragmatic way that included, for example, unsafe refactorings, bug-fixes and API-level changes (e.g., [67, 108, 72, 35]). The latter, pragmatic kind of refactoring research is often more developer oriented and is the category to which the research in this dissertation belongs.

Despite the influence of Mens and Tourwé's model, it did not accurately describe the workflow that developers use. Steps 1 and 2 were commonly thought of as the detection and removal of *code smells*, that is, of patterns in source code that makes the code difficult to work with, for example due to overly long methods or too deep nesting of logical structures [46]. However, later research into how developers refactor (Chapter 2.4-2.5) found that this is not the main motivation for the application of refactorings. Instead, developers rather apply refactorings frequently, in small steps, and often interleaved with other changes [87, 126, 67, 91] for the purpose of preparing or completing functional software changes [112, 74, 95]. Muprhy-Hill et al. [86] coined the term "floss-refactoring" for these types of changes and found that developers often apply them during functional changes to avoid introducing smells in the first place. Roberts, Brant, and Johnson also describe this workflow in their original paper on the Refactoring Browser [104]. However, some of the early works on refactoring tool usability was made under the assumption that the tool should automate the process that Mens and Tourwé had laid out (e.g., [77]), which includes locating and removing code-smells. In contrast, developers were refactoring to prevent the smells from ever occurring.

These discrepant workflows can give rise to a number of the usability problems once developers try to replace their manual process with existing refactoring tools. For a simple example, consider a refactoring tool that is designed to ensure that program behavior is preserved. When a developer invokes this tool, it might check behavior-related conditions and even present the developer with information about any potential changes to the program's behavior. However, the developer might be applying the refactoring with the intention to progress towards a software change goal, and consequently, they may be concerned both with the behavior changes but also with how the refactoring tool will impact subsequent steps in their workflow. They may, for example, worry that the tool will prematurely remove information that they need to understand other areas of the code, or that the tool will make it more difficult to recover impacted areas later in their workflow. Such usability problems can occur even with tools that are correct.

2.3 Refactoring Tool Usability

The first investigation of refactoring tool usability was done by Mealy and Strooper in 2006, five years after the first refactoring tool crossed Fowler's Refactoring Rubicon by supporting `extract-method`. The reason why usability of refactoring tools is important is because the enticing promise of refactoring to make software evolution faster and cheaper by order an order of magnitude [120] is realized only if developers use the tools that are available to them. If the tools have usability problems, developers may fail to use these tools or fail to achieve such gains when trying to use them.

Mealy and Strooper performed a theoretical evaluation of the usability of six refactoring tools for Java by comparing their capabilities to the steps in Mens and Tourwé's model of the refactoring process [76]. They concluded that "usability of refactoring tools requires further research/consideration". Next, Mealy et al. operationalized the ISO 9241–11 interface design standard [4], again using Mens and Tourwé's refactoring model, and distilled a list of 81 usability requirements for refactoring tools [77]. While this work was an important early investigation into the usability of refactoring tools, the premise of the refactoring process was later found to be subtly different. For example, it was found that developers largely do not use refactoring tools to remove smells, but rather as part of functional code changes. Thus, some of the requirements formulated by Mealy et al. are less relevant to tool usability than first assumed.

For example, one requirement from Mealy et al. is that the tool must "[a]utomate code-smell identification" [77], which corresponds to Step 1 in Mens and Torwé's model [78], so for a tool to fully support their model, this is indeed necessary. However, later re-

search has found that refactoring is largely performed as part of functional changes and for preventative purposes rather than to remove existing code-smells [112, 87, 95]. Consequently, this usability requirement is not necessary for a tool that supports refactoring.

Another guideline is that the tool must "[a]llow suggested changes (to the system being refactored) to be viewed, changed, or cancelled prior to transformation" [77]. Interestingly, this guideline is in line with current refactoring tools and the practice of offering the user a Preview View and a Configuration View before the transformation is applied (Chapter 2.1). However, no studies have shown the effectiveness of Preview Views. In fact, early studies of developers found that many developers do not inspect these views [126] a recent survey of developers finds that many developers do not find these views effective even today [51].

I take a similar approach to Mealy et al. in Chapter 4 where I operationalize the usability framework put forth in ISO 9241-11 [5]. However, the approach I took is different in the following aspects: 1) I based the operationalization on recent findings into how and why developers refactor that differ significantly from the process put forth by Mens and Tourwé [78] and that Mealy et al. relied on, and 2) I refine the resulting framework based on experiences from practitioners and distill a theory of refactoring tool usability that can be reused by researchers and practitioners, rather than creating specific usability requirements and performing a theoretical evaluation of tools.

2.4 Studying Developers' Refactoring Experiences

While the early research into refactoring tools was largely based on a theoretical understanding of how refactoring could be used to evolve software [121, 35, 94], later research began investigating how developers use refactorings. Learning how developers apply refactorings manually or how they use existing tools can enable the development of more effective and satisfying tool support. For example, by learning how often individual refactorings are applied, toolmakers can prioritize automation of the most frequently performed refactorings, and by learning ways in which existing refactoring tools are inadequate, toolmakers can improve these aspects of the tools. For these reasons, it is interesting to learn about how developers refactor and whether they use refactoring tools. If they use tools, it is useful to know how they experience these tools, and if not, it is interesting to learn why they are avoided.

There are multiple ways to investigate these questions. One way is to *ask* developers about their refactoring habits through surveys or interviews. Another is to *observe*

developers as they change software; for example through on-site observational studies, laboratory studies, or by recording invocations of refactoring tools in their IDEs. One can also *mine* refactorings from source code changes, such as the ones found in open source code repositories.

2.4.1 Asking Developers

The approach of surveying and interviewing developers is a popular way of learning about their refactoring behavior. For example, Kim et al. surveyed developers at Microsoft [67], Sharma et al. surveyed software architects at Siemens Corporate Development Center India [108], Murphy-Hill and Black surveyed participants at the Agile Open Northwest conference [89], Leppänen et al. interviewed software architects and developers from Finnish software companies [72], and Tempero et al. surveyed a large number of software developers [118]. Both approaches to collecting information have merits and drawbacks.

Surveys provide a scalable and cost-effective way of collecting information from developers. It is often preferred due to the high number of responses that can be collected. However, these surveys are usually kept short and brief and consequently, questions might be kept somewhat shallow. For example, surveys provide limited possibility for follow-up questions. In contrast, interviews tend to be conducted in person and are more time-consuming and less convenient for both the researchers and the subjects. However, they afford researchers the opportunity to explore the subject more in-depth and to ask clarifying questions from the subjects. A mix of the two can also be utilized by performing "interviews" via email to inquire about, for example, source code changes that the subject has performed during a study [87] or in open-source repositories [112].

In the context of research into refactoring, there is an important limitation of both surveys and interviews. Any methodology that requires developers to respond to questions about their refactoring and experiences with refactoring tools, suffers from a potential terminology gap in the use of the term "refactoring".

The researcher might use the term "refactoring" to refer to a change pattern from a catalogue [46], or to only special kinds of code-editing activities that preserve the program's behavior [94, 98]. Meanwhile, the participant might consider "refactoring" to refer to bug-fixes, library-upgrades, and to larger architectural changes such as turning a monolithic software application into micro-services [67, 72, 108, 118, 98]. Leppänen et al. describe their participants' usage of the term in this excerpt from their paper: *"Interviewees usually reserved the word "refactoring" for restructuring and redesigning the system, as in preparatory or planned refactoring. They didn't consider daily small*

changes—for example, method renaming and moving the code around a bit—as refactoring." [72] A similar observation was made by Kim et al. [67] in their study of Microsoft developers. One of their participants stated: *"[Refactoring is] changing code to make it easier to maintain. Strictly speaking, refactoring means that behavior does not change, but realistically speaking, it usually is done while adding features or fixing bugs."* As a result, developers may agree with statements like this one *"Refactorings supported by a tool differ from the kind of refactorings I perform manually."* [67] because they are referring to the lack of tools for micro-service migration.

Moreover, practitioners may be unfamiliar with most or all refactorings that are supported by tools [51]. Consequently, findings from such studies are limited because practitioners may respond based on their interpretation of the concept of refactoring or their awareness of tool support. For example, a developer may estimate that they perform 80 % of their refactoring with tools while the real number is 10 % based on the existing tool support.

2.4.2 Observing Developers

Observational studies offer an alternative to surveys and interviews, and have the benefit that the *actual* developer behavior is collected rather than their self-reported behavior. In the past, these studies were time-consuming because, similarly to interviews, an observer needed to be present. Today, advances in the techniques for collecting interaction data from IDEs enable large-scale observational studies of refactoring "in the wild". For example, the Mylyn monitor[4] has been used in multiple such studies of refactoring tool use [86, 126]. This enables investigating how developers interact with refactoring tools that are available to them in their IDEs during their usual work tasks. Through such studies, researchers can recognize cases of repetitive invocations, cancellations, and reversions of refactorings as well as common configurations and patterns of batch applications [125, 91, 87].

However, when observations are made during regular work tasks, it can be difficult to compare between developers because they might use different IDEs, work on different codebases, and use different tools and plugins. Moreover, there might be difficult to get access to developer's environments due to concerns about security and code ownership. If an observational study is conducted as a laboratory study, then it is challenging to create realistic tasks. In those cases, the tasks used in previous work has typically investigating refactoring tool use in isolation from other tools, on subgoals that are taken out of context of an overall change goal (e.g., [84, 50]). While such studies provide valuable

[4] `https://www.eclipse.org/mylyn/`, accessed August 14, 2021

insights and enable improvements to individual tools, the improvements are not always compatible with the use of the tool within a larger change. It is also non-trivial to merge these disparate study findings into a framework that is shared between tools, since they focus on different tasks and experience measures.

2.4.3 Mining Source Code Changes

Mining studies have become popular ways to quantify the occurrences of individual refactorings and evaluate their impact on quality measures. With the increasing popularity of open-source version control systems, it became possible to apply mining tools to the source code histories that are stored in these repositories. RefactoringMiner [123] is a state-of-the-art tool for this purpose and the tool that I used to locate the commits that I used to build tasks for my laboratory study. By developing algorithms that detect refactorings in these version histories, researchers could investigate how refactorings occurred "in the wild" on a larger scale than before.

By combining this with the aforementioned approaches, researchers can even investigate the degree of use and *disuse* of refactoring tools: if a refactoring is detected both in the version control history and in the IDE, then the tool was used, whilst if a refactoring is detected in the version control history and *not* in the IDE, then the tool was not used. Occurrences of the latter can be interpreted as a missed opportunity for the developer to use the tool.

Unfortunately, even when these mining-based studies focus on tool-supported refactoring operations they rarely involve the developers who performed the study or the experiences they had. Only a few studies have combined repository mining with information from developers' IDEs make for interesting insights about potential and actual tool use [86, 87, 91, 126]. However, these studies largely do not involve developers. In contrast, Silva et al. contacted developers who made the commits to ask them about the changes they made [112].

2.5 Studies of how Developers Refactor

Researchers have employed multiple methods to learn about how developers apply refactorings and how they use and experience refactoring tools. These studies show that a number of refactoring tools are in *disuse*, such as `move-method`, `inline-method`, and `remove-parameter`, and that developers are often perform refactorings in the context of

funcational changes.

Murphy-Hill, Parnin, and Black [87, 86] aggregated and analyzed data from several datasets: the Eclipse Usage Collector dataset of anonymized Eclipse IDE interactions, the version control history of the open-source projects Eclipse and JUnit, the refactoring histories from four developers who maintain Eclipse's refactoring tools, and volunteer IDE event data from 41 developers that was collected with the Mylyn monitor and compared with the version control history of the source code these developers were working on. They used this information to characterize how often different refactoring operations were invoked, completed, and integrated into the code repository. Through their analysis, they found that developers' refactoring behavior differs from the process that Mens and Tourwé described (so-called *root-refactoring*). Instead, most of the refactorings that they detected were small and interleaved with other code changes (*floss-refactoring*). Moreover, 90 % of the refactorings that they detected were performed manually, such as `move-member`[5], `inline-method`, and `remove-parameter`. That makes these refactorings interesting to study in order to learn *why* developers avoid using the tools to apply them, and what they do instead. Other researchers have also observed disuse of the same refactoring operations [126, 67, 112].

Vakilian et al. investigated refactoring tool use by analyzing data from IDEs that captured developer interactions. They investigated how developers invoked, configured, and applied refactoring tools and found further evidence for disuse [126]. Based on their findings, the authors conclude that refactoring tools should automate small, predictable code changes that developers can manually compose into complex changes [127].

Negara et al. also combined the analysis of version control histories and tool interactions to study how 23 developers applied refactoring during their daily tasks [91]. They found that over half of all refactorings that they detected to be performed manually and that 30% of them never reached the version control system and corroborated many of the findings from the studies by Vakilian et al. [126] and Murphy-Hill et al. [87], such as that developers prefer to automate small changes and that many refactorings that are applied in an IDE never reach the version control system. Their findings further indicate the need for refactoring tool support with the appropriate usability support and trade-offs for use in software changes.

Silva et al. used an interesting methodological approach to investigate why developers use refactorings [112]. They detected refactoring commits to a number of GitHub open-source projects and immediately query contributors for the motivations behind their

[5]This refactoring includes both moving member methods and fields but excludes moving static methods and fields.

refactoring operations. They found that refactoring activity is mainly driven by changes in requirements or *functional* changes. They also ask contributors if the refactoring was performed manually or using a tool. Over half of the respondents report doing it manually, including for `move-method`, `inline-method`. This shows that refactoring tool usability must be studied in the context of functional changes.

Kim et al. were also interested in how software developers experience refactorings [67]. They conducted a survey of Microsoft employees and more in-depth interviews with a specific refactoring team and an analysis of that team's version history. Their participants self-reported doing 85% of refactorings manually. In their study, the meaning of the term 'refactoring' was broader and refactoring operations like `remove-parameter` was used interchangeably with large quality-improving program changes that occurred over several years. The latter category of refactorings is less amenable to direct tool support and varies substantially from the smaller operations for which automated support appears in development environments. This definition of refactoring is nonetheless common in industry and can lead to ambiguity in study results when the type of refactoring is not defined. In this study, participants also self-reported performing `remove-parameter` and `inline-method` mostly manually. The survey does not consider use of `move-method`.

Liu et al. were interested in what motivates developers to apply refactorings and focused their study on the `extract-method` refactoring. Through an analysis of version histories of open-source projects and interviews of developers, they found reuse to be the main motivation for performing this specific refactoring [74], but did not consider how the refactoring was applied or how the developers experienced it. Paixão et al. also investigated why developers apply refactorings and mined code changes from open-source communities, finding that refactorings are mainly driven by the intent to introduce or enhance features [95]. In both cases, developers perform refactorings with a larger goal in mind, rather than in order to remove smells.

Three other studies that summarize perspectives on refactoring from the industry are conducted by Sharma et al. [108], and Leppänen et al. [72], and Tempero, Gorschek, and Angelis [118]. Through interviews and surveys with industry practitioners, the authors investigate hinderances to refactoring adoption and how practitioners experience refactoring [72, 118, 108]. All the studies find that refactoring tools are not adequately supported or adopted by practitioners. Sharma et al. concludes that *"academia, industry, and tool vendors must collaborate to create a new generation of more useable, effective refactoring tools."* [108]

All of these studies found refactoring to be a frequent practice in the evolution of software. By mining repositories, extracting data about tool use from IDEs, and surveying and

interviewing practitioners, these studies show that refactoring frequently occurs, and that refactoring tools are often not used. Instead, developers perform most refactorings manually. These findings indicate that there is a high potential for automation which can save developers from the time-consuming and error-prone manual code changes.

2.6 Factors that Impact the Use and Disuse of Refactoring Tools

Numerous studies have found that refactorings frequently occur as part of software changes. Moreover, they also found that the refactoring tools that, in theory, can automate these code changes are underused by developers. It is interesting to understand *why* this underuse occurs and which factors contributes to it.

Murphy-Hill et al. investigated why developers that they had studied avoided using refactoring tools by performing followup surveys of 5 of these participants [86, 87]. Vakilian et al. took a similar approach and performed followup interviews with 9 participants to learn why they avoided using refactoring tools [126]. Silva et al. also found several factors that impacted whether the authors of repository commits that they surveyed had used refactoring tools or not [112]. Kim et al. speculate about factors based on qualitative analysis of their participants' comments [67]. Negara et al. did a quantitive investigation of automated refactoring versus manual refactorings and speculated about factors that could cause their findings [91]. In addition, Leppänen et al. [72], Sharma et al [108], and Tempero et al. [118] report on some factors that prohibit refactoring tool use based on their studies.

Table 2.1 summarizes the factors that were found to impact refactoring tool use or the disuse of these tools in these studies. As shown in the table, many of the same factors occur in multiple studies. Therefore, I organize this section around factors rather than studies.

Awareness is a factor that occur across multiple studies. Murphy-Hill et al. refer both to the developer's awareness of the refactoring tool's existence and to their awareness of performing refactorings [87]. For example, a developer may avoid using a tool to perform `inline-method` because they are unaware such a tool exists. Negara et al. also speculate that novices are less *familiar* with refactoring tools than experienced developers and therefore use them less [91]. On the other hand, a developer may be aware of this tool's existence but completes the corresponding code changes manually before they become aware that they are performing an `inline-method` refactoring that

Table 2.1: This table contains an overview of the studies that have investigated developers' adoption of refactoring tools. For each study, the middle column lists the percentage of refactorings which are found to be manual in that study and the rightmost column lists the factors that the authors report prohibit the use of refactoring tools. If a study did not report this information, the column contains a dash.

Source	Manual	Factors that prohibit tool use
Murphy-Hill et al. [86, 87]	90 %	Awareness, Trust, Opportunity, Touch Points, Disrupted Flow
Murphy-Hill et al.[84]	-	Inability to select code, Incomprehensible error messages
Vakilian et al. [126]	> 50%	Awareness, Trust, Predictability, Naming, Need, Configuration
Silva et al. [112]	> 50%	Awareness, Trust, Complexity, IDE Support, Familiarity
Kim et al. [67]	85%	IDE support is too low-level, Lack of support for the types of refactoring developers want to make
Negara et al. [91]	> 50%	Familiarity, Speed
Leppänen et al. [72]	-	Trust
Sharma et al. [108]	-	Lack of exposure, Finding tools, Differential code base
Tempero et al. [118]	-	Lack of information identifying consequences, Lack of certainty regarding risk, Difficulty translating a refactoring goal into refactoring operations

could be automated. Vakilian et al. and Silva et al. also suggest that lack of awareness or familiarity with reafctoring tools prohibit tool use [126, 112]. In addition, Silva et al. also describe participants who lacked *familiarity* with the refactoring capabilities of their IDEs [112]. In the same vein, Sharma et al. refer to *lack of exposure* and *finding tools* as factors that prohibit refactoring tool use [108]. Tempero et al. refer to "difficulty translating a refactoring goal into refactoring operations" which might be hindered by the developer's awareness or familiarity with the tools that are available to them [118]. Lack of awareness and lack of familiarity can both be mitigated by education and by developing tools that automatically recognize that a developer is performing a manual refactoring. Examples of such tools are BeneFactor [50] and WitchDoctor [45].

Trust is described as a function of usability and reliability by Vakilian et al. [126] who report that *"[w]e found usability to be a more important factor than reliability on users' trust in a mature refactoring tool like that of Eclipse."* [126, 238] Interestingly, it is unclear how to address trust nor what aspects of a tool impacts developers' trust. Murphy-Hill et al. describe trust as whether developers have faith in the refactoring tool; lack of trust indicates a fear of introducing errors or unintended side effects [87]. They propose to present the user with inspection checklists [86] to improve trust. This is similar to the Preview View that Campbell and Miller [28] propose to address the developers' uncertainty of how a tool will change their program or a fear that it will "mangle their code" [28, p. 1]. Campbell and Miller refer to "predictability" rather than trust, despite describing similar fears to the ones Murphy-Hill et al. propose as related to trust. Leppänen et al. mention that their participants "have little trust in automation" but do not elaborate on what that mean or why [72, p. 68]. Brant and Steimann [24] discussed whether tool correctness is necessary for developers to trust and use a tool and do not reach a conclusion. Tempero et al. list "lack of certainty regarding risk" as a factor that prohibits tool use [118].

Predictability indicates the developer's ability to *guess* the result of applying the tool [126], i.e. how it will change their code (similar to "trust" in [28]). Tempero et al. also refer to "[l]ack of (..,) information identifying consequences" [118, p. 60] as a factor that prohibits tool use. Oliveira et al. investigated the predictability of refactoring tools by conducting a survey where developers choose which output they expect from refactoring operations (e.g., `Rename Field` and `Pull Up Field`) and found that respondents rarely agree about what to expect nor do their expectations align with the behavior of their IDEs [92]. Developers' inability to predict a refactoring's outcome might also impact their trust in the tool. Vakilian et al. [126] point out that predictability is impacted by complexity: a more complex refactoring is harder to guess the outcome of and, if it impacts many locations, is harder to verify. Simple, local refactorings, in contrast, are easier to predict and verify. They suggest. the ability to preview the refactoring as

a way to improve predictability. However, Murphy-Hill et al. [86] speculate that developers find preview windows insufficient to understand the refactoring's *touch points*, i.e., the set of places in the code that the refactoring would effect. They also describe that preview windows *disrupt* developers' *flow* when requiring context-shifts into a separate window.

Configuration options afford users of refactoring tools control over the transformations that the tool will perform on their code. For example, the `change-signature` in IntelliJ and Eclipse can be invoked on a method and then be configured to remove or add parameters, change the method's return type, or even the parameter names (Chapter 2.1). Vakilian et al. report that configuration windows introduce additional cognitive overhead and disrupt developers' flow [126]. Murphy-Hill report that developers rarely configure refactorings and instead prefer to tweak the resulting code [87].

Need indicates that the developers are performing code changes that a tool can automate [126]. Some tools are not used because the operations they automate are rarely appropriate. This echoes *opportunity* in [86]: the situation in a developer know that the refactoring tool exist, but they do not recognize opportunities to apply it. Similarly, in Silva et al.'s work, if a refactoring has too low of a complexity, participants indicate that there is no need to use a tool [112]. These contradicting observations indicate that some tradeoff must be made in the design of tools.

Naming indicates that hard-to-understand names of refactoring tools act as a barrier for developers in finding and trying tools [126]. This was also briefly mentioned in [86] and addressed in [88] with gesture-based menus. Other works that has approached this problem is Lee, Chen and Johnson with Drag-and-Drop invocation of refactorings [71] and several works that can recognize a manual refactoring and invoke refactorings to complete it [45, 50, 100]. Some of these efforts aimed at addressing this factor overlap with the efforts related to *awareness* and *familiarity*, such as the detection and completion of manual refactorings. Toolmakers can also address this factor by simplifying refactoring *names*. One example of this is the decision to keep only high-level names like `move` and `change-signature` in the refactoring menu of IntelliJ and Eclipse, and then use techniques like Configuration View and automated detection of program elements to determine the exact refactoring (Chapter 2.1). However, this can cause other problems, such as the effort required to configure refactorings and an inability to understand how to invoke the right refactoring.

Complexity is both used to explain avoidance of tools due to lack of *need* to automate overly simple refactorings [112] and the difficulty of using overly complex refactorings [126]. Multiple studies have found that developers more often automate simple

refactorings and perform complex refactorings manually, even for refactorings that are supported by refactoring tools [91, 126, 87].

IDE support indicates that the IDE does not support a refactoring operation [126]. Lack of *awareness* or *naming* may contribute to developers falsely claiming this as a reason. However, many of the refactoring activities that developers perform may not be supported by IDEs. This can be mitigated by implementing support for more refactoring operations and support for a wider variety of programs. However, this might negatively impact the *complexity* of tools, the *predictability* and *trust* of tools, and developers may still not have *awareness* of these additional operations.

Differential codebase is listed as a factor by Sharma et al. [108]. They explain this as follows: *"We want to focus on assessing and improving only the changed or newly added parts in the source code, and refactoring tools can't run only on this differential code base"* [108, p. 47]. This is an interesting problem, and one that has not beed addressed in research that proposes tool improvements.

Some factors relate to specific usability problems that arise from the tools' user interface. Murphy-Hill and Black [84] investigated usability problems with the `extract-method` refactoring tool and found that developers had problems selecting the right code segments and problems understanding error messages. They acted on this information by proposing tools that could aid in making a correct code selection and proposing more understandable error messages. Several of the specific changes that they proposed, such as more understandable error messages for `extract-method`, were incorporated into the Eclipse refactoring tool.

While previous research has identified multiple factors that prohibit the use of refactoring tools, acting on these factors to improve the adoption of tools is not a trivial matter. Some factors can be isolated and addressed. While some factors can be isolated and addressed individually, others are tangled, difficult to define and address, and in some cases contradicting. For example, *awareness* can be tackled through education and support of invocations through manual changes and *IDE support* can be improved by increasing the number of refactorings that an IDE supports. Meanwhile, *trust* and *predictability* have been persistent topics in the research of refactoring tool usability, yet there is no agreed-upon definition of exactly what makes refactoring tools trustworthy or predictable.

Addressing such tangled and ill-defined factors are complicated by their contradictions. For example, a toolmaker might be able to improve *IDE support* and make a tool more flexible by increasing the number of situations a tool can work in and even accept some erroneous invocations. However, in doing so, the tool's *trustworthiness* might be reduced

because it will occasionally introduce errors. As another example, a toolmaker might increase the tool's *trustworthiness* and *predictability* by limiting the accepted invocations to programs that can be easily analyzed and transformed. However, that might reduce the *IDE support* because there may be many situations in which the tool can not be applied. Refactoring tools cannot always both meet developers desires and be fully correct [24]. As a final example, *naming* and *awareness* can be addressed by simplifying the invocation of refactorings through automated detection and simplified names. That, however, might result in less *predictable* tools as the developer might accidentally invoke the wrong refactorings. Such design tradeoffs must be made by toolmakers who wish to improve the usability of refactoring tool. This illustrate the challenges in addressing factors that prohibit the use of refactoring tools.

2.7 Software Changes

As refactoring tools support developers' work on software change tasks, one might expect studies of software changes to provide insights into how refactoring tools are used and experienced during these activities. Unfortunately, studies of software change tasks tend to focus on such aspects as how software artifacts are altered through the change process or the questions developers ask as the change progresses, without investigating the role of tools as part of the change process (e.g., [111, 96]).

The need for software tools to help support change to software systems has been recognized for many years (e.g., [65]). While the need for tools has been evident, our understanding of what kinds of tools are needed, how those tools function and how developers can best match tools to tasks is still evolving.

Early software change studies focused primarily on how developers approach comprehending the software they must change. Researchers developed models to try to explain how individual developers comprehend source code (e.g., [114, 97, 129]). At the time these models were developed, tools such as textual search and compilers were already prevalent. However, these models focused on human cognition models without considering how tools interplayed with the comprehension process. Flor and Hutchins took a different approach, considering programming as a complex activity in which multiple developers work collaboratively on maintenance tasks [44]. They described a distributed cognition model, but they also described the role tools played for developers in their study in helping them enact one out of several alternative strategies to tasks in their study. Although these insights were not drawn out as major findings, their work marks a consideration of how tools and tasks interact. Our work continues this direction of

inquiry.

Drawing on the importance of program comprehension for software change tasks, researchers have honed in on specific activities that form part of a change task and the specific tools used to support those activities. For example, Starke et al. studied how programmers use search tools as part of program comprehension activities undertaken during corrective maintenance tasks [115]. Fleming et al. used *information foraging theory* to evaluate tools for debugging, refactoring and reuse tasks [43] and found that these tools can alleviate the foraging activities that developers otherwise do manually.

As research on software change progressed, there became an increasing focus on understanding developer work practices during software change tasks, considering the challenges developers faced and how to support those challenges with tools. Singer et al. report on the work practices of professional developers and use insights from the discovered practices to design a tool to support software maintenance [113]. Ko et al. used a laboratory-based experiment to study the different approaches of multiple developers to corrective and perfective maintenance tasks and used insights gained to suggest design requirements for tooling aimed at supporting software maintenance tasks [68]. Sillito et al. conducted a laboratory-based study to consider the goals developers formed when performing software change tasks and how developers used tools available to perform those goals. From these observations, they described opportunities for further tool development to better support the developers [110]. However, none of these works has focused on refactorings as part of the change tasks.

Rajlich presented a phased model of the software change process that included refactorings as part of functional changes [99]. Their work, however, related to a higher-level view of these activities and did not include descriptions of how developers interleave the use of tools in these phases or how refactoring tools can better support them. In an unpublished paper, Wilson et al. [130] considers how the change propagation tool JRipples [27] supports such activities. JRipples uses code dependencies to support *change impact analysis* [21] and *change propagation* [26] by providing the developer with a checklist of locations in the code that might affected by a change. Buckley et al. has identified refactoring tools as another means of change propagation [26], but rather than using any code dependencies, refactoring tools propagate changes according to specific rules [46]. Despite the similarities between these types of tools, they are not typically compared in studies.

Chapter 3

A Study of Refactorings During Software Change Tasks

All too often, refactoring operations and tools are studied out of context of functional software change tasks. To explore how developers experience existing refactoring tools during software change tasks, I designed and executed a study. The study comprised one task segment and one interview segment. During the task segment, I observed the participant while they attempted to solve three software change tasks that contained steps amenable to refactoring while vocalizing their thoughts. Between each task, I conducted a short interview. After all three tasks were completed, I conducted a longer interview to better understand their approaches to the tasks as well as their use of and experience with refactoring during their usual work. 17 individuals participated in this study, resulting in a published dataset of 32 hours of screen recordings and a transcript of 130k words. This chapter describes the study's motivation, design, execution, data analysis, and the resulting findings.

Part of this chapter has been published elsewhere [41]. The dataset and a replication package is available in DataverseNO [38].

3.1 Introduction

In Chapter 2, I described that multiple studies have shown that some portion of the modifications developers undertake as part of change tasks are refactorings. Common refactorings like `rename`, `extract`, and `move`[47] occur both during software changes that are purely quality-improving and—more commonly—during changes to software functionality [95, 96, 112]. The automation of refactoring operations is intended to

enable developers to modify code faster and with fewer errors. This is a seductive proposition and has led to many research and development efforts aimed at enabling and supporting a broad set of refactorings in commonly used IDEs. For example, the standard installations of mainstream IDEs for the popular object-oriented languages Java (e.g., Eclipse[1]) and C# (e.g., Visual Studio[16]) each offer their users automated support for over 40 refactoring operations.

Despite a long-standing prevalence of refactoring tools in development environments, it is widely recognized that this refactoring support is not used as often as would be expected, and in fact, these tools are *disused* [126]. To address this disuse, researchers have studied a variety of different approaches, including improving developers' awareness of the tools [108, 45, 50], improving the usability [84] and correctness [106, 22, 42, 81] of the tools, and investigating factors that affect developers' acceptance of automation [127]. These approaches generally assume that increased use of automated refactorings is largely about the tools: if developers are made aware of tools and the tools are made usable, then the tools will be used.

I take a different perspective. I believe that to be both useful and used, tools must not only automate code transformations but must fit into a developer's overall workflow. In other words, I pursue an approach that is about the humans first and the tools second. Thus, I choose to investigate the broader context of refactoring during software change tasks by considering how developers approach the tasks, how their approach enables or blocks opportunities to use a refactoring, and their use and success with refactorings, whether manual or automated. To achieve this, I carried out an observational study to investigate, in a controlled environment, how 17 experienced developers approached three software change tasks that are amenable to the use of automated refactorings. With this study, I explore three research questions:

RQ1 What strategies do developers use to approach software change tasks which include refactorings?

RQ2 How often do developers use automated support for refactoring versus proceeding manually?

RQ3 How do developers experience refactoring tools?

3.2 Study Design

Here, I describe the study method, the design constraints, and the resulting study design.

3.2.1 Method

To support the investigation of how software developers use refactoring tools in software change tasks, I performed an observational study. I recruited software developers with professional experience and asked them to solve a set of software change tasks while following a think-aloud protocol [128]. After their work on the tasks, I conducted a semi-structured interview. This format made it possible to compare how multiple developers approached the same three software change tasks and compare their interview transcripts, as well as to ground the interviews in the activities observed during their work on the tasks. For the remainder of this chapter, I refer to myself in those settings as *the experimenter* and *the interviewer*. This study was carried out in collaboration with Gail Murphy, who I refer to as the *co-experimenter*.

The tasks were performed on a scaled-down version that I created of the Apache Commons-Lang project[6] and were modeled after commits to popular open-source systems, verified to contain refactorings [123] that previous work has identified to be performed manually more often than with automated tools: `change-signature`, `inline-method`, and `move-method` (Chapter 2.5). This makes the tasks realistic and relevant to understand refactoring tool use and disuse.

The study's design was motivated by several gaps in the scientific literature that has focused on developers' use of refactoring tools and what impacted their choice to use or not use these tools. For example, one important consideration was to disambiguate the refactoring terminology to ensure that the interviewer and the participant were referring to the same types of changes (Chapter 2.4). This was achieved by grounding the interview in their work on the tasks. Both the interviewer and a participant were able to use concrete changes that they had performed as examples during the interview.

Another consideration was to recruit practitioners with significant industry experience and expose them to realistic usage scenarios for refactoring tools. Refactoring is most often motivated by functional changes [95, 74]. In fact, developers often do not know upfront that they are refactoring [86, 45, 50]. Therefore, the tasks are described in terms of code changes that impact functionality, not as refactoring tasks. At no point were the participants instructed to refactor nor were they told which refactorings could be used on the tasks. To avoid biasing them, the experimenter and interviewer took great care not to refer to refactorings unless a participant had already done so, and in that case, made sure to use the same refactoring name that they had used, even when it was incorrect. Only at the end of the interview were participants informed about tools or refactorings that they did not use and asked about why they avoided them, or whether they would have liked to use them another time.

In Chapter 2, I described several factors that impact disuse that other researchers have identified. Two such factors that developers in previous studies frequently bring up are trust and predictability. Despite their frequent appearance, these factors are, in fact, not well understood [24]. Since developers frequently use these terms to describe their experience with refactorings tools, and they are not well understood, they were used as sensitizing concepts during all interviews: if participants spoke about topics related to predictability or trust, the interviewer inquired further, even during tasks.

3.2.2 Participants

Nineteen individuals with professional development experience were recruited in a North American city using snowball sampling [52] seeded by company contacts known to the experimenter and co-experimenter. Participants filled out an online pre-survey to determine eligibility and give consent. To be a participant, an individual needed at least one year of professional (employed and paid) experience with Java (or a similar) programming language, be able to provide a definition of refactoring, and be able to describe a few refactoring operations.

Two of the nineteen admitted individuals experienced challenges during the experiment that led them to be excluded from the participant pool and analysis: one encountered technical problems with the IDE that required stopping the experiment, and one was unable to make progress on simple tasks. The results from these two participants are not included in the analysis and results presented in this dissertation: I will use "participants" to mean the seventeen remaining participants and use the notation P_x to reference the experiences of a particular numbered participant. To maintain consistency with the data package, I keep the original indexing of the remaining participants. This explains the existence of P_{18} and P_{19} despite the results pertaining to only 17 participants. Table 3.1 presents an overview of the participants.

For the seventeen participants, the mean (and average) number of years of work experience was 10, with the range of experience being from one to more than 20. Two participants did not have previous experience in Java but were accepted due to extensive experience (more than 10 years) with other object-oriented languages. Only two of the participants presented as female. Four participants were currently enrolled as students. Each participant was allowed to choose between traveling to a study location hosted by the experimenter on the campus of University of British Columbia, or to host the experimenter at their workplace. Seven participants chose the campus location, and ten chose to provide a meeting room at their workplace. All participants received a $20 gift card as a token of appreciation.

Table 3.1: Overview of all the admitted participants including the two participants that were excluded from analysis. Participants' number of years of professional experience and their gender are indicated in the second and third column, the editors that they reported familiarity with and whether they showed familiarity with Java in the fourth and fifth column, and in the final column whether they were currently enrolled as students.

P_{nr}	Exp (yrs)	Gender	Editor*	Java Exp	Student
P_1	-	-	-	-	-
P_2	20+	M	$IENVs$	✓	✗
P_3	10	M	S	✗	✗
P_4	6	M	IE	✓	✗
P_5	2	F	I	✓	✓
P_6	5	M	IE	✓	✓
P_7	8	M	IE	✓	✗
P_8	20+	M	$IENVsXAs$	✓	✗
P_9	10	M	IE	✓	✗
P_{10}	10	M	I	✓	✗
P_{11}	16	M	$VsXc$	✗	✗
P_{12}	11	M	IVs	✓	✗
P_{13}	1	M	I	✓	✓
P_{14}	20	M	I	✓	✗
P_{15}	2	F	$ENAs$	✓	✗
P_{16}	10	M	$IEVs$	✓	✗
P_{17}	-	-	-	-	-
P_{18}	2	M	IVs	✓	✓
P_{19}	12	M	E	✓	✗

*I=IntelliJ, E=Eclipse, N=Netbeans, Vs=Visual Studio, As=Android Studio, X=Xamarin, Xc=XCode, S=Sublime

The experimental session was conducted in person with one participant at a time. Each participant was provided with the same laptop, system, and tasks. I acted as the experimenter and was present in the room during the experiment. The experimenter administered the consent form, the tasks, answered questions, gave prompts, and took notes during the tasks. The experimenter was positioned such that she could see the screen and could prompt the participant to keep vocalizing if the participant stopped describing what they were doing. In addition, the experimenter made notes of events that occurred during the tasks that could be investigated further in the interview segments.

After obtaining consent, an experimental session began with an explanation of the experiment setup with an introduction to the scope of the tasks, the participant's role, and the experimenter's role. The participant was told that they would be asked to undertake three provided change tasks in succession on a codebase loaded into a development environment. They were also told that there was no set time per task but that the experimenter would indicate if it was time to move to the next task to keep the overall session within two hours and make time for an interview segment at the end. All experimental sessions were completed within two hours.

The tasks were given in the same order to all the participants, with the task description being revealed to the participant when that task began. For each task, the participant was given a task description on paper and asked to describe their plan for completing the task. Knowing the upfront plan enabled the experimenter to detect deviations from the plan and to prompt for events or experiences that triggered changes to the plan. The participant was told that they would not be bound to their plan.

The participant was asked to think-aloud [128] as they worked and were prompted if they lapsed in voicing their thoughts and actions. The experimenter answered questions asked by the participant, such as helping to orient them in the development environment. Answers related to questions about the tasks themselves were limited to contextual information from the original commits (e.g., motivation for the task) and the scope of the tasks (e.g., questions related to client code).

For each session, the participant's screen and audio were recorded and the state of their code at the end of each task was saved. At the end of a task, the experimenter asked:

- Which source code changes did you make in order to solve this task?
- Do you know of any tools that could have automated any of the changes you made?
- Are there any changes you are unsure if you got right?

At this point, the experimenter also asked for clarifications on the participant's actions

and strategies. For example, participants might also be asked to explain specific things they had done or why they changed their plans. This was done between tasks while the task was fresh in their minds. If they expressed different strategies throughout the task, they may be asked how they would have solved it if they were to do it again. To avoid biasing the participant, the experimenter took care to not use refactoring names until after a participant did and subsequently used the name that the participant chose in conversation.

After all three tasks were over, the experimenter immediately conducted a longer interview with that participant. The interview included questions about their past experience with refactoring tools and what impacted their choices to use or not to use these tools during the tasks.

3.2.3 Experimental System and Tasks

The change tasks used in the study are based on actual changes to open-source systems. I sourced these change tasks from commits to open-source systems. I looked for commits that contained refactorings that had been identified by earlier research as more often manually performed than with a tool. Examples of such refactorings are `move-method`, `inline-method`, and `remove-parameter` (Chapter 2.5). In addition, the code changes needed to be understandable and replicable by experienced developers in a reasonable time. I investigated the dataset from Silva et al. [112] and used a state-of-the-art tool, RefactoringMiner [123], to mine several additional repositories, including Apache Commons-Lang [6].

The resulting three change tasks were based on the repositories of Apache Commons-Lang (Task 1-2 [11, 12]) and Quasar (Task-3 [14]). I ensured that the descriptions of tasks did not instruct developers to perform specific refactorings, nor did the instructions include the word refactoring. Indeed, out of the three tasks, only the first task is a *pure* refactoring, i.e., a code change that only alters structure and not functionality. This task involves moving methods around, whilst the other two tasks alter APIs and thus require corresponding changes to tests. Previous studies have identified such API-level changes as containing refactorings [35].

The study underwent iterative pilot sessions to refine its design. The pilot participants were not counted among the study participants. A total of 6 pilot sessions were performed before arriving at a satisfactory design. The initial pilot sessions were performed on the systems in which the commits were detected. Those pilot participants solved the first two tasks in Apache Commons-Lang and the third task in Quasar.

These sessions revealed that it was challenging for participants to switch context between multiple software projects and to solve tasks in source code about an unfamiliar domain and with frameworks or coding style that were unfamiliar to them. All of the pilot participants who tried performing the change in the Quasar codebase ran into problems because they were unable to comprehend the code. In order to understand its functionality, it was necessary to comprehend its domain, which was unfamiliar to these individuals. To act on this feedback, I re-created a part of Apache Commons-Lang as a stand-alone system and mapped the three commits onto this system.

The Apache Commons-Lang project provides common helper methods such as string manipulation and basic numerical methods. I chose this system because of its relative simplicity: its target domain should be familiar to most Java programmers; it is relatively simple to scale it down in size and complexity (e.g., removing external dependencies) so that an experienced developer can become familiar with the code in a short time; it is large enough that a developer will benefit from using tools to change the code; and finally, the system follows common programming patterns such as JUnit tests and Maven standard layout for files and consists mainly of utility classes with static methods. The two commits that came from Apache Commons-Lang were easy to map to this system. I adjusted the third task so that it fit into the new system and verified that the task contained the same refactorings as in the original commit with the RefactoringMiner tool.

The final experimental system was self-contained yet large enough to simulate a realistic project, comprising 78K lines of Java, version 1.8, and 1335 JUnit tests. The system is built using Maven [7], and version-controlled through git. Participants worked on a provided Macbook Air, 13", 2017, and were asked to use IntelliJ CE IDEA 2017.3. In order to solve all three tasks, participants needed to change (e.g., move, alter or delete) 45 methods distributed across 5 files. The final pilot sessions showed that in the new system it was feasible for an experienced developer to complete the three tasks in the allotted time (1.5 hours). The three tasks and the experimental system are all available in the online dataset [38].

Task-1: Organize Test Methods

In this task, a participant is asked to reorganize 12 listed test methods from two large existing test classes into a new class, such that similar testing functionality is gathered together. This task replicates part of an Apache Commons-Lang commit[11] that reorganized their test methods.

To complete this task, a participant needed to create a new class into which the 12 listed methods were to be placed. For the code to compile, there was also a need to include the required imports and resolve a reference in the moved methods to a local constant defined in the original test class. The constant could be resolved by duplicating it in the new test class, by increasing visibility of the originally defined constant, or replacing references to it with its value.

Work on this task could benefit from the help of such automated refactorings as `move-method` and `extract-class`. The mining tool shows this task as `move-method` and `extract-class`.

Task-2: Removing Redundant Methods

In this task, a participant is asked to remove two methods, that are negated versions of two other methods in the API. The two other methods depend on the negated versions. Listing 3.1 shows one of the original method pairs, and Listing 3.2 shows part of the solution for that pair. Each of the four methods has a single JUnit test. This task replicates a commit [12] to Apache Commons-Lang that is discussed in this pull-request [13].

The minimal changes required to solve this task are to remove the two designated methods and their test methods and then to restore behavior in the project. Restoring behavior requires either inlining the methods before removal or reimplementing the callers.

Work on this task could benefit from the `inline-method` refactoring. RefactoringMiner shows this task as `inline-method`.

Task-3: Overloaded Method Reduction

In this task, a participant is asked to remove particular functionality from a designated class. The class consists of eight method pairs that offer functionality related to reading from and writing to objects of some datatype. In each method pair, one method implements some standard behavior, and the other method takes an extra flag argument that lets callers toggle some special behavior (e.g., by passing in `true`). The first method is a wrapper to the second method, calling the second method with the extra flag argument set to a particular value (e.g., `false`). Listing 3.3 provides one example of such a method pair. All methods in this class have multiple tests each. Both the standard behavior and the special behavior are tested.

A participant is guided to remove the parameter such that clients can no longer detect the

Listing 3.1: The method isAllEmpty depends on isAnyNotEmpty. Task-2 involves removing isAnyNotEmpty.

```
60 public static boolean isAllEmpty(final CharSequence... css) {
61    return !isAnyNotEmpty(css);
62 }
63 public static boolean isAnyNotEmpty(final CharSequence... css){
64    if (ArrayUtils.isEmpty(css)) {
65       return false;
66    }
67    for (final CharSequence cs : css) {
68       if (isNotEmpty(cs)) {
69          return true;
70       }
71    }
72    return false;
73 }
```

Listing 3.2: The resulting code after removing isAnyNotEmpty and moving its implementation to isAllEmpty.

```
60 public static boolean isAllEmpty(final CharSequence... css) {
61    if (ArrayUtils.isEmpty(css)) {
62       return true;
63    }
64    for (final CharSequence cs : css) {
65       if (isNotEmpty(cs)) {
66          return false;
67       }
68    }
69    return true;
70 }
```

functionality (e.g., changing the visibility of methods is not sufficient). Each method that has the flag also has an overloaded version without the flag in the original code that acts as a wrapper method of the default functionality. This task replicates part of a commit to Quasar that was collected and analyzed by Silva et al. [112]. I mapped the refactorings in this commit onto a class in our experimental system and verified that the task contained an equivalent refactoring both manually and using RefactoringMiner [123].

Listing 3.3: One of the eight method pairs in Task-3. The first method provides the standard behavior and the second method offers some special behavior (i.e., forcing access) if the flag parameter forceAccess is called with a `true` value.

```
112  public static void writeField(final Field field, final Object
         target, final Object value) throws IllegalAccessException {
113    writeField(field, target, value, false);
114  }
115
116  public static void writeField(final Field field, final Object
         target, final Object value, final boolean forceAccess)
         throws IllegalAccessException {
117    Validate.isTrue(field != null, "The field must not be
           null");
118    if (forceAccess && !field.isAccessible()) {
119      field.setAccessible(true);
120    } else {
121      MemberUtils.setAccessibleWorkaround(field);
122    }
123    field.set(target, value);
124  }
```

To complete this task, all eight method declarations should have the boolean parameter removed, and all calls should be updated. The method pairs should be reduced into one method, such that only the wrapper method signature exists and contains all of the logic. All logic that relies on the parameter should perform as if the parameter was false. For example, an if-branch that executes if and only if the parameter is true can be removed altogether. All tests associated with the boolean parameter are duplicated and can be removed. The many changes needed to be performed are intended to make automated support more attractive to a developer.

Work on this task can benefit from the `safe-delete`, `inline-method`, or the `remove-parameter` refactoring, which are available through the `change-signature` menu option in IntelliJ. RefactoringMiner shows this task as `inline-method`.

3.3 Data and Analysis

The study data comprised approximately 32 hours of study sessions. I transcribed the audio portion of the video captured from the 17 experimental sessions and created one transcript for each session from the recorded screen capture and audio capture information. In these transcripts, I marked the sections related to interview questions and answers. In total, the 17 transcripts comprise 134,947 words and can be accessed in the replication package along with the screen recordings, participants' solution source code, and a report from the mining tool on refactorings that can be detected in each of their solutions. This data was analyzed in several steps.

I performed the initial labeling of the transcripts for easier retrieval of points of interest by marking sections that pertain to events in the videos or comments made by participants. The labels are shown in Table 3.2 and identify actions or events that were significant to the participants' individual workflows, such as their stated *plans* for how to solve the tasks, which refactoring tools they *named* or *invoked*, other tools they invoked, and actions they took to *validate* that their changes were correct before continuing their work. While labels pertaining to plans or sentiments could be identified solely by the transcript and labels pertaining to invocations solely by the video, the remaining labels needed to be determined by reviewing both audio and video. The labeling was discussed with, and reviewed by the co-experimenter.

We then created summaries of the plans and actions that each participant used to approach each task. In order to develop a schema that captured a participant's approach to each task, the summary schema included the following information: quotes that indicate their plans or *intent* (if any were made); a natural-language summary of their *approach*, written by the experimenter; which refactoring tools they *mentioned*, *tried*, and *used* with and without prompts; other *tools* that they used to solve the task; how they guided their changes (e.g., emphasis on navigating to and fixing compiler errors, avoiding errors, top-down in the file etc.); and how they verified their changes. The types of tools and information sources that the participants could use are listed in Table 3.3. The set of possible refactoring tools is taken from the IDE that was used [3]. If a participant mentions or invokes a refactoring tool, then they are considered *aware* of the tool; if they invoke the tool, then they are considered to have *tried* it; and in order to *use* a refactoring tool, the participant needs to both apply it and continue working with the resulting code.

I created the summaries by applying the schema to all participants' workflows on each task. The schema was developed by the experimenter and co-experimenter by reviewing several video recordings and discussing participant workflows. I reviewed the video and

Table 3.2: The labels that were used for initial labelling of transcripts.

Code	Description
Plan (Keywords)	The participant expresses actions they intend to take to solve the task. Keywords include short, descriptive statements such as break-fix or systematic.
Change Validation (Method)	The participant takes an action to validate their change. Method is a short, descriptive statement of the method they employed such as static (e.g. compiling, syntax errors, warnings), dynamic (e.g. running tests), compare (e.g. using git diff of comparing to commented out code).
Named (Refactoring)	The participant names a refactoring tool. Refactoring tools include the common refactoring tools: Rename, Move, Inline, etc.
Invoked (Tool)	The participant invokes a tool. Tools include the common refactoring tools: Rename, Move, Inline, etc. as well as non-refactoring tools like Find Usages, etc.
Restart	The participant restarts the task.
Mistake	The participant makes a coding mistake such as changing the wrong code location.
Error (Not) Understand (Refactoring /Other)	The participant encounters an error that they do (not) understand, either arising from a refactoring tool, the compiler, or tests. Mark errors that are significant to their work, not all encountered errors. Mark source of error (e.g. Move, compiler) where non-obvious in transcript.
Trust	The participant expresses a sentiment related to tool use or disuse and trust or lack of trust.
Predictable	The participant expresses a sentiment related to tool use or disuse and predictability or lack of predictability.
Refactoring Insight	The participant makes an insightful statement regarding tool use or disuse that may be useful to retrieve later.

Table 3.3: This table lists the tools that developers were observed to use in their workflows. The Tool column shows a short name for the tool and the Description column gives a brief description of the tool and its purpose.

Tool	Description
`VCS-diffs`	Participants used diff tools like git diff to understand how their code had changed and verify that a change was correct.
`structural-navigation`	Participants used structural navigation like Find Usages and Go To Declaration to understand the code and locate areas that should be changed.
`refactoring-tool`	Participants used refactoring tools to change the code and to organize their changes.
`test-suite`	Participants used the test suite to understand the code, to understand the impact of their changes, and to verify that a change is correct.
`debug-tool`	Participants used the debugger to comprehend what the code did.
`text-search`	Participants used textual search like Control+F or `grep` to search for references, callers, and the declarations of methods and parameters.
`compiler-output`	Participants used compiler output like errors and warnings to navigate the code, to understand the impact of their changes, and to verify that a change is correct.
`copy/cut/paste`	Participants used copy, cut, and paste, to edit the code.

Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.

audio recording of each participant to create a summary for that participant. Figure 3.1 shows the schema completed for a participant working on the second task. In this summary, the participant's *intent* is represented by a quote that summarizes their intended plan, while the *Approach Summary* contains the experimenter's description of the actual actions the participants took, problems they encountered (e.g., tool error), and how they organized their changes (e.g., "Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step.."). Then, I list any refactoring tools that the participant *mentions*, *tries* to use, or *uses*. In addition, the summary contains records of other tools that they relied on (such as compiler errors, navigational tools and diff-tools). It contains a brief description of how they guided their changes and how they verified their changes. To verify that the summaries appropriately capture a participant's workflow, the co-experimenter reviewed several summaries, comparing them to the video recordings. The two experimenters also discussed any difficult to interpret actions by developers. This process resulted in 51 summaries.

Once the set of 51 summaries was created, we (the two experimenters) compared them and looked for patterns in the following information: the *intent* that participants ex-

pressed and the actions they used to *change* the code, to *guide* their changes, to *verify* their changes and to *recover* from unwanted errors. Based on the patterns between summaries, we identified different *strategies* that participants employed in each task.

3.4 Results

Overall, participants were able to make good progress on the tasks: all participants made some progress on all three tasks. Furthermore, all participants completed Task-1, sixteen (94%) completed Task-2, and twelve (70%) completed Task-3. Twelve (70%) participants completed all tasks. The progress made by participants on the tasks provided substantial opportunities to observe the workflow and tool use during each task. I describe the results in terms of the research questions posed in Chapter 3.1 and use the notation P_x in to reference to the experiences of a particular numbered participant.

I consider the three research questions outlined in Chapter 3.1 in turn.

3.4.1 RQ1. *What strategies do developers use to approach software change tasks which include refactorings?*

The participants took a variety of approaches to the three software change tasks but there were commonalities in the workflows that they employed. The analysis identified three distinctly different strategies for interacting with the code:

- **Local**, in which participants guided their changes by *local* and immediate information such as feedback from the compiler.
- **Structure**, in which participants guided their changes by code *structure* such as callers, conditional branches and references.
- **Execute**, in which participants guided their changes by *executing* the test suite.

Figure 3.2 shows how the strategies were distributed across each participant's approach to a task. Each of the strategies was successfully employed by one or more participants on each task, indicating that a task did not dictate which strategy to use. Instead, the choice of strategy was dependent on the combination of the individual and the task, with some participants changing strategies between tasks. Figure 3.3 further shows that there was no one dominant strategy used for a task.

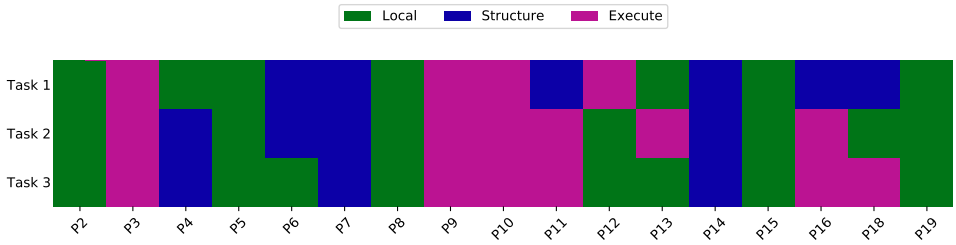


Figure 3.2: The strategies were distributed both across participants (x-axis) and tasks (y-axis). In total, 22 workflows were identified as **Local**, 13 were identified as **Structure**, and 16 were identified as **Execute**. As can be seen in P_4, P_{16}, and P_{18}, individual participants sometimes changed strategies between tasks.

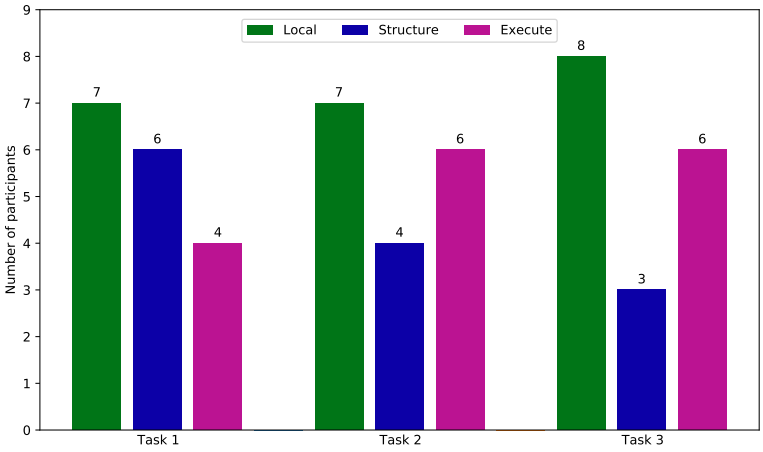


Figure 3.3: The workflows are first grouped by task, then by strategy, so that the 17 workflows from Task-1 are represented by the three leftmost bars, the 17 workflows from Task-2 by the three middle bars, and the 17 from Task-3 in the three rightmost bars. Within each group, the three different-colored bars indicate how the strategies were distributed across participant's workflows for that task. For example, in Task-1, there were 7 participants who employed a workflow that matched the **Local** strategy, 6 participants with a workflow matching the **Structure** strategy and 4 matching the **Execute** strategy.

Despite the variety of strategies, the participants would often produce the same source code solution in the end. For example, a participant who was employing the **Execute** strategy might first isolate the functional changes that need to be performed in order to achieve the task goal, and then update the code to match this new behavior. Meanwhile, a participant who was utilizing a **Local** strategy might immediately alter the code that seems to implement the behavior that is required to change, then—almost accidentally—propagate the change to the tests by iteratively fixing compiler errors. A participant using a **Structure** approach will likely make the same code changes but may make them by studying the program's structure and propagating local changes through these structures rather than isolating either local changes or changes oriented to the program execution.

Local

Participants who used a **Local** strategy had workflows that were characterized by interacting predominately with information that was immediately available to them (i.e., *local* information), and changing elements that were present in their editor. Their workflows included actions that intentionally provoke feedback from the IDE, such as deleting or commenting out code elements that have references elsewhere in the code, thereby introducing compiler errors or "red lines". When stating their intent, participants made comments such as P_4 on Task-1: *"copy-paste them all from one place to another and see what breaks"* and P_2 on Task-3: *"So what I'm going to do now is rely on my friendly compiler to do this. So now things are going to break"*. Their workflows were oriented around introducing, navigating between, and fixing this feedback. As a result, the source code frequently underwent long periods in non-compiling states and the impact that source code changes had on their tests in Task-2 and Task-3 was often not known until at the very end.

For example, in Task-2, participants who used a **Local** strategy in this task (P_2, P_5, P_8, P_{12}, P_{15}, P_{18}, P_{19}) began by deleting the method that contained the implementation, which leads to compiler errors in the caller. In order to "fix" the compiler error, one needs to either obtain the previously deleted code or reimplement the method from scratch. Only three of the seven participants obtained the previous implementation (through git, undo, or comments), while the remaining four reimplemented the method from scratch. All of these four spent significant time on resolving bugs or errors that appeared in their reimplementation. In comparison, only one other participant with a non-**Local** strategy (P_{16}) had a similar problem.

Participants who relied on the **Local** strategy were able to guide their navigation and

changes by utilizing compiler errors and warnings. Yet, when they encountered errors that they did not understand, they had little choice but to "backtrack" and redo the task, hoping that their guidance would work better the next time. For instance, P_2 had to restart Task-3 after encountering a compiler error that they did not understand. This error originated from a binding change of an overloaded method that they had not noticed. They said *"Just confused by this error. Wrong first argument type. That's weird. (...) So I don't understand the error."* Since the participant did not understand the error or how to solve it, their solution was to restart and redo the previous steps in another order. They did not run into the same error again. On the same task, P_6 decided where to work based on how many compiler errors arose and described that they would monitor the number of red lines that a change induced and, if above a certain threshold, they would undo that change and look for any other place to work.

Structure

Participants who used a **Structure** strategy were orienting their workflows around source code structure, such as considering references or callers to an element. When starting a task, they typically explored code from one or more starting points using structural queries before altering the program. The understanding a participant gained about the code structure through their upfront explorations was used to formulate plans for their changes according to the impact it would have on other places in the code.

In Task-2, four participants (P_4, P_6, P_7, and P_{14}) used this strategy. All of them formed plans to inline the method that was to be removed to its caller and executed this plan either manually or using tools. For example, P_7 stated in his upfront plan that *"Any invocations would need to be dealt with. So what I'd like to do, is look in StringUtils, look where these are used (..) and then depending on the implementation of the anynot [method] I may just be able to inline them"*. When they changed code, they intentionally propagate changes across code structures by means of navigational tools or refactoring tools. Where other participants might organize their changes so that they can act on local information (**Local**) or around the execution of tests (**Execute**), these participants made efforts at organizing changes along structural edges even when it required more navigation or "leaving behind" problems in the code.

Both users of the **Local** and **Structure** strategy delayed test changes to the end of Task-2 and Task-3. This may not be intentional—participants did not express such an intent—but may be simply due to the compiler errors they needed to navigate during the tasks. This is in stark contrast to participants who used the **Execute** strategy who relied heavily on the tests' behavior throughout the entire task.

Execute

Participants who were using an **Execute** strategy had workflows that were oriented towards executing the tests, and thus, keeping the code compiling so that the tests could be run throughout the task. At the beginning of tasks, they explored the codebase using structural navigational tools, similarly to someone using a **Structure** strategy. However, rather than using these tools to propagate changes in the code, they did so in order to locate the relevant tests and exploring the call graph.

These participants searched for a start location that allowed them to isolate functional changes from syntactic changes. For example, for Task-3, many of these participants began by locating the program element responsible for the behavior of interest and, rather than removing it, made a controlled change to the functionality while preserving program structure. The controlled change allowed them to run the test suite and locate the tests impacted by the purely functional change. In this way, the participants could safely redefine the test suite to the target state of the software change before making the structural changes to the code. For example, P_{11} described a strategy for Task-3 as:

> *"one strategy would be to change the internal definition of the methods first so that it always executes as if it was false. That will reveal the test cases that I need to look at more closely"..*

In contrast, participations using **Local** or **Structure** strategies typically relegated interacting with the test code until the very end. In order to keep the tests running, these participants traversed the call graph to locate "leaf nodes", nodes that do not call other methods. This helps them limit the impact of their code changes and test each change before moving on the next. For example P_{18} used this strategy on Task-3 and noted:

> *"if I change it in a method where it is being passed down, it is going to lead to a lot of cascading effects, that are going to have to change other methods [. . .]. Whereas if I do it just here, I am going to only focus on this method and worry about everything else, like up top, later".*

Summary and Relationship to Earlier Work

I described three strategies that participants used to approach tasks: **Local**, **Structure**, and **Execute**. In considering how developers approach a change task, these findings are similar to those who have studied general, non-refactoring specific change tasks. Several

of these earlier works have reported the use of structural investigations as a strategy for pursuing a task (e.g., [69]). Fewer of these studies have described a focus on the execution of the system as a strategy; the closest of which I am aware is Maleej et al.'s report on the use of a debugging strategy by industrial developers [75]. I am not aware of descriptions of the use of a strategy similar to **Local** for solving tasks; however, developers have reported avoiding refactoring tools in favor of compiler errors [50, 86, 126], This study is the first to my knowledge that observes these different strategies on the same tasks.

3.4.2 RQ2. *How often do developers use automated support for refactoring versus proceeding manually?*

I investigated how often participants used refactoring tools to solve tasks by analyzing the 51 task workflows, one for each participant working on each task. A participant is considered to *use* a refactoring tool if they apply it and continue working with its output.

Out of the 51 workflows, 17 workflows (33%) contained one or more recorded *uses* of refactoring tools. In the remaining 34 workflows (67%), participants solved the tasks without using refactoring tools. As seen in Table 3.4, this was true both for more and less experienced participants. Even in an environment where experienced developers are primed to use refactoring tools, and the tasks are amenable to use, participants did not use refactoring tools extensively.

Each task in this study included steps amenable to consideration as refactorings (Chapter 3.2.3), which means that when a participant does not *use* a refactoring tool on a task, it is an occurrence of tool *disuse*. Table 3.4 shows how the refactoring tool use and disuse are distributed across participants and tasks. This data shows that all tasks had three or more participants that used refactoring tools, showing that each task was indeed amenable to the use of refactoring tools. Furthermore, the data shows that different participants used refactoring tools on different tasks, indicating that participants were not just invoking tools at every opportunity.

I also investigated whether tool use occurred more often in workflows that followed certain strategies. Since participants sometimes changed strategies, I consider, for each participant's workflow on each task, which strategy they employed and whether they used refactoring tools or not. Figure 3.4 shows how tool use and disuse occurred in workflows from each strategy. This shows that the workflows in which a participant employed a **Structure** strategy more often include refactoring tool use.

To ensure that the participants who did not use tools had created solutions that were

Table 3.4: Overview of all the refactoring tools that participants used to solve tasks and the experience of each participant. A participant used a tool if they continued working with the output of the tool.

	Exp (yrs)	**Task 1**	**Task 2**	**Task 3**
P_2	20+	-	-	-
P_3	10	-	-	-
P_4	6	-	-	ChS
P_5	2	-	-	SD
P_6	5	ExCon	-	-
P_7	8	InCon	ELVar	IM, ChS
P_8	20+	-	-	ChS
P_9	10	-	-	ChS, IM
P_{10}	10	-	IM, ELVar	
P_{11}	16	-	-	
P_{12}	11	RC	-	
P_{13}	1	-	-	
P_{14}	20+	MM	ELVar	IM, ChS
P_{15}	2	-	-	
P_{16}	10	-	-	ChS
P_{18}	2	ESc	-	
P_{19}	12	MM	-	ChS

ExCon = Extract Constant InCon = Inline Constant RC = Rename Class
MM = Move Method ESc = Extract Superclass ELVar = Extract Local Variable
IM = Inline Method ChS = Change Signature SD = Safe Delete

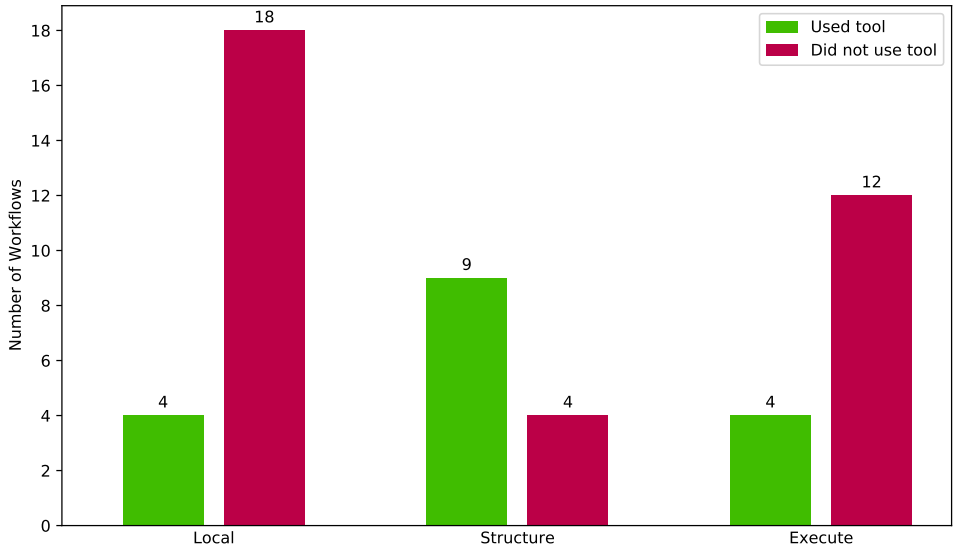


Figure 3.4: The number of times a participant employed each strategy in their workflow and used or did not use a tool in the same workflow. All the 17 participants worked on all the three tasks, which resulted in a total of 51 workflows.

amenable to automated refactoring tools, I ran the RefactoringMiner [123] tool on all participant's code solutions. If their solutions did not contain any refactorings, that would mean that these participants did not encounter opportunities to use the tools. However, the mining tool identified 385 refactorings in the code that they had produced, distributed across all participants' code solutions for all tasks. This means that every participant solved every task by changing the code in ways that were amenable to a refactoring tool being used. By comparing these results to actual tool use, I found that even participants who did use tools did in fact miss several opportunities for use by solving parts of the task manually.

By collecting all of the refactoring invocations that participants performed throughout all three tasks, I found that in total the participants invoked refactoring tools 100 times. Figure 3.5 shows how the invocations are distributed across participants. The data shows that even participants who did not use refactoring tools tried to invoke them, which invites the question of why these participants did not end up using tools.

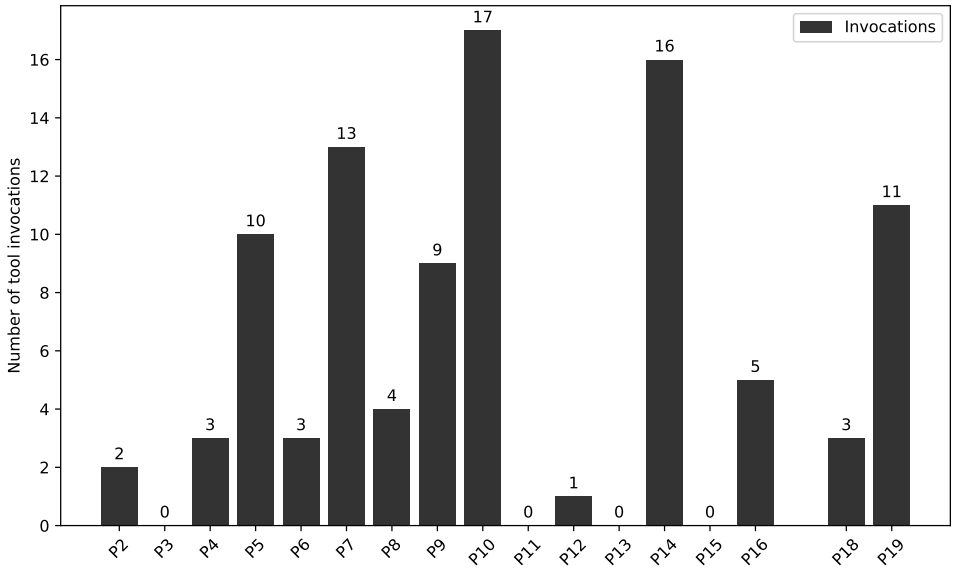


Figure 3.5: The number of refactoring tool invocations that each participants performed across all tasks.

Summary and Relationship to Earlier Work

I observed whether participants used automated support for refactoring or proceeded manually. In the 51 different cases that the three tasks posed across all 17 participants, I found that in 17 (33%) cases, the participants *used* a refactoring tool whilst in the remaining 34 cases (67%) participants solved the task by changing code manually. These numbers are aligned with previous findings in which developers self-report using tools less than half of the time [67, 112, 126, 91]. By analyzing the refactoring opportunities that were identified in participants' source code, as was done by Murphy-Hill et al. [87] and Negara et al. [91], I also found that even participants who occasionally *use* tools on a task missed a number of opportunities for tool use by performing parts of the changes manually.

3.4.3 RQ3. *How do developers experience refactoring tools?*

I investigated how participants experienced the refactoring tools that they interacted with during the study and asked them to reflect on previous experiences with refactoring tools. Previously, I reported cases in which a participant *used* tools; here it is also interesting to consider cases in which the tool was *not* used, such as when participants had experiences that led them to avoid the tool or to revert the effects that it had on

the code.

Consider that for an participant to *use* a refactoring tool, they have to go through the following process:

- they must be *aware* of the tool (e.g. by recalling a tool that can help or recognizing that a tool might be available even if they do not know the name of the tool),
- they must *try* to use the tool (e.g. by locating and invoking it),
- they must successfully apply the tool to the code (e.g. avoiding errors and not clicking "cancel"), and
- they must continue working with the tool's output (i.e. not reverting the application).

Many of the participants who did not use tools also went through some of this process but had experiences in various stages of this process that ultimately led them to not use tools.

For each task, a participant is considered *aware* of a refactoring tool if they mention refactoring tools that can help them during the task or subsequent interview segment or if they try to invoke one. This means that a participant may be aware of a tool that may help them in one task whilst not in another task. A participant is considered to have *tried* a tool if they invoke it, regardless of whether the tool produces errors, they cancel the invocation, revert what the tool did, or continue working with the resulting code. However, only in the latter case are they considered to have *used* the refactoring tool.

Figure 3.6 shows, for each participant and each task, whether the participant indicated *awareness* of refactoring tools, whether they *tried* invoking a refactoring tool, and whether they *used* a refactoring tool. The figure shows that most participants showed awareness of refactoring tools during the study but far fewer ended up using one. In fact, in all tasks, there occurred a drop-off from awareness of refactoring tools to the actual use of the tools. For Task-1, twelve (71%) participants showed awareness of a refactoring tool but only six (35%) of participants used such a tool. For Task-2, six (35%) participants showed awareness and three (17%) exhibited use, and for Task-3, fourteen (82%) showed awareness and eight (47%) made use of a refactoring tool. Only three participants (17%) made use of refactoring tools in two tasks (P_7, P_{14}, and P_{19}) despite all but one participant indicating awareness of refactoring tools.

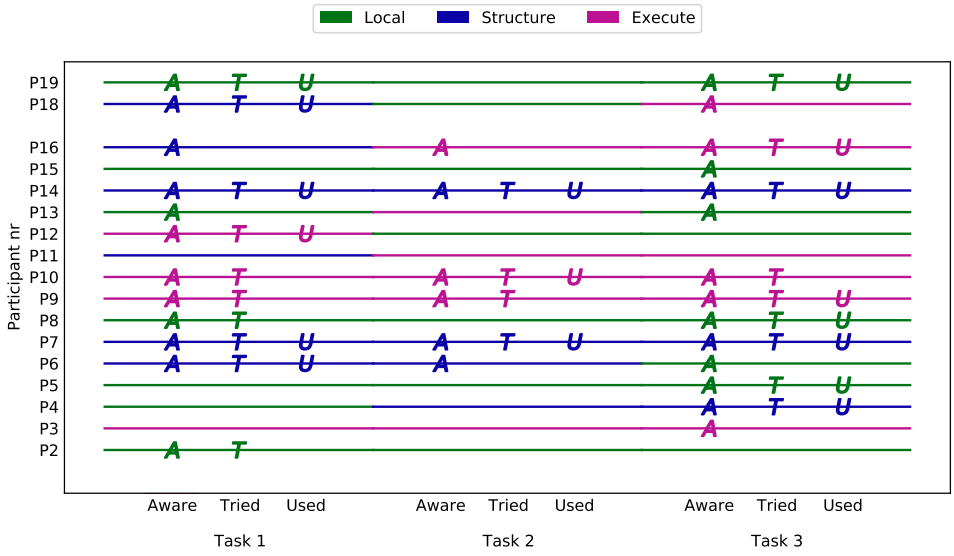


Figure 3.6: Each task presented each participant with the opportunity to use refactoring tools, for a total of 51 opportunities. In each of these 51 cases, I investigate whether the participant was *aware* of refactoring tools, *tried* to use tools, and *used* a refactoring tool once or more.

In total, out of the 51 opportunities for tool use that the seventeen participants had across all three tasks, participants expressed awareness of tools in 32 (63%) cases, and out of these 32 cases, participants tried a refactoring tool in 23 cases (72%). Out of the 23 cases in which participants tried a refactoring tool, they ended up using a tool in 17 (74%) cases. For every step, there is a drop-off of around 30 % resulting in tools being used in only 33% of cases. To understand the drop-off from awareness to attempted use and from attempted use to use, I inspected the transcripts, including the answers to interview questions, to learn why participants chose to not continue the process of tool use. I also investigated how the participants who had used tools had experienced them. In each of these steps—*awareness*, *tries*, and *usage*—I observed a number of factors that impacted how participants experienced the tools.

Awareness

Participants were mostly aware of refactoring tools. In Task-1, twelve participants (71%) showed *awareness* of tools, all of whom indicated that they could use `move` or `extract`. In Task-2, six participants (35%) showed *awareness* of tools, five of whom indicated that they could use `inline-method`, and one participant (P_6) mentioned `safe-delete`. In Task-3, fourteen participants (82%) indicated awareness and referred to `change-signature`, `inline`,

or `safe-delete`. All three tasks presented the 17 developers with 51 cases in which they had an opportunity to use a refactoring tool, and participants were *aware* of tools in 32 (63%) cases. From these 32 cases, participants moved on to *try* a tool in 23 cases (72%), while in the remaining 9 cases, they did not. I found multiple reasons why the remaining participants decided not to even *try* the tool to help with a task. The reason that was given most often by participants was that they had proceeded too far in the change before realizing a tool could help them. Another recurring reason was that the tool might negatively impact later steps in their workflow.

In Task-1, P_{13} expressed that they realized after the fact that they could have used a tool, while P_{16} did look for a way to invoke *move* from the "structure" view, which shows a truncated view of the class by listing class member declarations such as methods and fields. They failed to do so and said, *"I spent a couple of minutes and didn't get it working. I decided it is faster to do manually."*

In Task-2, P_{16} made the manual change before realizing that they could use `inline`, and P_6 did not invoke `safe-delete` because they did not "trust" the tool. When asked to explain, this participant expressed that they did not trust that the tool would present them with information about the code that would teach them the same amount of information that they would learn by manually making the change. They said, *"I prefer to see these things happening step by step because I know what's there, I can double-check any of the files that are to be changed and see if there are any other changes that need to be done in them."*

In Task-3, five participants (P_3, P_6, P_{13}, P_{15}, P_{18}) did not try refactoring tools despite being aware of them. P_3 mentioned the tool `inline` but said they would look for an automated approach only if they *"wasted too much time doing it by hand and if they trusted their tests"*. P_{13}, P_{15}, and P_{18} named `change-signature` during the interview but said that they were not sure how much this tool would have helped them. P_{13}, and P_{18} also did not realize upfront that they could use this tool, while P_{15} feared that it either would not help them or impact code that they did not yet want to change:

> *"Change signature might have helped you but not really. It would tell me that the method with the signature which is without the boolean already exist[s]. So not really. I also don't want to remove all the callers of the method because I want to look at them and see whether or not it makes sense for them to be removed or whether they should stay." (P_{15})*

A similar sentiment was expressed by P_6, who mentioned `safe-delete` but said they did not trust the tool and explained: *"I would have the problem of doing a refactoring and not knowing if the behavior of the test should be the same or the behavior of the code."* They added, *"I don't want to rely on a tool that will not help me become familiar with the source code."* explaining that they gained an understanding of the code by making changes manually and would be able

to make future changes faster due to this experience. They said that using a tool would deprive them of gathering similar knowledge.

In summary, I found that participants who were aware of tools without trying them experienced the tool as unable to present them with satisfactory information about the code and unable to distinguish test code from functional code. Both of these problems were noted by participants as making later steps in their workflow more difficult, both during the same task and if they were to work with the same codebase again. These experiences have not, to my knowledge, been previously reported. I also observed experiences that support previously reported factors that motivate disuse, like realizing too late that a refactoring tool can be used [86] and lack of trust [86, 126, 112, 72]. The data that I collected provide more detailed examples of what developers mean with "lack of trust".

Tries

In the 23 cases where participants *tried* refactoring tools, they made one or more invocations of the tools that are listed in Table 5.2. In 17 of the cases (74%), this resulted in *use*, whilst in the remaining 6, participants did not end up using any refactoring tool on that task. One could expect that participants who experienced so-called *barriers to use*[84]—warnings, problems, or errors—might avoid using tools; however, although participants did encounter such barriers in these 6 cases, so did the participants who went on to use tools. In fact, in every task, most of the participants who *tried* a refactoring tool encountered some variation of tool warnings, problems, errors, or the introduction of compiler errors, and while these experiences led participants to abandon the tool in Task-1 and Task-2, participants repeatedly relied on tools in Task-3, despite encountering barriers to use. In some cases, these were even the same participants.

In Task-1, P_2 and P_{18} had experiences when they invoked the tools that led them to avoid using the tools in the end. Both participants tried the `move` refactoring but had problems invoking the right version of `move` due to incorrectly assuming that the tool would understand their code selection as arguments to the refactoring. Unlike previously reported problems with selecting statements incorrectly (Chapter 2.6), these participants did make a correct selection of the methods they wanted to move in the editor. Then they right-clicked somewhere on the selected text and invoked `move` in the refactoring menu, thereby indicating that they wanted to `move` all of the selected methods. However, the tool failed to recognize the selection altogether and instead interpreted the invocation on the surrounding class, thereby launching `move-class` instead of `move-method`. Both participants repeated this interaction several times before giving up and indicating that they might as well move the methods manually due to the the low complexity of the task.

In Task-2, all participants who tried a refactoring tool (P_7, P_9, P_{10}, P_{14}) initially invoked `inline-method`. These participants experienced a problem that has previously been reported for other tools: misinterpreting error messages[84]. When invoking `inline` on the method to

be inlined, the tool produced an error due to this method's code structure. All four participants misinterpreted the error as relating to the structure of the caller. As a result, all four participants performed `extract-local-variable` in the caller in an attempt to bypass the error, with no success. By the time they understood the problem, three of them decided it was more efficient to manually inline the method than to fix the error. Only P_{10} performed another `extract-local-variable` correctly in order to benefit from the tool.

In Task-3, all participants experienced problems with colliding method signatures, the parameter they wanted to remove being in use, or the same problem with `inline-method` as described for Task-2. Five participants (P_4, P_5, P_8, P_{10}, P_{19}) initially cancelled the refactoring tool invocation in response to problems and inspected or edited the offending code in the editor before invoking the refactoring again, while four participants (P_7, P_9, P_{14}, P_{16}) accepted the problems upfront and applied the tool. In this task, most participants accepted that the refactoring tool took them "part of the way" so they could complete the change themselves, similar to workflows suggested by Vakilian et al. [126] All of the participants who tried a tool on this task eventually applied it, despite initially encountering problems.

The only participant who *tried* a tool without *using* it in Task-3 was P_{10}. They invoked `change-signature` and `safe-delete` several times but decided to approach the task manually to stay in control of the change. This participant first spent a few tries on getting configuration options right for `change-signature` before realizing that the tool also updated test callers. Upon discovering this lack of separation between tests and production code, they decided to not use the tool due to the inability to isolate the changes. They said, *"I don't want to like change my production code and change my test and then run my tests and see if everything is fine."* To continue on the task, they reset the code using git and tried `safe-delete` but eventually decided to solve it manually. When prompted to explain, they said, *"My fear is that safe delete would delete anything that calls it."*

In summary, I observed that participants who tried tools experienced novel problems such as incorrectly assuming that the tool would understand their selection as the refactoring argument and problems controlling the impact of the tool. The latter experience lends credibility to participants who avoided trying tools in order to not have them impact test code. I also observed previously reported experiences, like misunderstanding error messages[84] and violated refactoring preconditions [84, 126].

Furthermore, I observed that when participants encountered problems, they decided to use or not use tools based on a *cost-benefit analysis*. By evaluating whether the perceived *benefit* that the tool provided outweighed the perceived *cost* of overcoming these problems, they made the decision between pursuing solutions to problems that they encountered or switching to a manual approach. The participants seemed to evaluate these costs and benefits differently: for example, P_{10} was the only participant who only *tried* a tool in Task-3 and decided that the cost outweighed the benefits. This analysis is also dependent on the difficulty associated with performing the refactoring manually. Participants in the first two tasks indicated that

Table 3.5: Summary of refactoring tools invoked during the study. The refactoring, on the left, was invoked on the code elements seen in the middle. For example, both `inline-constant` and `inline-method` was invoked.

Refactoring	*Code Element Type*	*Count*
Change Signature	Method	36
Inline	Constant, Method	25
Move	Instance Method, Static Method, Class	19
Safe Delete	Method, Parameter	12
Extract	Constant, Local Variable, Superclass	7
Rename	Class	1

the benefits of using tools in these tasks were low due to the perceived simplicity of the code change and the low number of repeating changes. In these tasks, when the cost of using the tool increased, such as when encountering problems or failing to invoke the right tool, the threshold for favoring a manual approach was low. In contrast, the third task was more complicated and required a higher number of repeating changes which motivated a higher number of participants to "pay the cost" by figuring out how to overcome these barriers.

This is the first study that observes developers evaluate the cost and benefit of using refactoring tools on the fly. However, our findings echos a theoretical perspective from Fleming et al., who employed information foraging theory in a theoretical analysis of refactoring tools used for smell removal [43]. Tempero, Gorschek, and Angelis [118] also speculated based on their survey responses, that developers employ a return-of-investment (ROI) approach to deciding whether or not to use tools. Our findings provide empirical evidence for Fleming et al.'s prediction that, in cases where tools may have difficulty accurately approximating the human judgment required to determine whether individual code changes are appropriate, the cost of using automation increases. Similarly, in cases where the tool presents the user with information that they otherwise would have to forage for, the tool decreases the overall cost involved in performing the refactoring.

Use

Participants *used* refactoring tools in 17 workflows (33%) with varying results. In some cases, participants applied the tool with ease, such as for `rename`, `extract-local-variable`, `extract-constant`, and `inline-constant`. However, in the case of more complex[126] tools, such as `change-signature` and `move-method`, some participants made the tools work for them while others reverted to a manual approach. Similarly to participants who *tried* tools, these participants expressed that they evaluated whether the benefit of automation outweighed the cost of using it. Some factors that impacted this cost-benefit analysis were the tool's impact on test code, difficulties understanding the tool's impact on code, and code layout. The partici-

pants who successfully used tools were careful with the order in which they applied refactorings or used the information presented by the tools to organize their manual changes. In many cases, these participants made a few attempts at using the refactoring operations before they found a way to apply them that they were satisfied with.

In Task-3, participants experienced problems with the refactoring tool's impact on the test code. In this task, participants mainly invoked `change-signature`, `inline-method` or `safe-delete`, tools which propagate changes to all callers. Many participants organized their code changes in the source code file separately from the test file. However, refactoring tools do not distinguish code from tests and treat all as one program, leading to callers in the test code being changed as well despite participants invoking the tool in the source code file. While this is a natural and correct behavior from the tool, it was unwelcome by many participants. I already found that several participants who avoided trying and using tools in Task-3 mentioned this problem. Several of the participants who used tools (e.g., P_4, P_9) realized its impact on test code after applying a tool and refrained from using the same tool again. Other participants (P_5, P_7, P_{19}) only realized this impact once they began working on the test code, at which point they were unwilling to revert all their work and instead utilized git diff to learn about how each location had changed. One of the participants who used `change-signature` throughout all of Task-3, P_{19}, gave this response when asked whether the tool did what they wanted:

> *"Not completely. I think if I'd played a little more I could have excluded part of the refactoring. I'd like to say I want to go ahead with the refactoring in this area and not in this area, because if I could exclude the tests I could better see the impact of what tests are actually broken. But you want to have the proper changes done inside the code." (P_{19})*

The problem with test code highlighted the participants' dependence on diff tools to understand how the refactoring tool impacted their code. Out of the nine participants who tried a refactoring tool in Task-3, seven participants (P_4, P_5, P_7, P_9, P_{10}, P_{16}, P_{19}) resorted to git to disambiguate callers both in the source code file and the test file after applying the refactorings. These participants inspected the Preview View and Problem View before applying the tool, which, in theory, should have communicated the consequences of applying the refactoring. Nonetheless, they found that afterwards, they needed to know what the code used to do before the tool changed it, or how the tool had changed it. Several participants expressed dislike over this workflow. P_9 and P_7 made the following comments:

> *"Some of these tests should be failing because they should be testing behavior that is no longer supported. So I should go through and look at the tests. Maybe I should have done this before deleting the parameter because now it's going to be hard to look at the tests and know what they were doing before. Oops." (P_9)*

> *"By invoking the refactoring tool it changed code I wasn't looking at. So a good way*

to see those changes is through the git integration. If this wasn't in a git repository, I'd be more screwed". (P_7)

In order to limit problems related to code impact, some participants carefully orchestrated the order in which they applied refactoring tools to the call graph, starting on what they referred to as "leaf nodes", or nodes with few callers. In this way, they controlled the impact of the tool since they made sure that there were only ever a few test callers to propagate changes to. P_{16} employed this workflow throughout the entire class, while others like P_4, P_7, P_9, and P_{10} realized that this was a useful approach partway through the task. In this case, the order in which the tools were applied impacted the cost associated with using them.

Participants experienced code layout as relevant in Task-1 and Task-3. In the first task, P_{14} and P_{19} used the `move` refactoring to move one method and indicated that using the tool was not worth it. They continued the task by manually moving code. P_{14} commented that the layout of the code (i.e., the methods being declared after one another) made the manual approach easier: *"It's nice that these are all together. Then I can just wholesale them over. If they were scattered throughout then maybe it would be worth the extra steps".* Similarly, in Task-3, many of the participants (e.g., P_8, P_{11}, P_{12}, P_{14}), who manually inlined methods to their wrapper method observed the convenience of the wrapper being located just above each method declaration, allowing them to make only a single text selection. Even P_9 and P_{14}, who initially used the `inline-method` on this task, eventually reverted to this workflow. In terms of ROI, this behavior can be interpreted such that a convenient code layout means that the tool had to provide a higher benefit to be "worth it" to use.

Amongst the participants who successfully used refactoring tools throughout the entirety of Task-3, I observed an interesting workflow. Two participants used the Problem View or Preview View (Chapter 2.1) to learn about the code before changing it *manually* by acting on the information these views presented. In Task-3, P_5 used `safe-delete` and invoked it twice for each of the eight methods that had to be changed: the first time, they used the Preview View to navigate to code that would be impacted by the change and update some of these elements manually before invoking it again and applying the refactoring. P_8 used a similar workflow, where they invoked `change-signature` and inspected the "problem" view before canceling the refactoring and locating the problematic elements in the code, changing them, and reapplying the refactoring. This also provides empirical evidence for the claim posed by Fleming et al. that a developer who is refactoring needs to forage for *cues* for how the refactoring should be performed and that refactoring tools help present such cues [43].

In summary, I found that participants' success with tools were impacted by experiences with the tool's impact on test code, difficulties understanding the tool's impact on code, and code layout. The participants who successfully used tools were careful with the order in which they applied refactorings or used the information presented by the tools to organize their manual changes. To my knowledge, there is no earlier work that brings up this distinction of test code

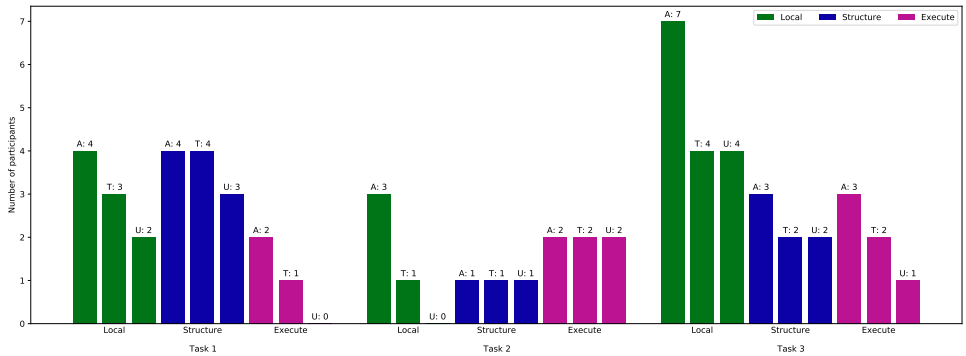


Figure 3.7: In each of the three tasks, developers from all strategies indicated awareness of tools. The leftmost bar of each color indicates, for that task, the number of participants who employed that strategy on the task and expressed awareness of tools. The middle bar of each color indicates how many participants tried a tool, and the rightmost one (if any) indicates how many participants from that strategy *used* a tool on that task.

from functional code in refactoring tools, nor reports of developer reliance on diff tools like git to learn about a tool's impact after it is applied. Some earlier works have indicated that developers avoid using refactoring tools due to fear of merge conflicts or code ownership [67]. The observations that participants organize refactoring applications along the call graph, use information from the tool to organize manual changes, and that code layout impacts the choice to use or not use tools are, to my knowledge, novel.

Experiences and strategies

Here, I considered whether participant strategies impacted their experience of tools. Figure 3.7 shows the drop-off trend relative to their strategies. The figure shows that a drop-off happens more often when the participant uses a **Local** strategy and less often when they use a **Structure** strategy. This shows that participants who used a **Structure** approach are not the only ones who are aware of tools but awareness led to use more often in this group. I speculate that participants who employed the other strategies might more often encounter experiences with the tools that lead them to avoid using them due to their workflows.

In Task-2, participants who employed a **Local** strategy were prone to removing the method that implemented the functionality *before* navigating to callers. The workflow that these participants used blocked them from using the refactoring tool unless they restarted the task. Once these participants were in a position to invoke `inline-method` (after they realized there was a caller), the tool no longer supports the refactoring because the method has been deleted, as illustrated in Figure 3.8.

In fact, across all tasks, the only participants that ever invoked `inline-method` used either a

Figure 3.8: Participants who began their work on Task-2 with deleting the method and only later attempted to invoke the `inline-method` tool were unable to do so since the declaration no longer existed.

Structure or **Execute** strategy. One explanation could be that participants who employ a **Local** strategy did not look for this tool because its name does not represent a local change. In comparison, `change-signature` or `safe-delete`, have names that might be more aligned with these participants' intent. This indicate considerations that toolmakers might use to determine the naming used in refactoring menus.

Amongst participants who encountered problems with unintended code changes in test code in Task-3, the ones using an **Execute** strategy expressed a lower tolerance for this: P_9 and P_{10} restarted the task to find a different approach and P_{16} carefully ordered the applications along the call graph. These participants' strategies were focused on keeping the tests running and isolating code changes. In comparison, many of the participants who approached the tasks with a **Local** or **Structure** strategy did not inspect test code until the end of the task, at which point they realized the tools' impact.

Finally, the two participants who exhibited the interesting refactoring tool workflow in Task-3, P_5 and P_8, were both utilizing a **Local** strategy. I observe that these participants were able to have information from the Preview View and the Problem View guide their changes in a way that is comparable to participants who had their changes guided by compiler errors: rather than considering the information that these views presented as a hindrance to applying the tools, these two participants gathered information that they used to guide their subsequent (manual) actions. While other participants sometimes inspected these views before taking other actions, I saw no other participants employ this workflow throughout the entire task.

3.4.4 Results Summary

This section presents a summary of the findings from the study and their implications.

Participants approached each task using one of the three strategies: **Local**, **Structure**, and **Execute**. The choice of strategy was dependent on the individual, not the task. Refactoring tools were used across all strategies employed by participants, but most commonly seen when participants approached the task with a **Structure** strategy despite participants with all strategies showing awareness of tools.

All participants had one or more opportunities to use refactoring tools as part of their workflows on all of the three tasks, but out of these 51 workflows, only 17 (33%) included refactoring tool

use, despite participants expressing awareness of tools in 32 (63%) of them. While the rate of tool use and disuse is comparable to that of other studies (Chapter 2.5), an exact comparison is difficult. To illustrate why, I consider what could be used as an exact measure of tool *disuse*. Out of the 17 developers that participated, 12 (71%) used a refactoring tool at any point during the study. However, when considering the 51 workflows that resulted from these 17 participants' work on three tasks, only 17 out of 51 workflows (33%) contained refactoring tool usages. Furthermore, even if a participant used a refactoring tool once during a task, they may miss additional opportunities to do so in the same task. For example, P_4 used a refactoring tool on Task-3, which means that their workflow on this task is one of the 17 cases of tool use. However, this participant only used the tool once on this task, before reverting to a manual approach. An analysis of this participant's code changes for Task-3 with the RefactoringMiner tool [123] shows the presence of eight refactorings. Consequently, the rate of refactoring tool use for this participant on this task is 1 out of 8 opportunities (14%). If a similar analysis were done of all participants, the degree of tool use would be very low.

The following experiences impacted the participants' choice or ability to use refactoring tools. Participants who both used and did not use tools experienced lacking control of, and understanding of, the impact of the tool. This occurred particularly frequently when the tool updated test callers in Task-3. Participants navigated this problem by relying on git diff or by carefully ordering the applications of the tool according to the call graph. Several participants experienced this reliance on git diff as bothersome yet unavoidable. Participants who were *unwilling* to use tools experienced that the tool deprived them of knowledge about the code that they could use later in their workflow. Participants who were *unable* to use tools experienced problems invoking them due to incorrectly assuming their code selection was passed to the tool as a refactoring argument.

When participants used refactoring tools, I observed novel workflows in which participants used the Preview View or Problem View (Chapter 2.1) of refactoring tools to organize manual changes amongst the participants using a **Local** strategy. I found several examples of participant strategies impacting their experiences with refactoring tools.

Finally, participants who decided to use or not use refactoring tools, performed a *cost-benefit analysis* on the fly, in which they compared the perceived cost associated with using a tool against the perceived benefit that the tool posed. Examples of factors that impacted this analysis are the aforementioned experiences and code layout.

3.5 Implications

The results that I presented here suggest several ways to improve refactoring tools and to better investigate the disuse of these tools. These results also call into question whether refactoring tools, as currently formulated, are the most effective way to aid developers in facing refactoring

steps that can be automated as part of their work.

3.5.1 Improving Refactoring Tools

In my study, participants were observed to struggle to use refactoring tools because of problems that they had with *controlling* the impact of the tool. They also had problems due to the tool preventing them from gaining *knowledge* about the code and about the changes that the tool made to their code. Both of these experiences gave them difficulties with subsequent steps in their workflow. While these observations did not necessarily *prevent* the participants from using tools, it led them to decide that the tool was not worth it. This implies that toolmakers must not only make tools that *can* automate refactorings but must aim at making tools that provide a larger "return on investment" (ROI) for developers than the alternative tools.

Controlling the impact of refactorings

It was surprising to observe how many participants used **Execute** strategies to perform the tasks. The participants who used this strategy were focused on keeping the tests running and in many cases, this led them to elegant solutions. In particular, after isolating the functional change of interest to the system, these participants could perform the remaining changes, such as inlining the methods (Task-2) or removing the parameter (Task-3) as a *pure* refactoring akin to removing dead code.

It was also surprising to see the participants using an **Execute** strategy frequently avoided using refactoring tools, or stopped using the tool despite initially trying it. Although the **Execute** strategy could help enable the use of refactoring tools, developers who take this approach are hindered in their use because the tools do not distinguish between functional and test code.

Throughout this work, I was reminded of the description of refactorings from Opdyke's thesis:[94, §.4.1]

> "Imagine that a circle is drawn around the parts of a program affected by a refactoring. (...) the key idea is that the results (including side effects) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle."

If the participants could have drawn a circle to exclude the tests and then could have invoked a refactoring tool on the inner part of the circle, I believe they may have been more inclined to use a refactoring tool.

Although tools frequently offer preview windows that communicate the refactoring's impact

on the program, this study illustrates the lack of fine-grained control over this impact. I saw instead that the participants who attempted to gain such control resorted to first understanding the call graph, then applying the tool to callers in order. This process had to be organized manually by the participant, which might increase the perceived "cost" of using the tool, in terms of effort. Interestingly, the importance placed on remaining in control of the change was highlighted in past literature aimed at developers but might have been neglected in tool design (Chapter 2). Toolmakers may enable such control by, for example, integrating into the tool a richer program model that goes beyond the target programming language, and take into account, for example, the separation between test code and production code, different levels of security or ownership, or that offer user-defined control of impact areas.

Understanding the impact of refactorings

When using a refactoring tool, participants in this study frequently wondered what in the program the tool had changed. Despite using the preview capability of the refactoring tool, after the tool was run and its transformation applied, every participant questioned at some point what had happened, asking: "What did the tool do?" or "How did this code change?" These participants relied on undoing and redoing the changes made by the tool or referring to git to understand how the refactoring tool had changed their code. Several described this lack of knowledge about the impact a tool had on their code as a reason they did not *trust* tools.

Previous research has linked the inability to understand the tool's impact to a lack of preview windows [126, 86, 28]. However, no previous studies has evaluated the usefulness of such views. Clearly, the preview capabilities of refactoring tools as provided in IntelliJ do not suffice for developers.

This may be explained by the tool impacting code that they were not yet attuned to think about. In several cases, the participants in my study asked these questions at a later stage in their work, such as when they moved from the functional code file to the tests. At the time that they previewed the refactoring, they were not yet focused on the test code. In addition to a preview capability, I suggest refactoring tool designers consider a comprehensive and undo-able record of changes that is accessible after the fact to help developers understand the changes the refactoring tools made to the code. This topic is explored further in Chapter 6.

3.5.2 Investigating Refactoring Tool Disuse

Studies of refactoring disuse have sometimes taken an approach of mining potential refactorings from version histories of code. [123] Such studies consider how prevalent the different refactorings may be [86, 126] and have attempted to learn why an automated approach may not have been used by interviewing the developers who committed the code [112]. In essence, these stud-

ies take a backward approach to investigating refactoring tool disuse: they locate places where refactoring may have happened and attempt to understand how tools failed or how they could have helped. The observational study reported on in this chapter takes a forward approach to investigating refactoring tool disuse by considering how the approaches used by developers enable or remove the possibility of refactoring tool use.

I based the tasks that were used in this study on commits from open-source systems for which refactoring mining tools indicated a particular refactoring had occurred. For Task-3, the RefactoringMiner tool reported the use of `inline-method`. Interestingly, only three developers in the study applied this refactoring, while the remaining participants used different refactoring operations. This is not a limitation of the tool: the refactorings were correctly identified in the source code change; instead, this discrepancy should be understood to illustrate the limitations of both the forward and backward approaches to studying refactoring tool (dis)use. The backward approach suggests a refactoring that may have been impossible for that developer to use; that developer's approach may not have enabled them to use a refactoring tool. The forward approach is limited in the breadth of approaches that the developers participating in this study considered.

I hope that the observations from this study, which show the importance and impact of the context of how a developer work, may help broaden the study of refactoring use beyond the actions of a developer close in time to when a refactoring tool is used.

3.5.3 Bridging the Gap Between Developer Workflow and Refactoring Tools

In this study, I saw evidence that the way a developer thinks about the task and the plan they make to perform the task impacts the universe of tools that might be usable as they work.

As a simple example, consider Task-1 in which participants were asked to organize test methods. Regardless of the strategy a participant used, they all solved Task-1 in one of two ways: 1) either by *creating* a new class and *moving* methods into the class; or 2) by *duplicating* code (e.g., copy-pasting an entire file) and *reducing* each file to contain only the appropriate code.

The universe of tools that the participant might consider to help perform the task was impacted by which of the ways they proceeded to work on the task. Some participants using the *create-move* approach considered a `move` refactoring tool for moving methods into their new class. I conjecture that it was easier for the participants to map the way that they were working to a tool that was similarly named and seemed to match the actions that they wished to perform.

Other participants struggled with mapping the description of their approach—their cognitive model—to the refactoring tools available. For example, P_2 did attempt to use a tool after a prompt but was unable to invoke the correct refactoring: *".. what am I moving, am I moving*

the entire class? That's not what I want. Extract? Oh that didn't work. Extract .. Method? Well that's not what I want either." They reverted to their manual approach.

Developers who approached the task using *duplicate-reduce* immediately also reduced the opportunities to employ a refactoring tool. Neither the operations they wished to perform, nor the way they were thinking about the problem, lent themselves to refactoring operations.

A similar reduction of opportunity occurred for `inline-method` in Task-2. Participants who approached this task by deleting the method and inspecting the resulting compiler errors would learn about the caller too late. By the time that these participants are able to recognize that they should apply `inline-method`, there is no longer any method to inline. Consider that the "cost" of undoing their changes so far and restarting the task is higher than if they had used the tool from the beginning.

When participants could easily match their intent to a tool description, they also had more success in using refactoring tools. For example, during Task-3, several participants were able to select `change-signature` without ever having used it before. I speculate that this is because the name matched their immediate activity. For example, when P_4 was prompted about why they picked this tool, they said *"That seemed like the appropriate thing because that is what I was doing."* In this task, participants who were able to have the tool invocation capture their intent could also utilize the information it provided them in their manual changes, as was observed with P_5 and P_8.

A challenge in improving refactoring tools is to bridge the gap between the way a developer approach a task that they are performing and the functionality of the tools that they have available. Tempero, Gorschek, and Angelis [118] speculate that an unstated barrier for developers to use refactoring tools is to restate their goals as refactoring operations. I observed an even larger problem: many participants would have needed to deviate from their workflows in order to encounter the code change that was supported by the tool. The gulf between tools and workflows may be larger than has been previously considered for refactoring tools.

Future studies should consider developer approaches to tasks in more detail. Perhaps it is possible to guide a developer to a strategy and workflow that enables the use of refactoring tools through suggestions? In other cases, such as where code layout enables easier manual changes, developers may benefit from help with a cost-benefit analysis to evaluate whether using a refactoring tool is worthwhile. Alternatively, there may be other tools more useful to developers, such as ones that help assess the impact, risk, or consequence of changes to the code. Such a tool may support manual changes a developer is performing that may or may not be similar to refactoring operations.

3.6 Threats to Validity

There are various threats to the validity of this study and the findings and conclusions presented here.

3.6.1 Construct validity

The construct validity of this study is impacted by various aspects of the study setup. The participants in the study knew the study was about refactoring tools and were thus primed and more likely to try to use the tools than in their normal workflow. I was willing to accept this threat to enable more occurrences of refactoring tool use. Participants reported making choices of proceeding manually on tasks, so I believe the impact of this threat on the study results is reduced.

In some cases, participants were also not working with their normal development environment and tools, which may have impacted their approach. I mitigated this threat by answering questions they had about the tools that were provided. Finally, participants were working on unfamiliar tasks on an unfamiliar system and were being asked to think-aloud as they worked. I accept these threats to enable detailed observation and comparison of different individuals on the tasks.

3.6.2 Internal validity

The internal validity of this study could have been affected by the choice to use an observational study in which the experimenter interacted with the participants. These interactions might have impacted participants' actions and responses, as can their awareness of being observed. I chose this study design because I determined that it was necessary to observe and interact with developers to ensure that they provided complete descriptions of their process and to pick up on topics to pursue in interviews. I took steps to mitigate these risks by remaining neutral during sessions, not bringing up refactoring names before participants did and delaying certain questions until after all tasks were done. I accept that this threat can not be fully mitigated.

3.6.3 External validity

This study also has threats to the external validity of the findings. The study included only three tasks and only seventeen participants. I chose the tasks to replicate ones that had been committed to open source systems, thus improving the realism of the tasks. Six participants made unprompted comments about having encountered similar tasks to Task-3 in their own

work, which indicates that this threat is mitigated. I recruited participants who represent our target demographic in terms of experience levels and gender distribution. With these steps, I believe the findings are sufficiently valid to generate hypotheses to drive considerations for tool improvements and for the role of refactoring in change tasks.

3.6.4 Conclusion Validity

In an observational study like this one, a risk to conclusion validity is to inadequately analyze or understand the available data. The labeling of transcripts used in the analysis may be biased by the experimenters' interpretations of developer statements and actions. The conclusions that were drawn from the analyzed transcripts may be biased by their priming from reviews of the previous literature. I attempted to mitigate the effects of these biases by detailed the coding method and by having more than one individual involved in the analysis. I also included in the study design an interview segment where participants were asked to elaborate on or explain points or actions that the experimenter noted during their work on the tasks.

Future observations of industrial developers at work or case studies could enable investigation of our findings in situations that reduce the threats associated with this study design. Moreover, these findings could be validated by means of an online survey in a larger sample of the demographic.

3.7 Summary

This chapter describes the design and execution of a lab study in which 17 practitioners each solved three software change tasks that contain steps that are amenable to automation with refactoring tools. The tasks were based on real software changes that were stored in source code repositories and were motivated by realistic software change goals. In Chapter 2, I described the necessity of studying how developers approach such tasks in order to improve the support that refactoring tools can offer in these scenarios.

With this study, I identified new insights into how developers approach such tasks and how they use and experience refactoring tools. I have described how these insights can be used by future researchers and toolmakers to improve tool support for developers. A full replication package is available for other researchers [38]. This contains both the experiment tasks and instructions on how to repeat it as well as the collected dataset which enables replication of the analysis presented in this chapter.

Chapter 4

Refactoring Tool Usability

Numerous studies have found that refactorings frequently occur as part of software changes and that developers underuse refactoring tools (Chapter 2.5). In attempting to understand why, researchers have discovered various factors that contribute to use and disuse of refactoring tools. For example, they have found that developers lack trust in tools, that the tools are unpredictable, that developers lack awareness and familiarity with tools, and that the tools disrupt their workflow and do not support the operations that they need (Chapter 2.6).

Despite this list of observations, previous work provides few actionable items. Many of these factors are not well understood and consequently, difficult to address. In particular, there is little known about how to make tradeoffs between different factors, such as supporting more operations (which can be improved by implementing more refactorings) and predictability (which can be improved by implementing fewer and simpler refactorings). For example, *trust* and *predictability* have been persistent topics in the research of refactoring tool usability, yet there is no agreed-upon definition of exactly what makes refactoring tools trustworthy or predictable (e.g., [24, 92]). Researchers have suggested addressing these two factors with Preview Views [87, 28] and yet, no studies have shown their effectiveness. In fact, in the laboratory study described in Chapter 3, I observed that Preview Views were not sufficient support for the participants to predict how the tool would impact subsequent steps in their workflow.

These examples illustrate the challenges in addressing factors that prohibit the use of refactoring tools. It is also difficult to merge the findings from different studies into a comprehensive framework that can be used to systematically improve these tools. The different studies employ different definitions of the term refactoring as well as the differences in the contexts, tasks, and experience measures (Chapter 2.4). To help researchers and toolmakers use a standard framework to approach the experience of refactoring tool usability, I use a standard definition of usability and apply it to refactoring to provide a *theory* of usability for refactoring tools. I then use the theory as an "orienting lens [to help guide] what issues are important to examine" [32, p.69] in refactoring tools. With the transcripts described in Chapter 3, I investigate

the participants' experiences with tool usability and usability problems. This analysis resulted in four usability themes and help me refine the proposed theory.

Part of this chapter has been published elsewhere [40]. The analysis artifacts are available in the online datapackage [37].

4.1 A Theory of Usability

I based the framework on the ISO 9241-11 [5] standard, which defines usability as *"the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"*.

4.1.1 Application

I apply this definition of usability to refactoring tools, considering the user, their goals, and the meaning of effectiveness, efficiency and satisfaction to a user employing one or more of these tools.

The ***user*** of a refactoring tool is a software developer who has access to, and a need for, the tool. Refactoring tools are employed by software developers during all stages of software development, including the creation, evolution and maintenance of software. Developers primarily apply refactoring tools as part of other changes [86]. Both novice and expert developers have easy access to invoke the automation in commonly used development environments during software change tasks, through top-level menu options, keyboard shortcuts and Quick-Assist options directly in editor windows.

The ***goal*** of a developer who applies a refactoring tool is to change the code according to their intent. Several studies find that refactoring most often occurs with the intent of preparing or completing functional changes [112, 95]. This use may seem surprising: why apply a refactoring if the goal is to alter, and not preserve, behavior? Developers choose to use the tools as part of functional changes to avoid making similar, error-prone code changes in many locations [83, §2.2], [46, 104] or to enable reuse of existing code [112, 74]. Thus, the goal is to perform the functional change, and the tool helps achieve it without degrading code quality or introducing errors.

The quality of ***effectiveness*** indicates that a developer can successfully use a refactoring tool to achieve a desired code change. For a developer, effectiveness is a result of their ability to locate and apply desired refactoring operations. This differs from the tool perspective where effectiveness is achieved by ensuring that the tool supports the specifications of refactoring operations.

The quality of ***efficiency*** relates to the overall time or effort that is required to achieve the intended change. Efficiency is impacted by the speed with which the developer can operate the tool and integrate its result into their workflow and the speed with which the tool processes files (see e.g. [66]). For example, if the developer has difficulty finding [88] or invoking [84] the correct refactoring operation, it may be more efficient to use a manual process or a simpler tool, such as `grep` [2]. Efficiency can also be impacted by the time that the developer may need to spend locating, verifying or reverting code changes after applying a refactoring tool. From the tool perspective, efficiency is addressed as the speed required to analyze and change code.

The quality of ***satisfaction*** indicates that the developer experiences using the tool as a positive, such as the tool adding value and acting reasonably. Satisfaction is naturally impacted by a lack of effectiveness and efficiency: if error messages produced by the tool are not understandable to the developer or the tool is slow, the developer is unlikely to be satisfied.

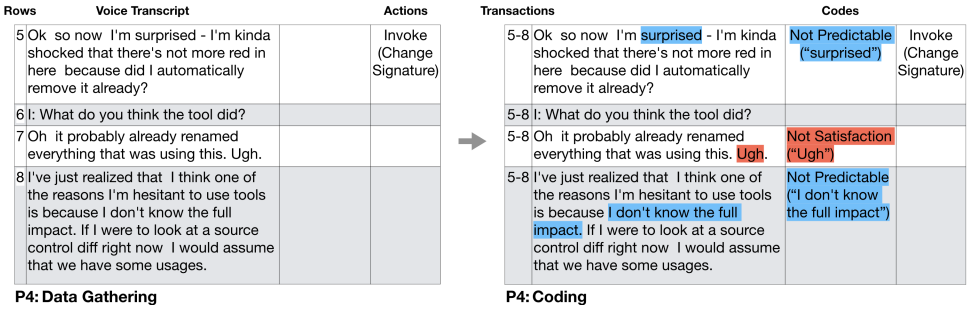
4.1.2 Proposed Theory

This application of the ISO definition of usability to the context of use for refactoring tools leads to the following proposed theory:

> Software developers employ refactoring tools to help prepare or complete functional changes to a software system. Software developers seek these tools to be reasonable to locate and apply for a desired intent (*effective*), reasonable in terms of time and effort required to use the tool (*efficient*), and to add value to the development process (*satisfying*).

4.2 Analysis

I use the theory of refactoring tool usability and concepts from related work (Chapter 2) to code the transcripts resulting from the laboratory study described in Chapter 3. The study comprised approximately 32 hours of study sessions in which 17 individuals with professional development experience and some knowledge of refactorings solved three software change tasks. The tasks contained steps that were amenable to the use of refactoring tools but the participants were not instructed to perform refactorings or to use refactoring tools. Each study session was screen-captured with audio. As seen in Figure 4.1, I created a transcript for each session from the recorded screen capture and audio captured information (labelled Data Gathering in Figure 4.1). To ease analysis, the transcripts are marked with refactoring tool invocations. In total, the 17 transcripts comprise 134,947 words. These form the main data source for this



Rows	Voice Transcript		Actions
5	Ok so now I'm surprised - I'm kinda shocked that there's not more red in here because did I automatically remove it already?		Invoke (Change Signature)
6	I: What do you think the tool did?		
7	Oh it probably already renamed everything that was using this. Ugh.		
8	I've just realized that I think one of the reasons I'm hesitant to use tools is because I don't know the full impact. If I were to look at a source control diff right now I would assume that we have some usages.		

P4: Data Gathering

Transactions		Codes	
5-8	Ok so now I'm surprised - I'm kinda shocked that there's not more red in here because did I automatically remove it already?	Not Predictable ("surprised")	Invoke (Change Signature)
5-8	I: What do you think the tool did?		
5-8	Oh it probably already renamed everything that was using this. Ugh.	Not Satisfaction ("Ugh")	
5-8	I've just realized that I think one of the reasons I'm hesitant to use tools is because I don't know the full impact. If I were to look at a source control diff right now I would assume that we have some usages.	Not Predictable ("I don't know the full impact")	

P4: Coding

Figure 4.1: Excerpt of transcript from participant 4 as seen in the data gathering phase (on the left) and coding step (on the right).

Table 4.1: Summary of refactoring tools invoked during the study.

Refactoring	*Code Element Type*	*Count*
Change	Method Signature, Class Signature	36
Inline	Constant, Method	25
Move	Instance Method, Static Method, Class	19
Safe Delete	Method, Parameter	12
Extract	Constant, Superclass	7
Rename	Class	1

analysis. All analysis artifacts I use are available to other researchers [37].

The total study data contains 100 invocations of refactoring tools by 15 (88%) participants. The remaining two participants did not invoke refactoring tools during the study. Five participants (29%) did not use refactoring tools to solve the tasks despite awareness of refactoring tools. Three out of these five participants tried (invoked) tools during interviews after being prompted, but did not use them to solve the tasks. Table 4.1 describes which refactoring tools the participants invoked during the study. These tools form the basis of participant experiences during the study. All participants also recalled and reflected on their experiences with refactoring tools they typically use; I consider participants' comments related to tool invocations and their recall of previous experience equally in this analysis.

4.2.1 Coding of Transcripts

As the goal in analysis is to refine the theory, I created a codebook from the three usability factors described in the theory—*effectiveness*, *efficiency* and *satisfaction*—and two dominant factors from earlier studies—*trust* and *predictability*. I added these last two factors because of their prevalence in earlier work and because they pertain to the developer's overall experience with refactoring tools. In contrast, other factors noted in earlier work, such as *configuration*,

Table 4.2: Codebook for Analyzing Transcripts

Code	**Description**
Effective	User experiences the tool as successfully performing a desired result.
Efficient	User experiences the tool as being fast or increasing productivity.
Satisfaction	User experiences having their needs or wants met by the use of the tool.
Trust	User experiences the tool as trustworthy, safe or reliable.
Predictable	User understands upfront what will happen by using the tool or indicates a lack of surprise when using the tool.

focus primarily on the mechanism or event of refactoring. This approach is consistent with using a theory early in a qualitative analysis [32]. Table 4.2 summarizes the codes used in analysis.

This analysis was conducted in collaboration with Gail Murphy. I annotated each transcript with the codes from the codebook. A comment made by a participant in a transcript could be assigned more than one code. Each code was recorded either as a positive statement or as a negative statement. Gail Murphy reviewed the coding and where disagreement on assigned codes occurred, we discussed the difference until agreement was reached. We used this approach to improve the likelihood of applying the codebook definitions consistently. This approach for reaching agreement has been used elsewhere (e.g., [60]).

As an example, the right-hand side of Figure 4.1 shows comments with associated codes for a portion of participant #4's (P_4's) transcript. In this case, the `change-signature` refactoring tool did not perform as P4 expected ("Not Satisfaction"), changed more than P_4 expected ("Not Predictable") and P_4 had to resort to using other tools to investigate what the refactoring tool had changed, expressed to be ("Not Efficient").

The coded transcripts are available for others to inspect [37].

4.2.2 Analysis of Codes

A naive analysis of the coded transcripts could lead to an over-approximation of some codes as the ways in which participants expressed usability factors in comments varied. Some participants were sparse in their descriptions, whereas others were more verbose. To help normalize across descriptions, we manually grouped sequences of coded comments in the transcript if they represented one thought of a participant. For ease of reference, I refer to a grouped set of comments as a *transaction*. For example, all comments in Figure 4.1 (rows 5-8 of the transcript) group into the same transaction as they refer to *thoughts about* the same tool invocation.

To support the investigation of the predominance of usability factors across the experience of participants, I used association rule mining [19], which reports itemsets—co-occurring sets of codes—above a pre-defined support level, where support is the number of transactions containing the subset of codes. An itemset of size one indicates occurrences of one particular code; an itemset of size two indicates two co-occurring codes, and so on. I applied the *apriori* algorithm [18] as found in the python `mlxtend` library [8] to find association rules in the transactions from the coded transcripts. When applying association rule mining, I consider the codes both with sentiments attached (e.g., "Satisfaction" and "Not Satisfaction") and not attached. I also count each code only once per transaction even if there are several occurrences of the code. For example, the transaction in Figure 4.1 is coded as {Not Predictable, Not Satisfaction} with sentiment and {Predictable, Satisfaction} without sentiment. The use of association rule mining lets us investigate if one usability factor may indicate the presence of other usability factors, as in whether predictability likely indicates satisfaction.

4.2.3 Analysis of Comments

To guide an investigation of the participants' views about refactoring tool usability, we use the comments organized by codes. The aim is to find themes across the transactions for an itemset. Card sort [31] is an approach to detecting themes in qualitative data where one or more individuals organize cards into groups until themes emerge. This is comparable to the thematic analysis employed in [111] whereas due to the richness of our data, we perform one individual card sort for each itemset of size one (i.e., each code) instead of searching for themes across all comments simultaneously.

I performed the card sorts in collaboration with Gail Murphy. We discussed and resolved differences of opinion.

We created "cards" (using Trello [15]) where each card represents a transaction (grouped set of comments) associated with that itemset; for each transaction, we included on the card comments immediately preceding and immediately succeeding the transaction to provide sufficient context for interpreting the information. Figure 4.2 shows a snippet of the card sort for transactions coded with "Efficiency".

By employing one card sort per itemset, we found themes that emerged from comments pertaining to that usability factor. As it was likely that some themes would be similar across factors, we looked for common themes across the card sorts and "collapsed" them together by performing a "meta-card sort". In the meta-card sort, each card was a theme from the previous iteration. For example, the theme "Cost-Benefit Analysis" in the "Efficiency" card sort in Figure 4.2 was a single card in the meta-card sort.

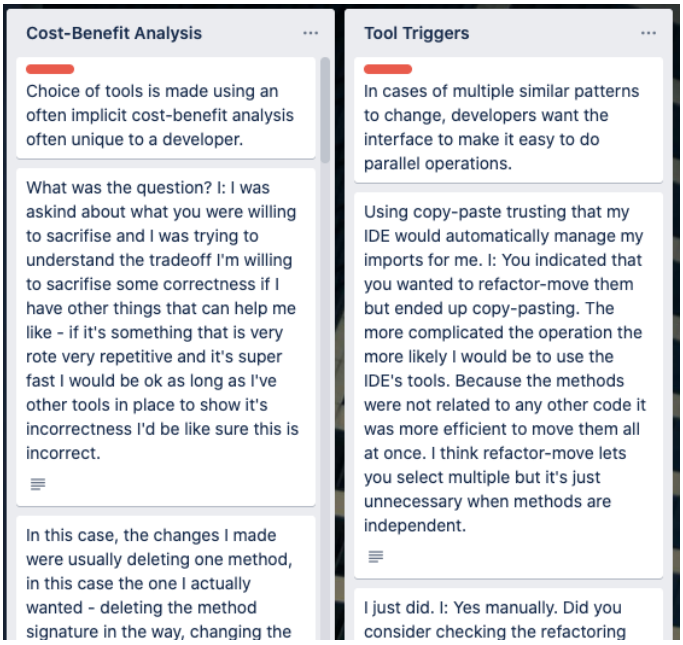


Figure 4.2: Snippet of Card Sort for Transactions including Efficiency code. Cards without labels are transaction comments from transcripts; cards with labels (the two topmost cards) are theme-cards.

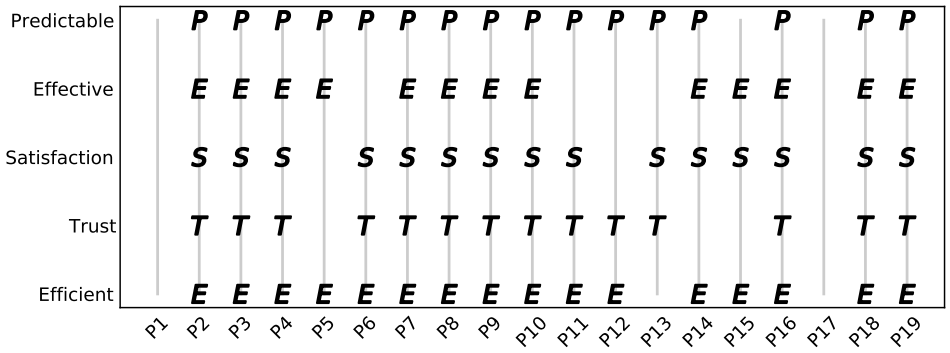


Figure 4.3: If the transcript of a participant (x-axis) contains one or more statements labeled with a usability factor (y-axis), then I mark the coordinate (participant, factor) with the first letter in the factor. For example, P_6 has a transaction labeled Predictable and is marked with a P. P_6 did not have a transaction marked Effective, so that coordinate is left blank. This figure shows that factors were distributed across the participants and that frequent factors like Predictable and Efficient were represented in statements from almost all participants.

4.3 Results

Coding the transcripts resulted in comments being labelled with 284 codes from which 143 transactions can be formed. Here, I report on the prevalence of codes and themes about usability factors based on this data. Figure 4.3 shows that the usability factors were distributed across participants.

4.3.1 Prevalence of Usability Factors

Table 4.3 reports the itemsets found by applying the *apriori* association rule mining algorithm to the codes in the transcripts when sentiments associated with the codes are ignored. The mining algorithm was applied with a support of 0.15 as a means of finding itemsets that appear in at least 15% of the transactions. The resulting itemsets are shown in Table 4.3.

The top row indicates that in over half of the transactions coded, participants refer to predictability of the refactoring tools. The prevalence of discussion of this usability factor over others indicates that refactoring tool designers may wish to pay special attention to how developers view the predictability of actions of a tool when invoked. In contrast, while participants also referred to trust, they did so the least of all factors. This data starts to bring out differentiations of usability factors to explore relative to earlier research that reported on multiple usability factors uniformly.

Table 4.3: Itemsets with Non-Sentiment Attributed Codes

{Predictable}	0.55
{Effective}	0.38
{Satisfaction}	0.36
{Efficient}	0.29
{Trust}	0.21
{Predictable, Satisfaction}	0.18
{Satisfaction, Effective}	0.16
{Predictable, Effective}	0.15

Observation #1: Participants commented frequently on the predictability of refactoring tools.

Table 4.3 also includes three itemsets with more than one usability factor. Predictability is present in two out of three itemsets and occurs together with satisfaction (in 18% of transactions) and effectiveness (in 15% of transactions). Interestingly, participants did not often refer to more than one usability factor (e.g., support values are less than 0.2). It is also interesting that participants did not often refer to efficiency or trust together with other factors: all three itemsets of size two involve predictability, satisfaction and effectiveness. I hypothesize that focusing on mechanisms that support these usability factors may help refactoring tool designers deliver tools of more interest and value to software developers.

I also ran the *apriori* algorithm on sentiment attributed codes, such as when a participant's comments were coded as "Not Satisfaction". The top two results had negative sentiments attached: "Not Predictable" with support of 0.31 and "Not Satisfaction" with support of 0.27. However, participants may be more likely to vocalize negative sentiments. Successful use of tools, such as `rename`, did not lead to many comments; participants were more vocal when encountering frustrating or confusing tools.

4.3.2 Usability Factor Themes

To gain a deeper understanding of the themes underlying participants' comments, we performed five cardsorts on each code (itemsets of size one in Table 4.3). Table 4.4 shows the number of cards considered in each card sort, the number of cards discarded and not associated with a theme, and the number of themes identified. For example, for the Predictable card sort, 79 cards were considered, of which 27 (34%) were discarded. A card was discarded if the comments on the card did not provide a coherent insight about refactoring tool use. From the cards considered, 7 themes were identified for the Predictability usability factor. The results of the card sorts are available in the data package [37].

To give a sense of the emergent themes, I describe a theme from the Predictable card sort and

Table 4.4: Card sort information for item sets of size one (individual usability factors).

Cardsorts	Cards	Discard	Themes
Predictable	79	27	7
Effective	53	22	6
Satisfaction	51	19	5
Efficient	42	12	5
Trust	31	7	5
Total	256	87	28

from the Efficient card sort.

One of the seven themes from the predictable card sort is *review and convince*. This theme describes 15 cards from 10 participants. This theme summarizes comments made by participants that expressed how participants wanted to check how a refactoring tool changed the code after invocation. Participants wanted to perform this check as a means of convincing themselves that the tool made changes that the participant expected. I saw several participants use other tools, such as git diff, to accomplish such a check. Participant P_7 commented:

> *"By invoking the refactoring tool it changed code I wasn't looking at. So a good way to see those changes is through the git integration."*

Observation #2: Participants rely on ad-hoc solutions to understand code changes after applying a refactoring.

As a second example, one of the five themes from the efficient card sort is *cost-benefit analysis*. This theme summarizes 22 cards from 13 participants. This theme describes that participants evaluated the benefits of using the tool against the cost of using it. For instance, participant P_8 tried using `change-signature` during Task-3. During the interview they said:

> *"I tried using the refactor tool, change signature. It is all right, it is useful I think when you are changing a lot of places. In some cases, it was not so useful because it was as much work to run that as to go and change those places."*

Observation #3: Participants lacked support for up-front cost-benefit analysis of applying a refactoring tool.

4.3.3 Usability Themes

The meta-card sort across the 28 themes resulted in 4 themes, listed in Table 4.5. Two of these themes capture requests about what refactoring tools need to convey to developers: namely to

Table 4.5: The Usability Themes that emerge in the meta-card sort of the 28 initial themes.

Theme	Description	P	Efe	S	Efi	T
Tool communicates capabilities	A tool's user interface must communicate clearly and directly to a developer in terms familiar to a developer. The tool should guide a developer in its use, including providing intelligible error messages.	✓		✓		
Tool communicates change	A tool should make the changes it will make to code clear before the tool is executed. It should be possible for a developer to inspect the changes that the tool has made.	✓	✓	✓		✓
Developer guides tool	Developers wish to guide how a tool executes in several ways: applying an operation to many elements easily, excluding the application to some locations of code, and altering how particular code is changed.	✓	✓	✓	✓	✓
Developer cost-benefit analysis	Developers assess the cost of applying a tool before they invoke the tool. Developers want to assess whether it is better to proceed with a tool or manually at the start of considering using a tool. The tool should help them with this assessment.	✓	✓		✓	

P = Predictable Efe = Effective S = Satisfying Efi = Efficient T = Trustworthy

make the capabilities of the tools more apparent to a developer and to better communicate to a developer what the tool does as, and after, the tool is invoked. The other two themes capture additional interactions that developers desire to have with refactoring tools, namely to guide how a tool operates and to enable a developer to better assess whether to use the tool or not.

4.4 Discussion

I consider how the results of the analysis impact the theory of refactoring tool usability introduced earlier, implications for designers of refactoring tools, the relationship of the insights into usability theories and factors to earlier work, and threats to the study results on which this dissertation relies.

4.4.1 Theory of Refactoring Tool Usability: Revisited

In Chapter 4.1, I presented a theory of refactoring tool usability derived from the ISO 9241-11 definition of usability. I use the insights gained from the study and resultant analysis of par-

ticipants' comments to refine this theory. In particular, I note that the four themes presented in Table 4.5 cover the comments coded as representing effectiveness, satisfaction and efficiency. I use these themes to revisit and refine a theory of refactoring tool usability proposing the following as a refined theory:

> Software developers employ refactoring tools to help prepare or complete functional changes to a software system. Software developers seek these tools to be reasonable to locate, and for the tools to help them assess the *efficiency* of the tool, in terms of the costs and benefits of the tool, before its use. To enable *effective* use in multiple situations, software developers seek to guide how a tool changes the source code for a system; this ability to tailor how a tool works can improve the *efficiency* of the tool for the developer. Software developers also seek refactoring tools to explain their impact to source code so that the software developer can understand the *effectiveness* of the tool. Software developers also expect tools to communicate clearly and directly in terms that match how software developers perceive refactoring operations. These characteristics in a refactoring tool increase the *satisfaction* of the software developer using a refactoring tool.

4.4.2 Implications for Refactoring Tool Designers

Over the years, substantial improvements have been made to refactoring tools to improve their usability. For example, many refactoring tools now include Preview Views that provide a glimpse into what parts of the code will be changed if the operation is applied. As another example, some refactoring tools present options to a developer: `inline-method` in IntelliJ lets the developer choose between:

- Inline one caller and keep the declaration unchanged.
- Inline all callers and keep the declaration unchanged.
- Inline all callers and delete the declaration.

These tool interfaces start exposing the complexity of refactoring operations and can be seen as inviting a dialogue between the software developer (the user) and the refactoring tool.

The observations that I made during the study and themes that I drew out about tool usability indicate three major and one minor set of implications for refactoring tool designers to hone the dialogue between developer and tool.

Up-front Impact. The first major implication, based on the 'Developer cost-benefit analysis' and 'Tool communicates change' themes (Table 4.5), is that refactoring tools need to commu-

nicate much more information about the potential impact of the refactoring at the start of the refactoring process. This information is necessary for two reasons: 1) users must be able to recognize the *impact* that a change will have on their code to understand if they want to apply it, and 2) users must be able to evaluate the *cost and benefit* of using the automated tool to apply the change rather than using a manual approach. Both of these needs relate to deciding whether to apply the refactoring.

At first glance, one might think this is precisely the reason for Preview Views. I found through the study that the information provided by Preview Views was insufficient for both purposes. Even developers who engaged with the Preview Views were surprised at the impact of the refactoring after they were applied, or ended up applying refactorings that introduced unwanted changes, which subsequently needed to be reverted or fixed. Using the dialogue metaphor, developers wanted a richer start to the conversation with which they engage a refactoring tool.

Guided Change. A second major implication, based on the 'Developer guides tool' theme (Table 4.5) is that once a developer decides to proceed with a refactoring, they want control of where and how a refactoring is applied. Refactoring tool designers should consider how to enable a developer to step through code changes associated with a refactoring, similar to how a step function in a debugging tool works. In each step, the developer could inspect the code location, inspect the change proposed by the refactoring tool, and perform potentially location-specific code alterations.

This approach would solve several usability problems that I identified in the study: it would let the developer perform *additional changes* on each location; it would guide the developer through the code so that they *gather knowledge* about it; it would allow the developer to *validate* each code change as it was applied; it would help the developer understand the tool and increase *trust*.

Of course, enabling developers to skip or change the refactoring tool's intended change at each step also means that the final code may not have preserved behavior, contradicting the safety guarantee of the refactoring tool!

Perhaps surprising to some, this lack of safety is already possible depending on choices developers make in interacting with the Preview Views and Configuration Views provided by existing refactoring tools (Chapter 2.1). For example, invoking `move-method` on a JUnit test method in IntelliJ, causes the refactoring tool to reject the invocation, and to propose making the method static and moving it. If the developer agrees, the method becomes static and the developer will cease to be able to execute it as a test method. The result is a behavior change despite using a refactoring tool. P_{19} encountered this situation in Task-1 but failed to recognize that he agreed to an unsafe change. There are opportunities for refactoring tool designers to walk developers through changes and explain the impacts of skipping or performing code changes suggested by a refactoring tool. In particular, the distinction between behavior-preserving changes and non-behavior-preserving changes should be made explicit to the developer.

There is an interesting design space to explore between safety guarantees that can lead developers to disuse tools and supporting unsafe code changes that developers might choose to use more often. By addressing this implication, tool designers can give the user more control of how the conversation unfolds.

Communicating Change. A third major implication, based on the 'Communicating Change' theme (Table 4.5), is the need for developers to understand what a refactoring tool did to their code. Numerous times in my study, I saw developers resort to ad-hoc solutions and additional tools (e.g., git) to understand what a refactoring tool had changed in their code. Participants were not always satisfied with this solution. P_{19}, for instance, said:

> *"Sometimes inspecting changes after the fact can be hard. There is going to be code that is unrelated to your refactoring, since normally refactoring tasks is done as part of other tasks. When you're looking at it after the fact you may also not be able to undo it if something doesn't look right . . . "*

Refactoring tool designers should consider how to provide summaries of their impact on the code. Perhaps this could be as simple as what is provided by a git diff operation. Or, perhaps a user could replay changes made by the refactoring tool to understand why changes are being made. This support can be considered a summary provided at the end of a conversation between the user and the refactoring tools.

Initiating Use. A more minor implication than the other three is how to start a conversation between the user and the appropriate refactoring tools. I observed developers struggle to understand what a refactoring operation might be, and to then invoke the desired refactoring operation. For example, in Task-1, participant P_2 made many attempts to invoke `move-method`, but failed to do so because their model of how they should invoke the tool differed from the tool's affordances, and the mismatch was not clear. Specifically, the participant made a text-selection across multiple methods as shown in Figure 4.4 since they wanted to invoke `move-method` on all of them and thought that the tool would recognize their intent. However, the tool recognized that the menu was invoked on the class body and invoked `move-class` instead, which resulted in the tool presenting them with a Configuration View that they did not understand, as shown in Figure 4.5. The participant finally realized that they were invoking the wrong type of move and moved on to `extract`.

> *"What am I moving am I moving the entire class? That's not what I want. Extract? Oh that didn't work. Extract .. Method? Well that's not what I want either. 'Cause that implies you just got a small section of code."*

Participant P_{18} had a similar experience that lead to them introducing a bug. These participants had the following problem: the tool failed to communicate which actions mattered (the

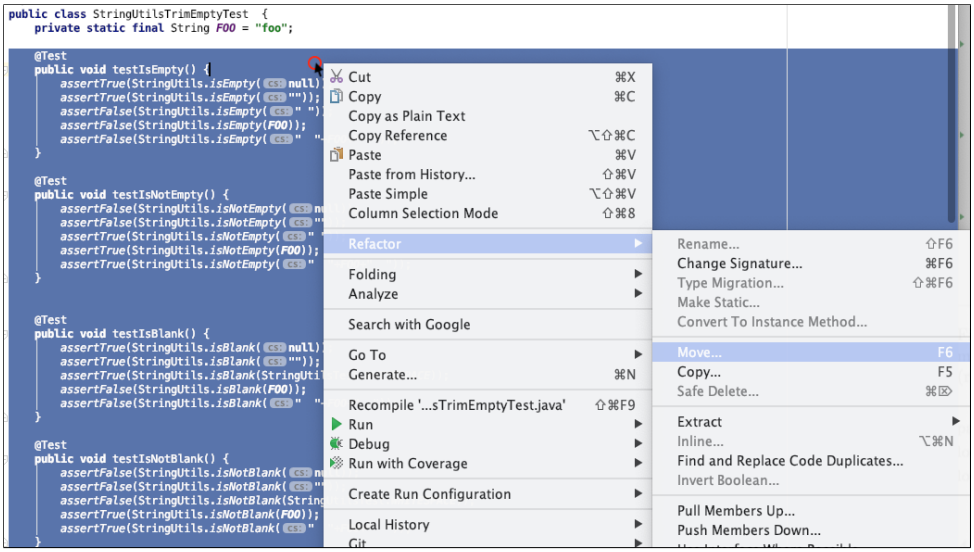


Figure 4.4: The two participants selected a number of methods that they intended to move and invoked the `move` refactoring by right-clicking in a seeminglys random location (red circle), which happened to be in the class body.

location of the right-click) and what didn't (the selected area). Therefore, they were unable to locate and invoke the functionality they wanted.

4.4.3 Relationship to Earlier Work

As described in Chapter 2.6, multiple studies have made observations about factors that contribute to use and disuse of refactoring tools, such as developers lack trust in tools, the tools are unpredictable, developers lack awareness and familiarity with tools, the tools disrupt their workflow and do not support operations they need. Despite this list of observations, previous work provides few actionable items. In particular, there is little known about how to make tradeoffs between different factors, such as supporting more operations (which can be improved by implementing more refactorings) and predictability (which can be improved by implementing fewer and simpler refactorings). By forming a theory about usability and investigating developer experiences through that theory, I have detected themes that capture developer desires and expectations from refactoring tools. These insights can be used to design the next generation of refactoring tools.

The four usability themes that were derived from the study (Table 4.5) encapsulate a number of steps in a refactoring process. Some of these steps are not included in the process described by Mens and Tourwé [78], such as having the tool better communicate the change and having more ability for a developer to guide how the tool proceeds with changes. Interestingly, one usability

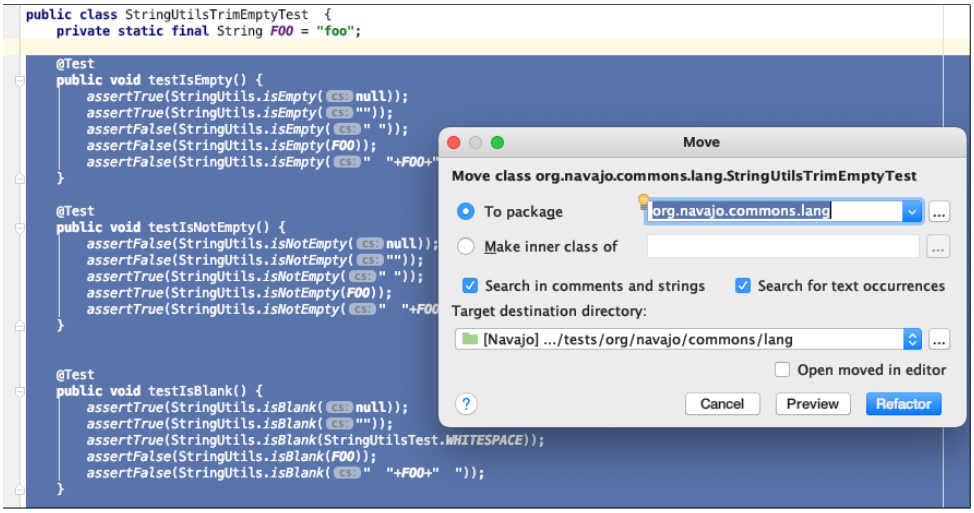


Figure 4.5: The refactoring tool detected the location of the mouse click rather than the selected methods and invoked the `move-class` refactoring. However, the participants intended to invoke the `move-method` refactoring and did not understand what they were doing wrong.

requirements produced by Mealy et al. is that a tool should let the developer inspect, change, and cancel the change before it is applied [77]. The Preview View do offer this functionality, however, while this might suffice for conceptually simple changes, I found it insufficient for the developers that I studied. They needed greater control over individual steps of the change than what a Preview View could offer.

The dialogue metaphor that captures many of these steps and that I discuss above aligns with findings by Vakilian et al. [126] and claims by Brant [24] that developers accept unsafe tools as long as the tools provide value. The emphasis of a developer's ability to guide how the tool proceeds with changes is also present in the usage pattern presented in Murphy-Hill's model of refactoring tool use [82], where the developer reacts to the output from the tool. In these ways, my findings confirm previous work while also extending with new observations and providing a framework in which to consider multiple aspects of usability together.

4.5 Summary

Despite the prevalence and availability of refactoring tools across many development environments, software developers do not use these tools. Previous efforts at understanding why have identified numerous problems in the interaction between the tool and the developer, such as lack of predictability, lack of trust, and inability to use the tool to automate the type of operations a developer want to perform.

With the work presented in this chapter, I have sought to investigate how to improve the usability of refactoring tools using the lens of a theory that I derived from an ISO standards definition of usability. Using this theory, I studied the experiences of 17 developers who were asked, in a lab study, to work on three change tasks designed to be amendable to the use of refactoring tools. The analysis of the data that I collected from them resulted in four emerging themes about the usability of refactoring tools, which I used to refine the theory. These themes provide insights into what developers desire and need from refactoring tools. In particular, these themes suggest that refactoring tool designers need to consider how refactoring tools provide information to a developer and how developers can be given more control about how refactoring tools operate.

These findings also provide insights for software engineering researchers. First, the theory of refactoring tool usability can be built on by others who are researching this topic. Second, I have introduced a systematic means of considering usability factors of software development tools from a user's perspective instead of a tool's perspective; this perspective change can lead to a questioning of implicit assumptions of what tool designers may believe (i.e., refactoring tools must be safe) and what developers want (i.e., the reduced invocation-cost of an unsafe change). I hope that this might enable the building of tools (refactoring or others) that are used more often and that developers find effective, efficient and satisfying.

Chapter 5

Usability Profiles of Refactoring Tools

Developers use refactoring tools to perform only between 10 % and 50 % of refactorings, with the remainder performed manually (Chapter 2). Manually performed refactorings still involve the use of tools, such as compiler outputs, search, and navigational tools. To decide which tools to use—including refactoring tools—developers perform a cost-benefit analysis on the fly based on their experiences with the tools and the tasks (Chapter 3).

In this chapter, I focus on understanding this cost-benefit analysis. I investigate how developers experience the usefulness and usability of the many tools they use to approach software change tasks that are amenable to automation with refactoring tools. With this insight, I aim to support toolmakers with insights into how design tradeoffs impact the developers' perception of the tool as useful.

This chapter reports on survey that I conducted to investigate how developers experience the various usability factors (Chapter 4) of tools that are used to approach software change tasks that are amenable to refactorings, and what factors might impact their experience. The survey resulted in a number of *usability profiles* for a number of tools and insights into factors that impacts developers' experiences. These insights can aid toolmakers in creating refactoring tools that are more frequently used, and to aid developers in deciding which tools to use.

5.1 Introduction

Developers use a multitude of tools to approach software change tasks that include refactorings. Ideally, automated refactoring tools would provide benefits during change tasks and developers would choose to use them. However, in practice, developers often choose to rely on other

tools. While previous work has collected a number of factors that characterize developers' choice of tools (Chapter 2.6), it is non-trivial to merge the disparate study findings into a shared framework that enables systematic investigation and evaluation of tradeoffs between such factors since the studies focus on different tasks, different contexts, and different experience measures.

In Chapter 3, I observed that developers perform a cost-benefit analysis on the fly to decide which tools to use, and base this analysis on their approach to a task at hand (Chapter 3) and their experiences with the tools (Chapter 4). In Chapter 4, I presented a framework for capturing such experiences. The study presented in this chapter aims to explore the cost-benefit analysis that developers employ to determine which tools to use on tasks that are amenable to automation with refactoring tools.

In Chapter 3, I also described three tasks and the approaches that developers took to the tasks. These developers used one of three different strategies. Developers who employed a **Structure** strategy were more likely to use a refactoring tool, while developers who employed a **Local** or **Execute** strategy were less likely to use these tools. However, while some developers used refactoring tools, they also relied on `compiler-output` (e.g., compiler errors and warnings), `text-search` (e.g., *find* and *find and replace*), `copy/cut/paste`, and `structural-navigation` (e.g., *find usages* and *go to declaration*). They used these tools to understand the code, and to perform and verify code changes. Both developers who used refactoring tools and the ones who not did, frequently relied on `VCS-diffs` (e.g., git diff) to learn how the code had changed and `test-suite` (e.g., JUnit) to verify that their changes were correct. Some developers also used `debug-tool` to understand the code. Earlier work has also observed that developers use tools like compiler errors [50] and various search tools [51] to perform refactorings "manually". In this chapter, I investigate how developers experience the usability and usefulness of these tools.

In Chapter 4, I found that the following factors captured developers' experiences with the usability of refactoring tools: *trust*, *predictability*, *satisfaction*, *effectiveness*, and *efficiency*. The individual factors varied in the frequency with which developers referred to them, potentially indicating variance in their importance. In this chapter, I investigate whether the strategy that a developer uses impacts the importance they place on the different factors.

Such insights can help developers in making tradeoffs between factors. Refactoring tools cannot always both meet developers desires (e.g., *satisfaction*) and be fully correct (e.g., *trustworthy*) [24]. For example, a toolmaker might be able to increase a refactoring tool's *effectiveness* and *efficiency* by accepting some erroneous invocations. However, in doing so, the tool's *trustworthiness* might be reduced because it will occasionally introduce errors. As another example, a toolmaker might increase the tool's *trustworthiness* and *predictability* by limiting the accepted invocations to programs that can be easily analyzed and transformed. However, that might reduce the *effectiveness* of the tool because there may be many situations in which it cannot be applied. This illustrates how toolmakers must make tradeoffs in the design of their tools and

the difficulties in doing so.

With this survey, I aim to investigate the following research questions:

- **RQ1.** *Which tools do developers find useful for software change tasks?*
- **RQ2.** *Which usability factors—efficiency, trust, effectiveness, predictability, satisfaction—do developers deem as necessary for a tool?*
- **RQ3.** *Do change tasks impact developers' experience of usability factors?*

5.2 Methodology

I conducted a descriptive study of developers' perceptions and experiences of the usefulness of various tools that support refactorings that occur during software change tasks, including the factors that affected their experiences. The set of tools is taken from the tools that developers were observed to use in the laboratory study described in Chapter 3. The software changes described in the study are also adapted from the tasks given in the laboratory study. To measure the developers' experiences, I used the five usability factors described in Chapter 3 and developed a survey instrument that was distributed for the purpose of recruiting individuals with recent, professional experience performing software changes or maintenance of software written in Java or C#. This section gives an overview of the defined study scope, the survey design, recruiting, and data analysis. The full survey can be found in the replication package [9]. The study was designed and executed in collaboration with Gail Murphy, whom I will refer to as the *co-investigator*, and myself as the *investigator*.

The full survey text is included in Appendix B.

5.2.1 Scope

This study addresses developers' perception and experience of usefulness of tools and the various usability factors that impact this experience. I investigate whether the developers' experiences of the usefulness of various tools vary when they consider software change tasks in general and when considering concrete software change scenarios. The survey asks respondents to consider scenarios which focus on software change goals that can be decomposed into subgoals involving refactoring. These subgoals are interesting because: 1) they are prevalent and frequent during the software change process, 2) they are time-consuming and error-prone to perform manually, and 3) there are a wide variety of tools available to support the activities but adoption of many of the tools is not widespread.

The study focuses on software written in Java, due to the popularity[1] and maturity of this language and its tool support. The survey scenarios contain code examples written in Java. Respondents were eligible to take the survey only if they had sufficient recent professional experience with Java or C#. C# was included despite the focus on Java because the languages, tools, refactoring and practices in these two languages are comparable.

5.2.2 Design

The survey consisted of three main parts: 1) background questions, 2) tool questions, and 3) three specific software change scenarios. Figure 5.1 depicts the overall survey flow and respondent requirements related to each part. If a respondent did not meet a requirement, the survey was terminated and that participant was shown a screen that explained the violated requirement; these responses were excluded from the study. Information pages were interleaved with questions to define survey terminology. For example, Table 5.1 shows the survey terminology that was presented to developers after the consent page.

Background questions

I collected the following demographic information: age, gender, highest level of completed education, current job title, and self-described proficiency with both Java and C#. Respondents were asked to rate their proficiency with changing software in these languages on a scale from one to five, where one indicates no proficiency and five indicates expert proficiency.

The respondents' relevant experience was recorded in the form of total years of experience in the software industry and total years of experience with developing or maintaining software written in Java or C# professionally. I also asked specifically for how many years the respondent has worked on such software within the last ten years (2010-2020). Respondents were only allowed to participate if they reported two or more years within the last ten. This requirement helps filter respondents whose habitual work flows are not aligned with modern tools and processes, and to ensure that respondents were able to understand the code change scenarios.

Demographic questions were placed at the start of the survey to ensure the information was available for analysis even if the respondent chose to end the survey before completion.

Tool questions

This part of the survey asked about the respondent's perceptions and experiences with tool use during software change tasks. The survey presented nine types of tools shown in the excerpt

[1]Java is the most popular object-oriented language according to the Tiobe index: `https://www.tiobe.com/tiobe-index/`, Accessed March, 26 2021

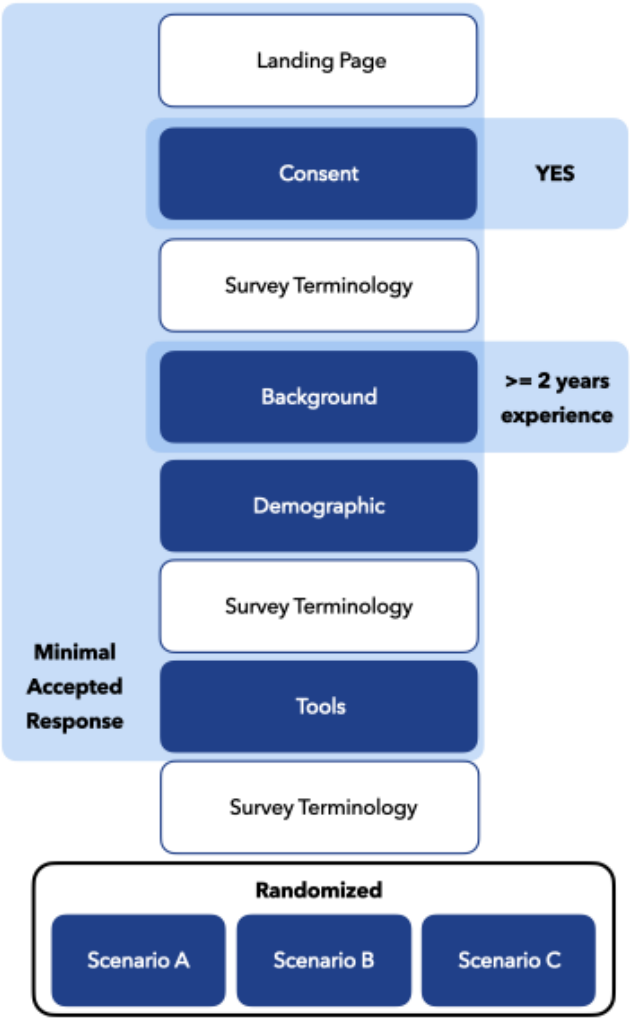


Figure 5.1: An overview of the survey's flow. Participants first arrive at a landing page that presents them with information (white), then move on to a consent screen which requires input (dark blue). If they consent (requirement in bold), they are shown another info screen that introduces the survey terminology. Then, they are asked qualifying background questions. If they indicate the required experience, they are shown demographic questions before another information screen presents more survey terminology. Then, they are asked about their experiences with tools. All survey responses that include answers to this question block are included in the analysis presented here, even if they exit the survey before responding to scenarios. If they continue from this screen, they are presented with another terminology screen and then with the scenario questions.

Table 5.1: Excerpt from the survey terminology page shown to participants.

SURVEY TERMINOLOGY
Software change tasks refers to activities where developers apply changes to existing software for purposes such as adding or refining a feature, correcting a mistake, or improving code quality. In addition to applying changes, they may need to take actions that help them understand the code such as navigating or reading source code, test code and documentation. Developers may also need to inspect or validate changes they have already made. We use the term **tools** to refer to programs that help support or automate the types of software change activities previously outlined: applying changes, inspecting changes, validating changes, and comprehending source code, test code and documentation.

Table 5.2: Excerpt from the tools question page.

The following is a list of tools that software developers may find useful during software change tasks. Please mark all the tools that you find useful during software change tasks.
☐ Compiler output (e.g. compiler errors) ☐ Debugging tools ☐ Copy/cut and paste code ☐ Version Control Systems "diffs" (e.g. git diff, etc.) ☐ Structural navigation (e.g. find references, go to declaration) ☐ Textual search (e.g. grep, find and replace) ☐ Test suites (e.g. JUnit, test errors) ☐ Simple refactoring tools (e.g. Rename, Extract Constant, Inline Method) ☐ Complex refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)

in Table 5.2. Types of tools were referred to instead of concrete tools (e.g., version control system `diff` instead of git `diff`, and test suites instead of JUnit) because the respondents might use different version control systems, test frameworks, compilers and refactoring tools, but the concepts are shared between the different concrete tools. The tools investigated in this survey are the types of tools that developers were observed to use in the software change tasks described in Chapter 3.

The survey also asked which IDEs a respondent uses to change software in Java or C#. Since the implementations of tools and the degrees of tool support vary between IDEs, this information helps contextualize the responses.

To capture the developers' experiences with the various tools in a survey format, I used the five usability factors introduced in Chapter 4: *effective*, *efficient*, *satisfying*, *trustworthy*, and *predictable*. In pilots of the survey, respondents reported difficulties interpreting the usability factors when they were given as keywords (e.g., *trust*, or *effective*). To improve the respondents' understanding of the terms, I developed a set of statements aligned with how developers in the

Table 5.3: Usability factors and practitioner statements used in survey.

Usability Factor	Statements
Effective	The tool is effective if it is reasonable to locate and apply for a desired intent.
Efficient	The tool saves time or effort if it is reasonable in terms of time and effort required to use.
Satisfying	The tool is satisfying to use if it adds value to the development process.
Trustworthy	The tool is trustworthy if it feels safe or reliable to use, for example by rarely or never making errors or mistakes.
Predictable	The tool is predictable if it behaves consistently and predictably each time you use it.

laboratory study expressed these factors. Table 5.3 presents these statements. Respondents in subsequent pilots of the survey after these statements were introduced indicated a better understanding of the terms.

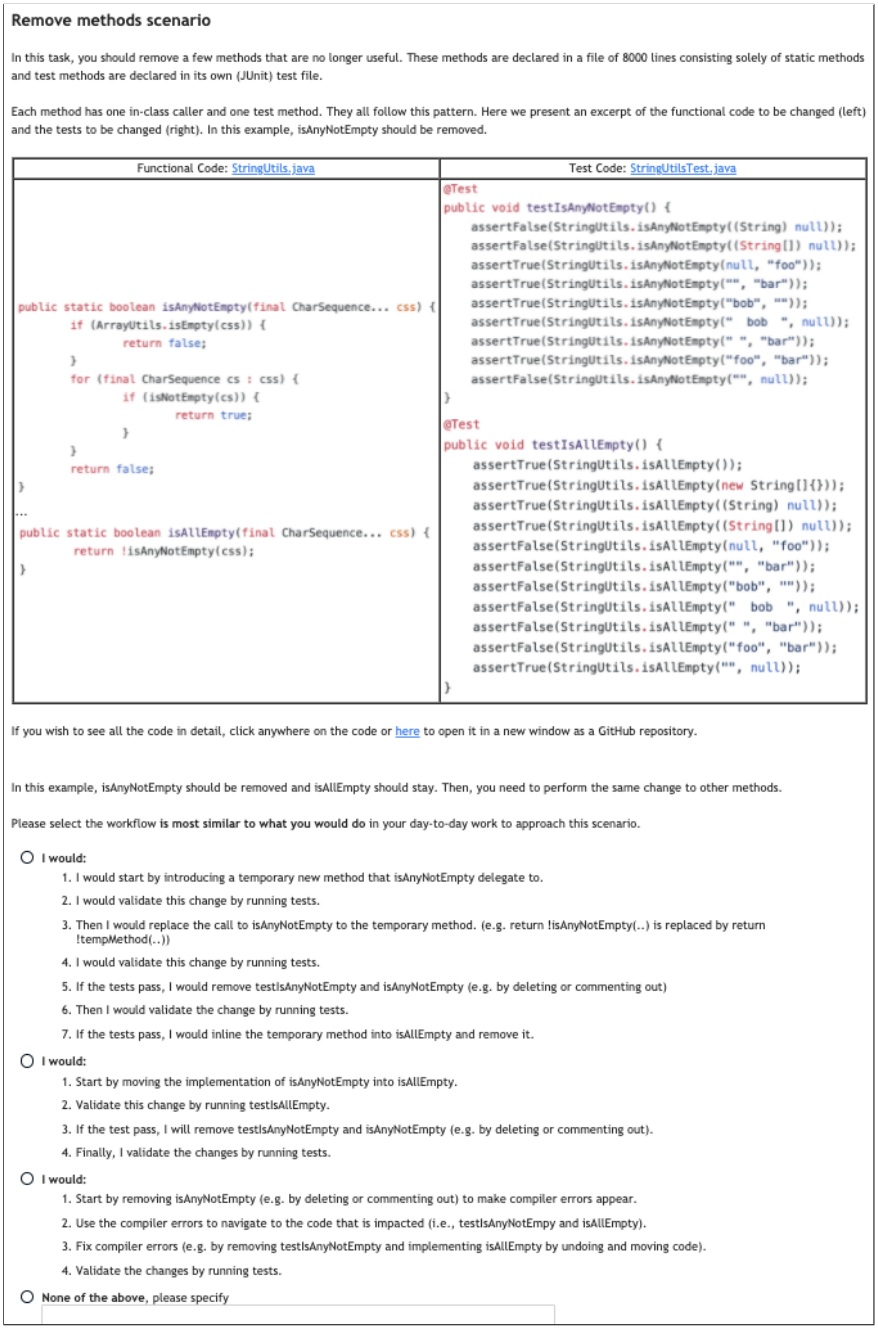
Respondents marked, for each tool in the list, all statements that they deemed as necessary for that tool to be useful. The question page contained a 9x5 checkbox matrix (9 tools by 5 factors) of all tools and all factors, so that each respondent was shown 45 boxes. Respondents could select none, some, or all boxes.

Scenario questions

The survey useed three scenarios of software change tasks to investigate whether developers have different perceptions or experiences of tools when considering the tools in the context of a task and in a general case. Table 5.4 describes the three scenarios, and Figures 5.2-5.4 shows Scenario B as presented in the survey over a sequence of three consecutive pages. The scenarios are based on the tasks described in Chapter 3.

Initially, each participant was shown all three scenarios in randomized order. However, it became evident from the drop-out-rates and feedback from survey participants that they found the workload of the survey to be too high. After several months of data collection, 27 participants had provided answers to one scenario, but only 17 of them had proceeded through all three scenarios. At this point, the two investigators made the decision to alter the study's design to present only one of the three scenarios to each participant. This reduced the estimated time of the survey significantly. The scenario was selected at random, and a balancing algorithm was used to ensure an even distribution of participants between the scenarios. After this, several additional participants responded to the survey.

The description of each scenario began with a high-level goal and asked respondents to select



Remove methods scenario

In this task, you should remove a few methods that are no longer useful. These methods are declared in a file of 8000 lines consisting solely of static methods and test methods are declared in its own (JUnit) test file.

Each method has one in-class caller and one test method. They all follow this pattern. Here we present an excerpt of the functional code to be changed (left) and the tests to be changed (right). In this example, isAnyNotEmpty should be removed.

Functional Code: StringUtils.java

```
public static boolean isAnyNotEmpty(final CharSequence... css) {
        if (ArrayUtils.isEmpty(css)) {
                return false;
        }
        for (final CharSequence cs : css) {
                if (isNotEmpty(cs)) {
                        return true;
                }
        }
        return false;
}
...
public static boolean isAllEmpty(final CharSequence... css) {
        return !isAnyNotEmpty(css);
}
```

Test Code: StringUtilsTest.java

```
@Test
public void testIsAnyNotEmpty() {
    assertFalse(StringUtils.isAnyNotEmpty((String) null));
    assertFalse(StringUtils.isAnyNotEmpty((String[]) null));
    assertTrue(StringUtils.isAnyNotEmpty(null, "foo"));
    assertTrue(StringUtils.isAnyNotEmpty("", "bar"));
    assertTrue(StringUtils.isAnyNotEmpty("bob", ""));
    assertTrue(StringUtils.isAnyNotEmpty("  bob  ", null));
    assertTrue(StringUtils.isAnyNotEmpty(" ", "bar"));
    assertTrue(StringUtils.isAnyNotEmpty("foo", "bar"));
    assertFalse(StringUtils.isAnyNotEmpty("", null));
}
@Test
public void testIsAllEmpty() {
    assertTrue(StringUtils.isAllEmpty());
    assertTrue(StringUtils.isAllEmpty(new String[]{}));
    assertTrue(StringUtils.isAllEmpty((String) null));
    assertTrue(StringUtils.isAllEmpty((String[]) null));
    assertFalse(StringUtils.isAllEmpty(null, "foo"));
    assertFalse(StringUtils.isAllEmpty("", "bar"));
    assertFalse(StringUtils.isAllEmpty("bob", ""));
    assertFalse(StringUtils.isAllEmpty("  bob  ", null));
    assertFalse(StringUtils.isAllEmpty(" ", "bar"));
    assertFalse(StringUtils.isAllEmpty("foo", "bar"));
    assertTrue(StringUtils.isAllEmpty("", null));
}
```

If you wish to see all the code in detail, click anywhere on the code or here to open it in a new window as a GitHub repository.

In this example, isAnyNotEmpty should be removed and isAllEmpty should stay. Then, you need to perform the same change to other methods.

Please select the workflow **is most similar to what you would do** in your day-to-day work to approach this scenario.

- ○ **I would:**
  1. I would start by introducing a temporary new method that isAnyNotEmpty delegate to.
  2. I would validate this change by running tests.
  3. Then I would replace the call to isAnyNotEmpty to the temporary method. (e.g. return !isAnyNotEmpty(..) is replaced by return !tempMethod(..))
  4. I would validate this change by running tests.
  5. If the tests pass, I would remove testIsAnyNotEmpty and isAnyNotEmpty (e.g. by deleting or commenting out)
  6. Then I would validate the change by running tests.
  7. If the tests pass, I would inline the temporary method into isAllEmpty and remove it.
- ○ **I would:**
  1. Start by moving the implementation of isAnyNotEmpty into isAllEmpty.
  2. Validate this change by running testIsAllEmpty.
  3. If the test pass, I will remove testIsAnyNotEmpty and isAnyNotEmpty (e.g. by deleting or commenting out).
  4. Finally, I validate the changes by running tests.
- ○ **I would:**
  1. Start by removing isAnyNotEmpty (e.g. by deleting or commenting out) to make compiler errors appear.
  2. Use the compiler errors to navigate to the code that is impacted (i.e., testIsAnyNotEmpy and isAllEmpty).
  3. Fix compiler errors (e.g. by removing testIsAnyNotEmpty and implementing isAllEmpty by undoing and moving code).
  4. Validate the changes by running tests.
- ○ **None of the above**, please specify

Figure 5.2: This shows an excerpt from Scenario B as shown in the survey. Respondents are first shown a page that lists the task and the three approaches.

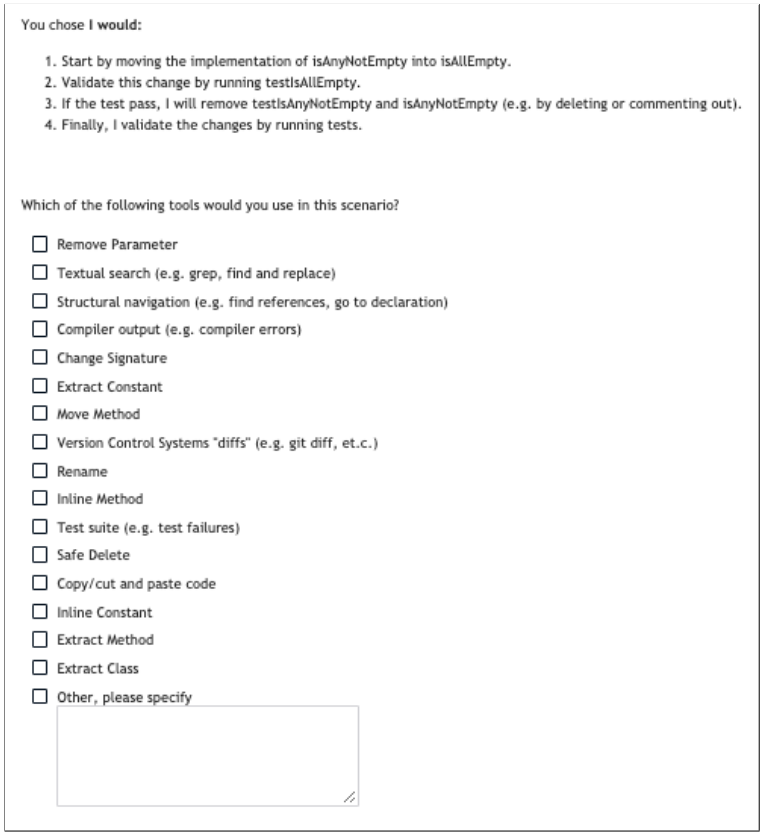


Figure 5.3: Once they select an approach and continue, the next page prompts them to select the tools they would use in that approach.

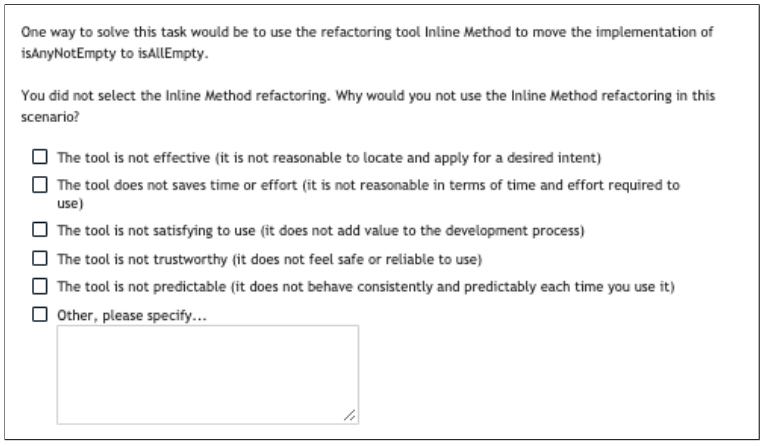


Figure 5.4: If they do not select the particular refactoring tool, the next page ask them to describe why not.

Table 5.4: Software change scenarios used in the survey.

Scenario	Description
Scenario A	Requires reorganizing test methods by creating a new test class and moving 12 test methods located in two different files into the new class. Relevant refactorings: `move-method`. Replicates part of an Apache Commons-Lang commit [11].
Scenario B	Requires removing two methods, each with a single method that depends on them, and their test methods. Relevant refactorings: `inline-method`. Replicates an Apache Commons-Lang commit [12] that is discussed in this pull-request [13].
Scenario C	Requires removing a parameter from eight methods in a single class; each method is overloaded with a wrapper method without the parameter. Relevant refactorings: `change-signature`. Replicates part of a commit to Quasar [14] that was collected and analyzed by Silva et al. [112].

one of the three approaches that they were most likely to use in this scenario, or to provide their own. The approaches were created based on workflow trends, referred to as strategies, that the participants in the laboratory study were observed to use (Chapter 3). In a **Local** strategy, a developer might rely on the editor and compiler: the developer edits the code they see in their editor and then uses compiler errors to navigate to affected code elsewhere and progresses by solving compiler errors. In an **Execute** strategy, a developer aims to make edits in groups that allow a test suite to continue to execute correctly: in this strategy, a central tool is the test suite. Finally, in a **Structure** strategy, a developer organizes their edits according to code structures (like callers and type hierarchies): in this strategy, structural navigational tools and refactoring tools are central.

Each scenario contains steps that are amenable to automation with certain refactoring tools, as indicated in Table 5.4. If a respondent did not select that tool, on the next page they were presented with the question in Figure 5.4 and asked to describe why they did not do this.

5.2.3 Data Collection

To recruit survey respondents, the two investigators used social media advertising and snowball recruiting [52] seeded through industry contacts known to them. The entire survey was estimated to take maximum 40 minutes with all three scenarios and 15 minutes with only one scenario. The resulting data is anonymous. Respondents received no compensation.

The survey response period lasted from January through September, 2021. During this time, the landing screen received 182 visitors. 120 visitors continued to the consent screen, from which there was a drop-off throughout the study. 119 visitors consented to participate in the study, out of which 60 did not proceed past the background question block, either because they were not eligible to participate or because they chose to leave the survey. Out of the remaining 59 respondents, 13 respondents left the survey before answering any of the questions relating to their experiences with tools. The responses from the remaining 46 participants form the dataset that is described and analyzed in this chapter.

This dataset represents the 46 respondents who were eligible to participate, and completed at least the questions pertaining to their experience with tools. These respondents have extensive experience from the software industry (on average, 12 years), from software changes in Java or C# (average total: 10 years; average for the last ten years: 6 years). All respondents reported at least three (i.e. above average) proficiency in at least one of the two languages.

Most of the respondents (64%) have a current title of Programmer or Software Engineer. Two thirds (77%) of respondents have completed higher education on the level of Master's or Bachelor's degree. Half of the respondents (50%) reported being less than 35 years old. 41 (87%) of respondents are male.

5.2.4 Analysis

The data gathered from the survey is largely nominal in nature; that is, it cannot be ordered or measured quantitatively. As a result, descriptive statistics are employed to summarize the gathered data. In the case of free-form comment data, the answers given were too few to perform analysis.

To describe the data, the following notation is used: scenarios are named Scenario A, B and C; elements in the set of tool types are written using typewriter font, as in `VCS-diffs` and `complex-refactoring`; strategies used to approach a scenario are identified by name, **Local**, **Structure**, or **Execute**; and usability factors are written in italics and referred to as *trust*, *predictability*, *satisfying*, *effectiveness*, and *efficiency*.

I wanted to compare responses about tool use in the scenario questions with responses from the general tool questions. Scenarios contain specific refactoring types (e.g., Change Signature, Move Method) while the tool questions refer only to simple and complex refactoring tools. In the scenario, a participant who selects one or more refactoring tools is considered to want to use a `refactoring-tool`. When comparing this to responses from the tool questions, I consider both simple and complex refactoring tools as `refactoring-tool`.

Where heatmaps are used to show relationships in the data, the numeric data in each cell is scaled to a number between 0 and 1 by dividing it by the sum of each row. To clarify, I

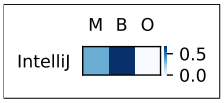
include a simple example in which I plot the relationship between three IntelliJ users, Alice, Bob, and Charlie, and their education. Bob has a Master's degree while Alice and Charlie have Bachelor's degrees. This raw numeric data can be presented as follows:

	Master's degree	Bachelor's degree	Other
IntelliJ	1	2	0

I scale this to a number between 0 and 1 by dividing by the sum, 3.

	Master's degree	Bachelor's degree	Other
IntelliJ	0.33	0.66	0

Then, I plot the scaled numbers in a heatmap where color saturation increases with higher ratios.



When there are multiple rows, this calculation is performed per row. When heatmaps contain IDEs, I omit those that were selected by less than two users.

5.3 Results

I consider how the data from this survey provides insight into each of the three research questions posed in the beginning of this chapter. I will use the term developer for survey respondent in this section to emphasize that the findings relate to developers.

5.3.1 RQ1. *Which tools do developers find useful for software change tasks?*

As context for interpreting the survey results, it is helpful to consider which of the tools that the developers find useful in general software change tasks. The developers largely agreed that most or all of the nine tools presented in the survey were useful during software change tasks: 14 developers (30%) found all tools useful; 10 (22%) found all but one tool useful, 11 (24%) found all but two tools useful. Table 5.3.1 summarizes the responses on this question. The two kinds of tools most frequently selected as useful were `VCS-diffs` and `structural-navigation`. The least frequently selected tool category was `complex-refactoring`.

Table 5.5: The Tool column lists the 9 tool types that were asked about in the study. Count and Percent indicate the number and percent of the developers who responded that they find that kind of tool useful during software change tasks.

Tool	Count	Percent
`VCS-diffs`	42	91%
`structural-navigation`	42	91%
`simple-refactoring`	41	89%
`text-search`	37	80%
`debug-tool`	37	80%
`copy/cut/paste`	37	80%
`compiler-output`	36	78%
`test-suite`	34	74%
`complex-refactoring`	31	67%

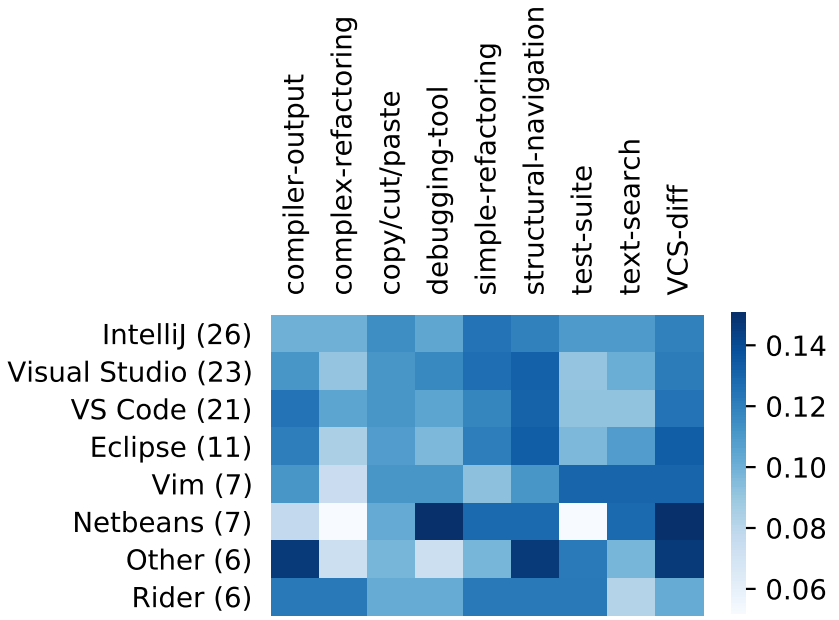


Figure 5.5: For any cell associated with an IDE (y-axis) and tool (x-axis), the color indicates the relative importance of that tool for users of that IDE. Darker colors mean that a higher fraction of the IDE users found that tool useful. For example, `simple-refactoring` tools were considered useful more often than a `test-suite` by the users of IntelliJ, Visual Studio, VS Code, Eclipse, and NetBeans. Meanwhile, the inverse was true for users of Vim and Other.

The participants in the study use a variety of IDEs. While many of these IDEs provide access to all the tools that are listed here, there are some differences between IDEs, such as the number of tools offered by that IDE and the degree of support offered by those tools. To investigate how the IDEs used by a developer affects the tools that they indicated as useful, I separated the responses based on the IDEs that the developers used.

Figure 5.5 provides a heatmap showing the correlation between the IDEs that a developer used and the tools they perceived as useful. Each cell's color indicates the fraction of users of the IDE who indicated the tool as useful in general software change tasks. The darker a cell, the more developers who use a particular IDE who find the tool useful. The most popular tools, `VCS-diffs` and `structural-navigation` are considered useful across most IDEs, while less popular tools like `complex-refactoring` are surprisingly popular amongst developers using Rider, IntelliJ, and VS Code. Less surprisingly, `simple-refactoring` tools are considered useful more often than `complex-refactoring` across all groups. The IDE row labeled Other represents largely simple text-editor-like environments like Atom and Sublime. It is not surprising that the users of these IDEs considered `complex-refactoring` and `debug-tool` tools to be useful less often than other groups, considering that these IDEs often do not have such support.

5.3.2 RQ2. *Which usability factors—efficiency, trust, effectiveness, predictability, satisfaction—do developers deem as necessary for a tool?*

Figure 5.6 shows which usability factors were important to developers overall (bottom) and per tool. Developers considered *efficiency* and *effectiveness* as the most necessary attributes for a tool, followed by *trustworthiness* and *predictability*. The least important factor was *satisfaction*.

Figure 5.6 also shows that the attributes considered necessary by developers were not evenly distributed between tools. Developers found *trustworthiness* to be the most important factor for `VCS-diffs`, `test-suite`, `debug-tool`, and `compiler-output`, while *efficiency* was most important for `complex-refactoring`, `copy/cut/paste`, `text-search`, `simple-refactoring`, `test-suite`, and `structural-navigation`.

Considering the practitioner statements, this indicates that for `VCS-diffs`, `test-suite`, `debug-tool` and `compiler-output`, it is most important that the tool feels safe and reliable to use, for example, by rarely or never making errors or mistakes. For the other tools, it is most important to the respondents that the tools are reasonable in terms of time and effort required to use and less important that they never make errors or mistakes. This was also true for `test-suite`. It is surprising that both kinds of refactoring tools are in this category where the most important factor is *effectiveness*, considering the high importance that previous work places on these tools' correctness.

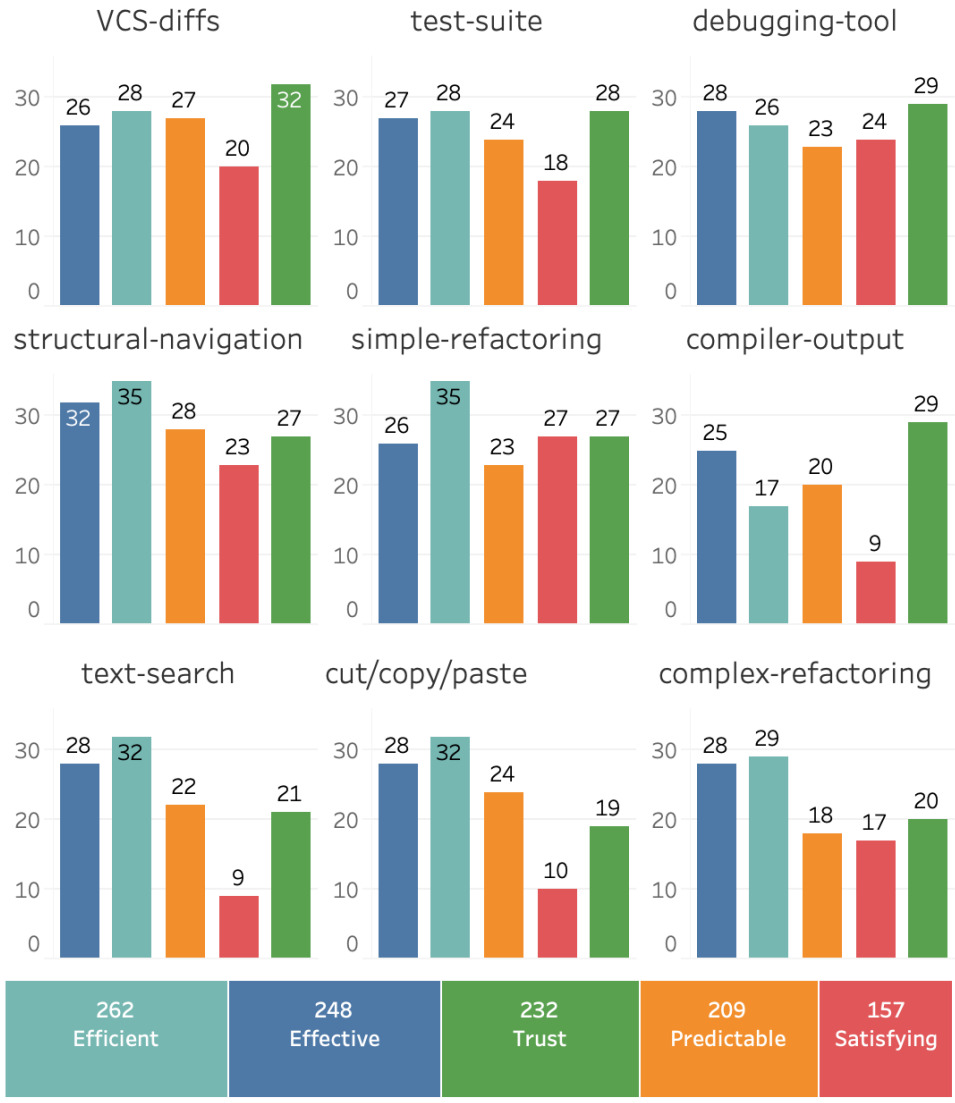


Figure 5.6: For each tool, the necessary tool attributes as selected by developers, with the total sum of each factor shown at the bottom and identified by a short keyword.

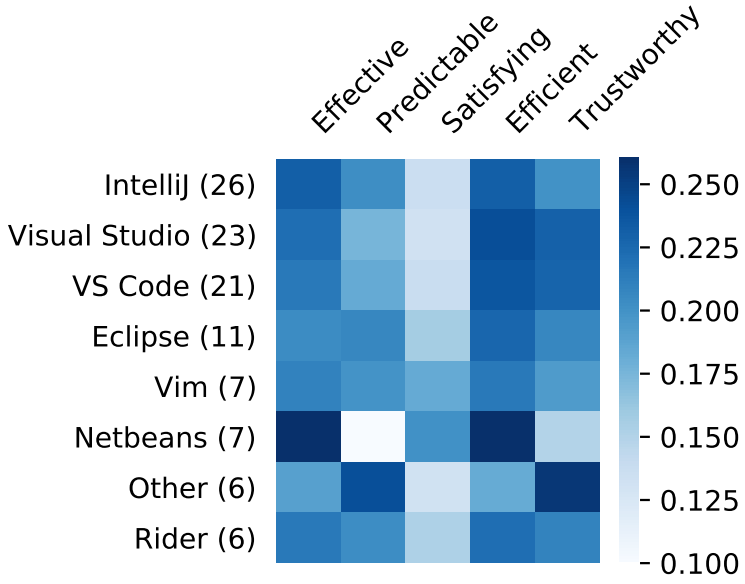


Figure 5.7: Each row indicates, for the set of users of that IDE (number of users shown in parentheses), how these users prioritized usability factors (x-axis).

I also consider which factors are least important. While *satisfaction* is the least important across most tools, there is significant variety between individual tools. For `text-search`, `compiler-output`, and `copy/cut/paste`, only around 20% of developers indicated a need for these tools to be satisfying to use in order for them to be useful. This is a markedly lower need for *satisfaction* than the other tools. `debug-tool` and `simple-refactoring` are the only tools that do not have *satisfaction* ranked the lowest; for these two tools, *predictable* is the least important factor.

It could be that developers who use certain IDEs are more likely to prioritize certain types of usability factors, so I investigated how the responses were distributed across IDE users. Figure 5.7 shows the trends in how different groups of IDE users prioritized usability factors. The middle column indicates that developers do not find it necessary for tools in most of the IDEs to be *satisfying* to use. Instead, the developers express the need for *effective* and *efficient* tools across all IDEs. Interestingly, users of NetBeans and the not-so-modern IDE, Vim, do consider it more necessary than others that tools are *satisfying*.

Table 5.6: The number of developers in each scenario (total count in parentheses) who selected each of the four approaches.

Scenario	Approach			
	Local	**Structure**	**Execute**	***other***
Scenario A (23)	13	7	2	1
Scenario B (26)	7	13	4	2
Scenario C (20)	2	8	6	4

5.3.3 RQ3. *Do change tasks impact developers' experience of usability factors?*

Figure 5.8 depicts the wide variety of tools used across the three scenarios. The most consistently used tools across the three scenarios were `test-suite` and `compiler-output`.

For each scenario, a developer could select one of three provided approaches to solving the task or describe another approach that they would take. Table 5.6 shows how many developers responded to each scenario and the approaches they selected. Developers largely selected one of the approaches presented to them, and for scenario A and B, a dominant approach emerged. Only a small number of participants selected ***Other***.

With this background, it is possible to consider whether the consideration of tools in the context of concrete change scenarios impacts the experience of developers with tools. I employed the method outlined in Chapter 5.2 to collect usability factors that were relevant to each scenario and show the results in Figure 5.9. The importance of usability factors remains relatively constant across scenarios, as seen by comparing rows of the figure. Scenario B shows a slight decrease in the importance of *predictability* and *efficiency* and increase in the importance of *effectiveness*.

For the same data (which factors that developers had associated with tools they opted to use), I tried grouping developers by approach rather than scenarios. Figure 5.10 shows the results. Developers using the **Execute** strategy seemed to prioritize *effectiveness* in the tools they used, and cared the least for *trust*. Developers who used the **Local** approach prioritized tools that they had labeled with *trust*, while developers using the **Structure** approach prioritized tools that were *efficient*.

As described in Chapter 5.2, each scenario contained steps that were amenable to automation with certain refactoring tools (Table 5.4). Most participants did not select to use a refactoring tool: 17 (74%) of the 23 participants in Scenario A, 21 (81%) participants of 26 in Scenario B, and 15 (75%) participants of the 20 respondents in Scenario C did not select the refactoring tool that was associated with that task. These participants were prompted to select the usability factors that contributed to the tool decision, as shown in Figure 5.4. Figure 5.11 shows the

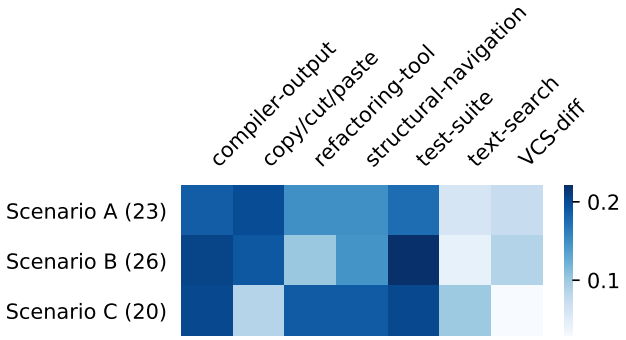


Figure 5.8: For each scenario (y-axis), the color indicates the fraction of developers who responded to that scenario (total in paranthesis) and indicated that they would use that tool (x-axis) to solve it. Tools with darker colors were selected more often.

distribution of responses. The variation in reasons indicates that tasks affected how tools were viewed. For Scenario B, developers were mostly concerned that the refactoring tool would not be sufficiently *efficient*, while this was not a concern in Scenario C. Across all scenarios, developers worried about the *trustworthiness* of the tools. The perspectives of the developers can help identify to toolmakers where the tools are deficient and perhaps preventing use. It would likely surprise toolmakers building safe refactoring tools that their tools are not trusted.

5.4 Discussion

Here, I discuss the results and their implications for toolmakers, potential future work and threats to validity.

5.4.1 Usability Profiles of Tools

The findings show it is possible to develop usability factor 'profiles' for tools, and that these profiles vary between tool types. The radar plots in Figure 5.12 show the profiles that developers found necessary for a few of the tool types that we investigated.

The shared terminology provided by usability factor profiles enables comparing developer experiences across tool types. In this way, usability factors might serve as a common framework for evaluating developers' experiences.

This can complement studies of tools in isolation: developer experiences can be investigated both on tasks with tool-specific subgoals, and on tasks with software change goals, and experiences can be compared to evaluate whether experiences worsen when the goals become more

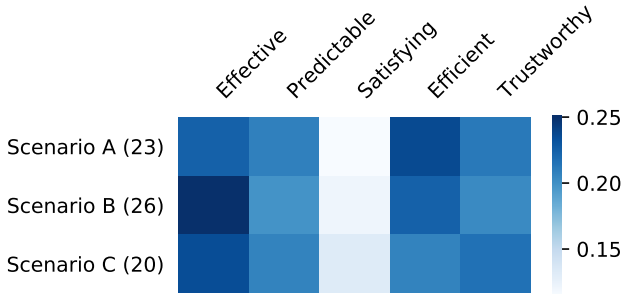


Figure 5.9: For each scenario (y-axis), the color indicates how frequently the developer selected a tool in that scenario that they had previously indicated the usability factor (x-axis) to be necessary for in order for that tool to be useful. E.g., while the 23 respondents in A most often selected tools that they considered *efficient* as necessary for, the 26 respondents in B most often selected tools that they considered *effective* as necessary for.

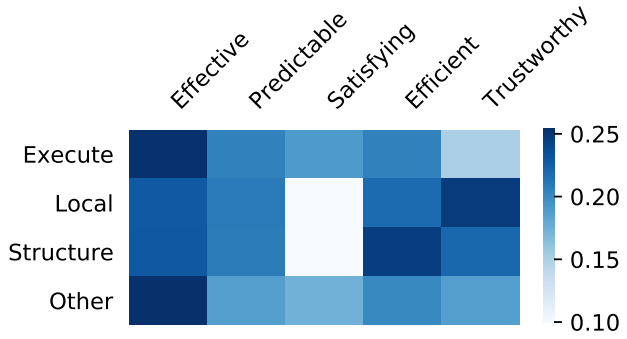


Figure 5.10: For each approach (y-axis), the color indicates how frequently a developer selected to use a tool that they had previously indicated that the usability factor (x-axis) was necessary for it to be useful.

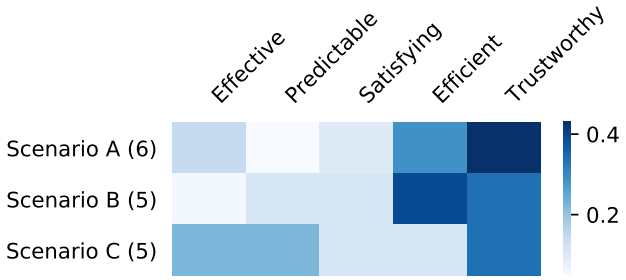


Figure 5.11: For each scenario, the developers who *did not select* the appropriate refactoring tool on that task were asked to select one or more factors that impacted their choice to not use that tool. The total number of participants who both did not select the right tool and provided one or more factors is indicated in paranthesis. A darker color in a cell indicates that a factor had a high importance for those developers' choice in that scenario.

general. Future research should investigate a wider range of usability factors, for example, by comparing these usability factors with the dimensions from cognitive ergonomics [54].

When comparing responses between different IDE user groups, we saw that the different groups placed different importance on individual factors. We looked into one of the most popular tools, `VCS-diffs`, to see how users of the two most popular IDEs (IntelliJ and Visual Studio) considered this tool. Figure 5.13 shows that there is a difference: for this specific tool, users of Visual Studio considered *trustworthiness* more important than users of IntelliJ. Perhaps this insight could be useful to toolmakers who can better align tool implementations with the expectations of users of their particular IDE.

Usability profiles for tools could help support toolmakers in making design decisions. Refactoring tools cannot always both meet developers' desires (e.g., *satisfaction*) and be fully correct (e.g., *trustworthy*) [24]. Toolmakers who are designing refactoring tools need help addressing these factors and determining how to make tradeoffs in the tool design. While other researchers have explored individual factors (Chapter 2.6), no other work has, to my knowledge, investigated how developers experience the importance of individual factors, nor whether or how this importance varies between developers, tools, and change tasks.

The profiles found in this survey indicate that developers consider it equally important that refactoring tools are *satisfying* and *trustworthy*, meaning that developers can tolerate both less safety and less value from a tool, but the most important aspect of a tool is that it is reasonable in terms of time and effort to locate and apply.

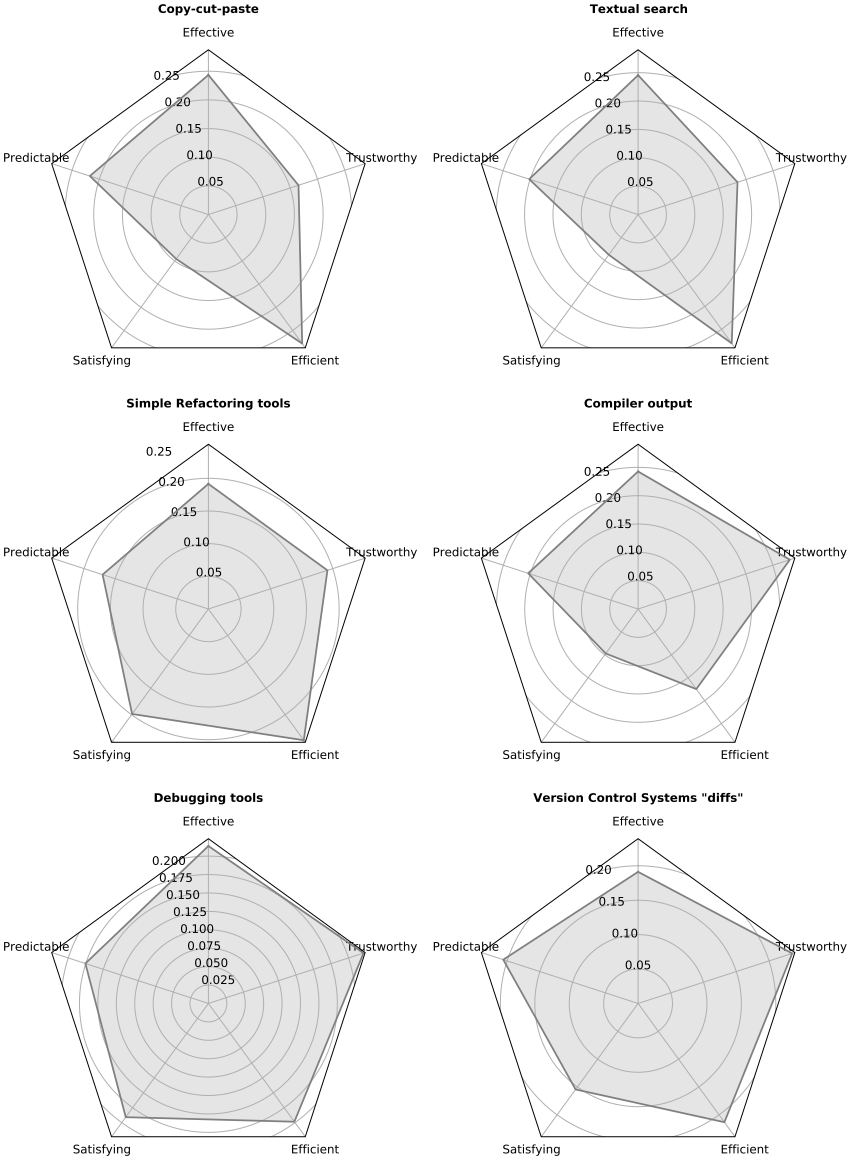


Figure 5.12: Radar plot of the tool attributes that are necessary for each tool's usefulness.

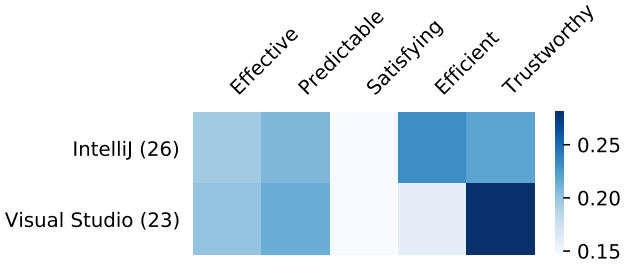


Figure 5.13: This figure shows the importance that the users of each IDE placed on individual usability factors for only a single tool: `VCS-diffs`.

5.4.2 Developer Strategies and Preferred Usability Profiles

The findings show that developers who use different approaches to software change tasks select tools that exhibit different usability factors. Figure 5.14 shows the differences between two approaches. Participants who used an **Execute** approach prioritized *satisfaction* and *effectiveness* higher than participants who used a **Local** approach who prioritized *efficiency* and *trustworthiness* more.

It is possible that this might be related to the different strategies used by developers. For example, the **Local** strategy is oriented around "breaking" and "fixing" the code by intentionally introducing compiler errors and using these to navigate and edit other locations in the code, whereas the **Execute** strategy is oriented around making carefully orchestrated code edits that allows the tests to keep running so that they can be used to verify code changes. It could be that in the **Execute** case, because developers are planning more than in the **Local** case, their tool use is oriented around tools that are *effective*—ones that are reasonable to locate and apply for a desired intent (Table 5.3). Meanwhile, the developers who employ the **Local** strategy seem to make more short-term plans and to largely act on information that is immediately available to them. These developers might be more reliant on their tools to be *trustworthy* because they rely on tool feedback to form their actions to a larger degree than in the **Execute** case, effectively delegating more of the planning responsibility to the tools, such as determining the impact of edits to the code. If it was possible to identify the strategy being used by a developer, this information might be used to improve recommendation techniques to recommend tools by matching the tool profiles to the strategy in use.

One could even imagine developing distinct tool profiles that can be accessed by recommendation techniques or preferences. To continue the example with refactoring tools, users might be allowed to toggle between various tool profiles so that developers in need of highly safe tools can enable maximal safety checks and warnings, while developers

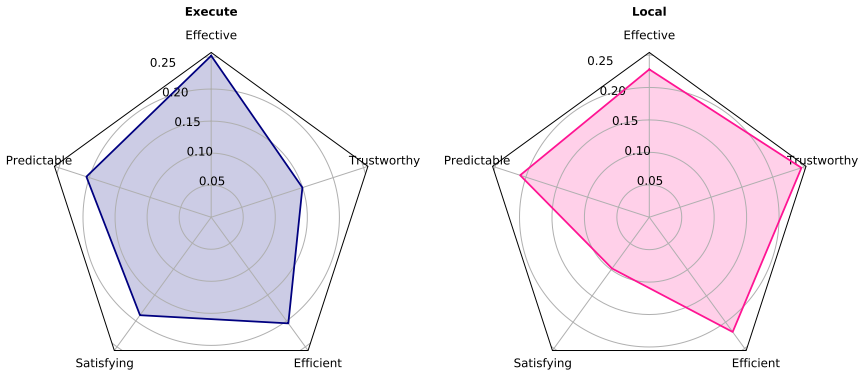


Figure 5.14: Radar plot of the usability factors that were considered necessary by participants using the two different approaches.

who need more time-saving tools but care less about correctness, can alter the interaction and modes of the same tool. In this way, profiles can be used to develop the design of tools that work for a broader range of tasks and developers.

5.4.3 Threats to Validity

Software developers comprise a large and varied demographic. There is always a threat that the limited responses to a survey may not be representative of the general developer demographic. I mitigated this threat by distributing the survey internationally through different channels (both social media and industry contacts) and by filtering participants that did not have qualifying programming experience. The use of a survey, compared to other instruments, such as a laboratory or field study, lets us reach a broader range of participants and is compatible with restrictions on personal meetings as per 2020-2021 (due to COVID-19). Overall, the demographic data of the participants are comparable to the demographic data for the "Professional Developers" category of participants in the Stack OverFlow survey [10].

A risk of using an online survey to learn about software change tasks and refactoring tool use is that participants respond based on what they *think*, and not what they *do*. Another risk is that participants may have difficulty interpreting the questions (Chapter 2.4). To mitigate these risks, I provided, and repeated, terminology descriptions throughout the study. I also included concrete scenarios for participants to consider in addition to the general questions. This use of both general questions and concrete scenarios made the survey longer, which caused some participants to abandon the survey. I chose to accept the increased length and loss of participants in order to allow for the collection more

fine-grained information.

Some potential participants commented on social media platforms that the survey was cognitively challenging as it asked them to consider how they would solve code scenarios. The code scenarios might have deterred participants from joining the survey or caused them to leave. I consider this an acceptable risk because the scenarios are important to disambiguate the participants' responses.

5.5 Summary

This chapter provided insights into developers' experience of the usefulness and usability of nine types of tools that are used to approach software change tasks. In addition to providing data about which tools developers find useful in general for change tasks, the survey results indicate which usability factors developers find necessary in various types of tools. As one example, developers found *trust* to be the most important factor for `VCS-diffs` and `compiler-output` while *efficiency* was the most important factor for `refactoring-tool`. By studying the tools developers found necessary for three specific change scenarios, I was able to learn about which usability factors may be important in tools to support different strategies used by developers. For example, developers who seek to keep tests running as they make changes (the **Execute** strategy), prioritize *effectiveness* in the tools they choose to use. I demonstrated that information about usability factor profiles for tools can help guide toolmakers to understand potential improvements to tools and can help guide toolmakers in design decisions.

Chapter 6

Stepwise Refactoring Tools

In the previous chapters, I have presented my investigation into how developers approach software change tasks that are amenable to the use of refactoring tools, and how they experience current refactoring tools during such tasks (Chapter 3). I found that developers experienced usability problems with these tools, and that these usability problems were associated with the different strategies that developers used to progress in the tasks. Although refactoring tools seemed well aligned with the **Structure** strategy, developers who employed one of the two other strategies found that the usage of refactoring tools hindered subsequent steps in their workflows. Based on these observations and my operationalization of a standard definition of usability, I identified four usability themes that are important for developers when they use refactoring tools during functional software change tasks (Chapter 4). I also identified some tradeoffs in the design of refactoring tools that could improve developers' experience of usability in these tools (Chapter 4-5).

In this chapter, I act on these insights by proposing a novel stepwise interaction model for refactoring tools. This model addresses several of the usability problems that were identified in this dissertation and in previous research. I call a tool that implements this interaction model a *stepwise refactoring tool*.

Part of this chapter is published elsewhere [39].

6.1 Introduction

Refactoring occurs frequently during software change tasks, even when the software change is functional (Chapter 2). A developer who receives a change request will typically realize it by identifying concepts related to the request, searching for the concepts in

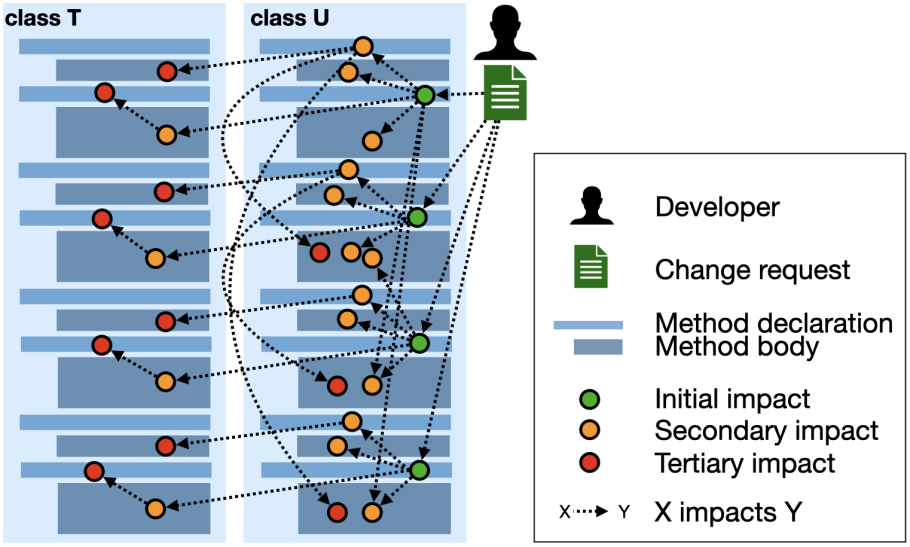


Figure 6.1: This figure illustrates the rippling changes that can result from software change tasks. A change request initially impacts 4 areas (green). Editing those areas impacts 17 areas (orange), which in turn impacts 11 new areas (red).

source code, and then editing the identified code [99]. The initially identified code is seldom the only code that needs to change. As illustrated in Figure 6.1, developers must also identify additional rippling changes [131] that are necessary to restore consistency and complete the change [53]. This identification of rippling changes is known as *change impact analysis* [21] and typically involves the analysis of dependent source code (e.g., via references, callers, etc.) [56, 27, 102]. When performed manually, change impact analysis is time-consuming and error-prone. However, often the needed change impact analysis, and subsequent editing of code (e.g., change propagation [26]), aligns with refactoring tools [35, 123, 47, 86, 112] found in modern IDEs. In Chapter 2.1, I described how developers can use these tools to automatically change source code, and how the tools can locate and update impacted areas of the program. In theory, developers can reduce their manual workload by using these tools to automatically perform subsets of their code edits. However, in practice, multiple investigations have shown that developers eschew refactoring tools in favor of editing the code manually (Chapter 2-3).

In Chapter 3, I identified three strategies that developers use to progress in software change tasks: a **Local** strategy, in which they organize their changes based on the information that is locally available to them, such as compiler errors; a **Structure** strategy, in which they organize their changes based on program structure such as references and callers; and, an **Execute** strategy, in which they organize their changes based on the impact that the changes will have on the program's execution. In the study, I observed

that when a participant employed a **Structure** strategy, they were better able to utilize refactoring tools than when they employed one of the other two strategies. With the other two strategies they more often had experiences where the refactoring tool hindered subsequent steps in their workflow. Moreover, the usability problems that participants encountered, and the ways in which they experienced these problems, also varied between the groups that employed each strategy.

This hints at a misalignment between the workflow that refactoring tools support and the workflows that developers employ. There is a potential for gains in usability by adjusting the interaction model of the refactoring tools to better accommodate the developers' workflow.

In the remainder of this chapter, I describe and discuss a novel interaction model for a stepwise refactoring tool. As a motivating example, I introduce a simplified version of Task-3 from the laboratory study presented in Chapter 3 and use three developer personas, Alice, Bob, and Charlie, to illustrate the different kinds of experiences that participants from the study had with the tools that were available to them. Then, I describe a stepwise refactoring tool and show that such a tool is better aligned with the workflows that each of the three personas use. I then compare the stepwise refactoring tool to the four usability themes described in Chapter 4 and show that it addresses all four themes. Then, I discuss some implementation challenges and compare it to existing tools.

6.2 Motivating Example

In Chapter 3, I described the design of three software change tasks that contained steps amenable to automation with refactoring tools. Task-3 was the most complex task and contained rippling changes like the ones illustrated in Figure 6.1. The task required participants to prevent a certain functionality from being accessed by users of a class and was based on a refactoring mined [112] from an existing system implementation [14]. During the lab study, six of the participants verified the tasks realism by making statements like "This is just like my day job" (P_2) and "I can see [this] coming up [at work]" (P_{11}). Despite the small size of the initial change and the simplicity of the changes (code removal), the task can quickly become overwhelming. Here, I present a simplified version of this task and three distinct approaches that could be used to solve it. Although the example might seem relatively straightforward, Task-3 required 29 methods to undergo one or more changes each. The approaches are illustrated by three developer personas, Alice, Bob and Charlie, that are based on the participants in the laboratory study and

the workflows that they used to solve Task-3.

Consider a software change task involving two Java classes, a utility class U, and its test-class, T. Class U includes 8 pairs of methods to read from, and write to, objects of some datatype, D. In each pair of methods, one method provides a default behavior and the second method takes an extra flag argument that lets callers toggle a special behavior. The first method is a wrapper to the second method, calling the second method with the extra flag argument set to a particular value. The code in (6.1) provides a simplified example of such a pair of methods and Figure 6.1 illustrates four such pairs and some of the rippling changes in the task.

```
D m(D d){return m(d, false);}
D m(D d, boolean toggle){ ...  }
```
(6.1)

The test class contains one or more JUnit tests for each of the methods in U. For the change task, the developer was to remove the special behaviour from U, retaining only the default behavior for each pair of methods. To preserve code quality, in each pair, the special parameter (e.g., `toggle` in (6.1)) should also be removed.

To complete this task across U and T, the developer was to perform the following editing pattern, for each pair of methods: 1) remove the parameter from the method signature, 2) update its callers, 3) handle the wrapper method, 4) remove the special behavior, and 5) update the corresponding tests.

Now, I consider the three different strategies that developers might use to approach solving this task, illustrated by the developer personas Alice, Bob, and Charlie.

6.2.1 Bob and the "Break-and-Fix" Approach

Bob usually organizes his workflow around introducing and fixing compiler errors. He begins the task by deleting the first `toggle` parameter. This results in a number of compiler errors: one indicates that the new method signature collides with the wrapper, another indicates a broken reference to the parameter in the method body, and several more indicate errors in both in-class and test-callers to both methods.

He fixes the errors one by one in random order. However, he finds that fixing one error often introduces multiple new ones and quickly loses track of which changes caused which errors. When he encounters an error that he does not understand, he can only assume that he has made a mistake somewhere. He restarts the task by reverting the entire codebase.

This time, he has a better idea of the repeating change pattern. He first steps through the file from top to bottom and removes the flag parameter from all eight methods. Next, he solves all the introduced compiler errors in the class, either by removing the wrapper or updating the unsafe behavior. Once class U compiles, he moves on to the tests in T. There, the compiler errors lets him navigate to impacted locations and the flag argument (`true` or `false`) helps indicate which behavior the tests are expecting so he can adjust them accordingly.

6.2.2 Alice and the Awkward Automated Approach

Alice prefers relying on automated tools. She recognizes that the removal of a parameter is supported by the `change-signature` refactoring tool and invokes and configures the tool to remove the parameter on the first method pair. The tool presents two problems: "the parameter is used in the method body" and "the method signature is already defined" (because of the overloading). She cancels the refactoring to locate and resolve the problems. Then she invokes and configures `change-signature` again. This time, the tool presents a Preview View listing the callers that will be changed: nineteen callers in the tests and three in-class callers. That seems reasonable, so she clicks "Do Refactor" and the parameter disappears. She repeats this workflow for each of the remaining methods: remove parameter usages, remove the wrapper method, invoke and configure `change-signature`, preview, and apply.

After completing the changes in class U, she runs the test code. Many tests fail. After debugging them for a while, she realizes that they might expect the special behavior. However, she cannot be certain because the refactoring tool removed the flag argument from the test callers too. The previous meaning of the test code is obscured by the refactoring and she cannot recover it without reverting her entire change.

6.2.3 Charlie and the Controlled Approach

Charlie prefers to work systematically and to control the impact of changes that he makes. He first locates the code that represents the special functionality and identifies that the parameter must be removed. He opens the refactoring menu and considers both `safe-delete` and `change-signature`. The `safe-delete` can delete the parameter but Charlie wonders how that action would impact the code. Could it be that it will delete the callers too? He is afraid to try it. Instead, he invokes `change-signature` and configures it to remove the parameter. The tool presents him with the same two problems that Alice's invocation did but Charlie continues to the Preview View (or Conflict View)

and inspects the callers. In this view, he notices some test-callers and some in-class callers including the wrapper method. That all makes sense so he continues and applies the refactoring. The resulting code has a couple of compiler errors that he resolves by removing the wrapper method and the code that represents the special behavior. Then, he opens the test class. Here, he is puzzled over the lack of compiler errors. When he tries to locate the tests that relate to the special behavior, he realizes that they are difficult to tell apart from the ones that test the default behavior, since both types of tests call the same method without any argument. The argument was removed by the `change-signature`, a correct but unexpected impact of the refactoring. He reverts the code using his git tool.

Next, Charlie considers how to best solve this. He wants to understand what the impact is of the functional change, and to do so, he wants to locate the impacted tests. But he also noticed that some of the methods were dependent on one another and worries that this change might get messy. To avoid becoming overwhelmed, he decides to first edit the leaf nodes of the call graph and then work his way up the root nodes. He uses the Find Usages tool to locate a method with special behavior that is only called by test callers and its wrapper method—a leaf node. Then, he adds a single line of code to the method and assigns the value `false` to the `toggle` parameter:

```
D m(D d, boolean toggle){toggle = false; ...  }
```
(6.2)

This means that regardless of which argument value that is passed in (`true` or `false`), the method will always produce the default behavior. He runs the tests, and only the tests for that method fail. They are easy to locate as the JUnit test suite is highlighting them. He inspects these tests and decides that they can be removed. Then, he applies the Find Usages tool again to verify that there are no more usages. This time he finds only the usage in the wrapper method so he re-runs the tests and all pass. He does not edit the wrapper method or the parameter at this point, but instead moves on to the next method and repeats this pattern. Eventually, he completes the changes in the test class in this way.

At this point, he considers himself finished with the functionality change. However, he should still clean up the unnecessary parameters that are always false, the wrapper methods, and the unnecessary special functionality code. He begins on the first method and replaces all references to the parameter toggle with the value `false` inside that method. That helps him ensure that the code is, in fact, *dead code*, that is, it will never be run. He removes that code. Then, he invokes `inline-method` on the *wrapper* method. This leads to all the locations that previously called the wrapper without the

extra argument, to now make the call *with* the extra argument `false`. Then, he applies the `remove-parameter` refactoring to update all the callers to that method and runs the tests. They all pass. He intends to repeat this pattern for each of the 8 methods but quickly realizes that he might as well do it manually since the number of callers are very low. He completes the task manually.

6.2.4 Relevance to Developer Participants

The three personas are based on participants in the laboratory study that I described in Chapter 3. The participants, their workflows, and the problems that they encountered are described in detail in Chapter 3 and the usability problems that they encountered are analyzed further in Chapter 4. Here, I briefly describe the relationship between the personas and the laboratory study participants.

Bob is based on the participants who employed the **Local** strategy to solve Task-3 and who approached the task without using refactoring tools. A typical participant in this group is P_2, who experienced the same problem that Bob did and restarted the task as a result. Other typical participants from this group are P_6, P_8, P_{12}, P_{13}, P_{15}, and P_{19}. These participants expressed their preference for introducing and fixing problems rather than planning their changes upfront. While they were able to use the compiler errors to evaluate the impact of the change and to navigate the code, this workflow often meant that by the time they were able to gain an overview of the task and what code changes had to be performed, they had already made many changes. Consequently, these participants often lost control of their changes which led them to either haphazardly continue the task until "there were no more errors" and ask if they had completed the task (P_2, P_{15}, P_{19}), or to make mistakes that took them a long time to fix (P_{15}), or to restart the task using git (P_2, P_6, P_{15}). One exception to this was P_{12} who took great care to organize their changes such that the compiler feedback could be introduced and resolved in a disciplined manner.

These participants often avoided using refactoring tools. They expressed that they trust the compiler more than other tools and appreciated its immediate feedback. They also said that using a tool would make them lose control of the change and prevent them from performing the change stepwise and from learning about the code that they changed. Even when they were aware of refactoring tools, they expressed that they did not trust the tool, they did not believe that it would help them, and they were unable to apply it correctly. In some cases, these participants realized too late that they could use a refactoring tool. For example, many of these participants began a task by locating the code element that should be changed and, if the changed required deleting code, they

promptly made the deletion. This led to compiler errors which they used to continue their work on the task. However, this workflow blocked them from invoking refactoring tools like `inline-method` and `change-signature`, as was illustrated in Chapter 5, Figure 3.8.

Alice is based on participants who used a **Local** or **Structure** strategy and who employed refactoring tools and experienced usability problems with the tool's impact on the tests. The participants that are represented by this persona are P_4, P_5, P_7, and P_{14}. These participants solved some or all of the task with the help of refactoring tools but complicated subsequent steps in their workflow by doing so. Alice's workflow is similar to that of P_7 and of P_5 who employed a different refactoring tool. In addition to representing these participants, Alice illustrates usability problems that are relevant to more participants. For example, P_{15} described the problem that Alice encountered when asked why they avoided using refactoring tools (Chapter 3.4), and P_8 encountered the same problem upon their first tool invocation and then decided to revert to a strategy focused on compiler errors.

Charlie is based on the participants that used the **Execute** strategy in the laboratory study, that is, P_3, P_9, P_{10}, P_{11}, P_{16}, and P_{18}. They employed a similar sequence of changes to that described here, although only P_9, P_{10}, and P_{16} used refactoring tools in the same way that Charlie did. The other participants expressed that they did not believe the tool would help them much, they did not want to spend time on figuring out how to use the tool, and they wanted to remain in control of the change. These participants wanted to maintain their *mental model* of their progress towards the task goal and to ensure that the code changes that they made were of high quality. Consequently, it was not only important for them to update all the impacted locations of the code, but also, to update them in the right *order*. The refactoring tools does not afford control over this, which made it more difficult for these participants to benefit from tools. The *cost* of using refactoring tools was higher for these participants. The overall workflow that Charlie used is similar to that of P_9 and P_3 and the usability problems that he encountered was also inspired by P_{10}.

Taken together, the three personas capture workflows and experiences from all of the developers that participated in the laboratory study described in Chapter 3. Table 6.1 summarizes relevant information about the participants and how they map to the developer personas.

Table 6.1: An overview of the mapping between the three developer personas and the participants that they represent. Each row represents one participant. The coloring indicates which persona that participant is represented by; this information is also presented in the first column. For each participant, the table also includes the number of years of experience in the software development industry that participant had. It also includes which refactoring tools they used on Task-3 in the laboratory study and which strategy they were observed to use on that task. The example task that the three personas solved in this chapter was based on Task-3.

Persona	P_x	Experience (years)	Task-3	
			Refactoring Tools	Strategy
Bob	P_2	20+		Local
Charlie	P_3	10		Execute
Alice	P_4	6	ChS	Structure
Alice	P_5	2	SD	Local
Bob	P_6	5		Local
Alice	P_7	8	IM, ChS	Structure
Bob	P_8	20+	ChS	Local
Charlie	P_9	10	ChS, IM	Execute
Charlie	P_{10}	10		Execute
Charlie	P_{11}	16		Execute
Bob	P_{12}	11		Local
Bob	P_{13}	1		Local
Alice	P_{14}	20+	IM, ChS	Structure
Bob	P_{15}	2		Local
Charlie	P_{16}	10	ChS	Execute
Charlie	P_{18}	2		Execute
Bob	P_{19}	12	ChS	Local

IM = Inline Method ChS = Change Signature SD = Safe Delete

6.2.5 Usability Problems

All three personas faced problems with the tools they used. The refactoring tool chosen by Alice and Charlie helped reduce some of the code's *viscosity* (i.e., the code's resistance to change [53]) by automatically propagating changes. But, the use of the tool also reduced the *visibility* of information [53] by prematurely hiding the meaning-carrying argument values in the test class. The compiler tool that Bob used helped increase the *visibility* of information as he could see the impacts of individual changes, but did not help with code *viscosity*.

In Chapter 2.1, I described how mainstream refactoring tools force the developer into an all-or-nothing situation where they either must accept that the tool makes all the changes at once or "return" to the source code as it was before the invocation. Alice experienced this twice. The first time, she discarded her configured refactoring in order to inspect and resolve the problems that was listed in the Problem View. All configuration steps had to be repeated the next time she invoked the tool. The second time, she applied the refactoring and in doing so, she lost control over the individual steps that were listed in the preview view. Afterwards, she could only revert the entire code change or manually change the code. Charlie encountered the same problem, but discovered it early and decided to revert the entire change.

Charlie encountered a usability problem with the `safe-delete` tool, namely an inability to *predict* the outcome of applying the tool. He also encountered the usability problem where the `change-signature` refactoring reduced the visibility of information in the test callers, but his workflow led him to quickly recognize it and revert the application.

Note that the refactoring tool executed the refactoring correctly and, in fact, even presented Alice and Charlie with the test callers in the Preview View. The developers, however, experienced *premature commitment* [53], the need to make a decision before the necessary information was available. At the time, they did not know how what the subsequent steps of their workflow would be, and as a result, they applied a transformation that made these steps more difficult. Despite the Preview View listing the impacted locations, the developers were unable to predict the implications of the changes at that time. If the tool instead had offered a more *provisional* [53] change that could be altered later or postponed, they could have deferred their decisions to an appropriate time.

Since Charlie organized his workflow into two separate steps, he had more opportunities to use refactoring tools than Alice and Bob did. His first step was to alter the code's functionality and update the test code and the second was to remove the code that was now unnecessary, such as the parameter and the wrapper method. The second step

seems like an ideal use case for refactoring tools. However, despite the refactoring tool's ability to support the second step, Charlie still had to perform repetitive navigation, invocation, and configuration of the tool. Consider that during Charlie's first step, he already navigated to all instances of the parameter that should be removed. Then, in the second step, he had to repeat the same navigation to invoke and configure the `change-signature` once for each parameter. The promise of the refactoring tool—to prevent error-prone and time-consuming navigation and editing—was not realized to a degree to made the automation "worth it". If he could have made a provisional change the first time he encountered these code elements, for example by indicating his intention to inline the wrapper methods and remove the parameters, and only apply those refactorings when he was ready, the refactoring tools could have saved him from a substantial amount of work.

Adding nuance to the all-or-nothing interaction with the refactoring tool could it more useful for Alice and Charlie, and even let Bob use them in a way that was aligned with his workflow.

6.3 A Stepwise Refactoring Tool

I believe that a refactoring tool that would allow a developer to step through a code transformation similarly to how a debugging tool enables stepping through a program execution could overcome Alice's, Bob's, and Charlie's tool challenges. I call such a tool a *stepwise refactoring tool*. A stepwise refactoring tool lets the developer transform the code in steps and enables editing of both the source code and the transformations in-between steps. This ability to interact with the code at each step could even entail invoking other refactorings, thereby enabling recursive refactoring [82]. The developer can choose which transformations to execute stepwise and which to batch-execute, similarly to how breakpoints afford control over program execution in a debugging tool.

This can be achieved by separating the code change *intent* from the code *transformations*. In an atomic refactoring tool, the developer performs one or more steps to define their refactoring intent (such as invoking the tool and configuring it), and the tool, in return, produces and applies code transformations, in a single step. This workflow relies on the source code remaining unchanged between invocation and application, as is enforced by the tool's interaction model, as described in Chapter 2.1. In contrast, for a stepwise refactoring tool to be useful and support all the three approaches that were illustrated by the three personas, the developer must be able to interleave the application and configuration of refactoring steps with code changes from other sources, such as manual

changes or even invoking other refactorings. In an atomic refactoring tool, attempts at editing the code midway through the transformation would render remaining steps invalid. I suggest instead to focus on capturing the *intent* of the change and in response, propose code changes that correspond to this intent. The developer must be allowed to perform a stepwise inspection and adjustment of each of these code changes before—and potentially after—they are applied. As the developer steps through the refactoring, the individual steps might even altered. The refactoring tool will change the source code in a stepwise manner by generating the code transformation upon the application of each, individual step.

To approach this refactoring tool design, I propose the following interaction model: a stepwise refactoring tool takes as input an *initial change* and identifies *impacts*, which are other code locations[1]. For each impacted location, if possible, the tool generates one or more *additional changes* that propagates the initial change to that location. The tool provides the developer with an interactive view that lets her inspect and alter the additional changes before applying them. The view also lets her interact with the source code without rendering subsequent steps in the refactoring invalid. The additional changes can be applied individually, in groups, or in batch, and are robust to the target source code being edited before, or between, applications. The view also presents *problems* or *warnings*. In this model, these are represented as impacts without any additional changes associated with them. The developer can navigate to these locations and edit them before applying the remaining steps. Note that if all the additional changes are applied using the Apply All button, the effect of this tool is similar to that of a traditional refactoring tool.

Figure 6.2 contrasts this workflow with the traditional refactoring tools described in Chapter 2.1. The topmost diagram depicts the workflow of mainstream refactoring tools, in which the developer must "leave" the code editing context (shown in blue) when invoking a tool and "return" either by applying an entire transformation (right green arrow) to the source code or by discarding the change (left red arrow). While the developer is interacting with the tool (gray box), they can inspect and edit the upcoming refactoring (i.e., navigate the black lines) but cannot interact with the source code. The tool can check for problems and inform the user about violated conditions (Problem View) and about the locations that will be impacted (Preview View); however, if the developer wants to change any of these locations, they must either complete or cancel the refactoring.

A developer who invokes a stepwise refactoring tool is supported in a sequence of changes

[1]Impacts can also be located in other artifacts such as documentation, build files, specification files, and so on. A more advanced tool could include such artifacts.

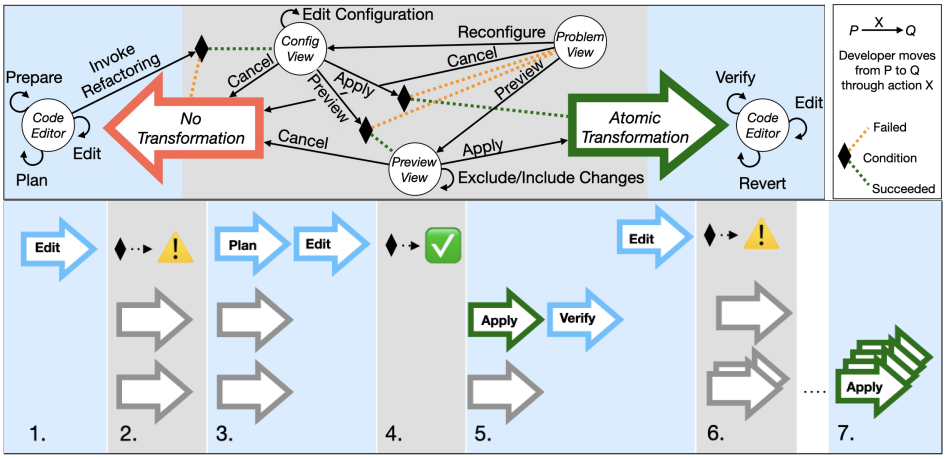


Figure 6.2: The topmost diagram depicts the workflow of most refactoring tools, in which the developer must "leave" the code editing context (blue area) to interact with the tool (gray area) and "return" either by applying an entire transformation (right green arrow) to the source code or by discarding the change (left red arrow). The diagram below illustrates how the tool allows the initiation, preparation, and application of individual refactoring steps (gray areas) with other interactions with the source code (blue areas).

that can be interleaved with other activities, as illustrated in the topmost diagram in Figure 6.2. The sequence begins with an initial edit (1) (blue arrow). The tool analyses the *initial change* (gray box) and detects any *impacts*, such as problems, indicated by the triangle icon (2). If possible, the tool generates *additional changes* (gray arrows) that can be inspected and applied. The developer can continue to work in their code editing context (blue box), and can inspect the code, plan changes, and even edit the code (3). The tool analyzes further edits but if it cannot detect impacts, it does nothing (4). The developer can choose to apply one of the additional changes at a later point (green arrow) and can immediately verify that the change did what they wanted by inspecting the code (blue arrow) (5). They can also continue editing the code (blue arrow) without applying the remaining additional change *yet* (gray arrow). If the tool does detect impacts it might generate more additional changes (6). This workflow can be repeated until the developer choose to apply all the remaining additional changes (7) or to discard them.

To explore the feasibility and implementation challenges of such a tool, I implemented a proof-of-concept stepwise refactoring tool for the `remove-parameter` refactoring as a plugin to IntelliJ. Figure 6.3 shows a screenshot of this tool. Here, I illustrate how this stepwise refactoring tool would work by revisiting the three workflows of Bob, Alice, and Charlie from the perspective of such a tool.

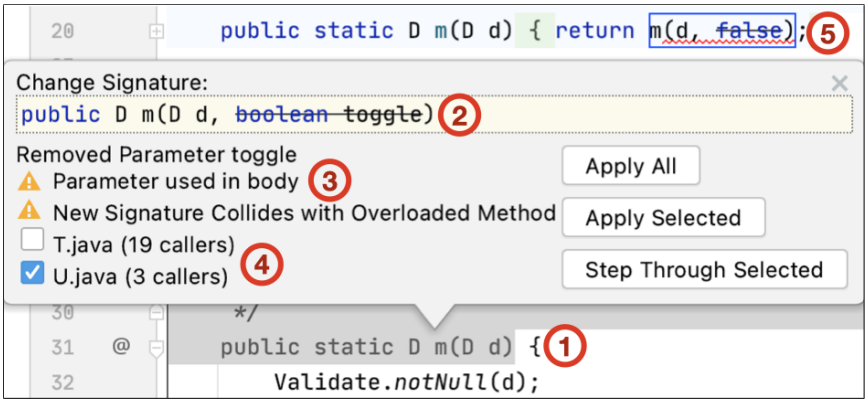


Figure 6.3: In response to a manual parameter-removal (1) a refactoring panel appears, indicating the initial change (2) and impacts (3-4). The tool suggests additional changes (4) that can be automatically applied stepwise or in batch, and can be previewed (5).

6.3.1 Bob and the Stepwise Approach

Bob began his workflow by manually deleting the parameter to see what the consequences would be. The stepwise refactoring tool automatically recognized the manual parameter-removal as an *initial change* for the `remove-parameter` refactoring. In response to that initial change, the view shown in Figure 6.3 appears and indicates to Bob what the tool considers to be the initial change (2). This feedback helps Bob understand which change the tool responded to (1). By naming the change, it also helps him consult mainstream sources (e.g., [47]) to build his understanding and trust of the tool.

The view also lists the impacts and additional changes in one Impact View (3 - 4), including the parameter usage and the overloaded method. This helps Bob avoid the disorientation he previously experienced when he relied on compiler errors. The Impact View lists all the locations that the compiler errors provided him with and lets him navigate to them by clicking on the various text-elements. However, in contrast to the compiler errors, this view groups the locations in a reasonable manner (e.g., by file) and indicates *why* the locations are impacted. For example, Bob might click the warning about the overloaded method and the IDE would navigate to the overloaded method on line 20. At that point, he can choose to textually delete the method, like in his previous workflow, and the Impact View would update to remove the warning about the signature collision. Instead, the view might present a new warning about the captured callers. The overloaded method had a number of callers in both U.java and in the tests: now these callers would bind to the refactored method instead. This information was previously not easily available in Bob's workflow but can be included in the Impact View. After removing the method, the stepwise refactoring tool lets Bob continue the refactoring

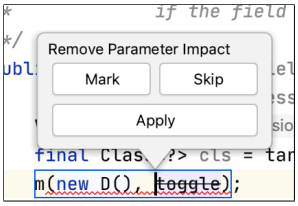


Figure 6.4: One step of the `remove-parameter` refactoring as executed with a stepwise refactoring tool. This view for a singular step (Step View) indicates where an additional change is going to occur, namely at the argument with the black line through that will be removed. The view lets the developer choose between marking the change and retain it for later (Mark), skipping the change and move to the next (Skip), or making the change right now (Apply) and actually remove the parameter. The developer can also click the information text on top (Remove Parameter Impact) to navigate back to the source of the initial change. They can also edit the code manually instead of interacting with the view. The red squiggle indicates the compiler error that originated from Bob's manual parameter removal.

without re-initiating it.

In addition to presenting information, the Impact View proposes additional changes that propagate the local change to the impacted location (5). Similarly to a traditional refactoring tool, Bob can choose to immediately Apply All. In that case, all the arguments will be removed. However, since the tool is stepwise, he can also step through some, or all, changes.

If Bob chooses to step through changes, a Step View presents information about the initial change that impacted that location and lets Bob navigate back to the original code, as shown in Figure 6.4. If Bob clicks on the information text "Remove Parameter Impact" in the view seen in Figure 6.4, he can navigate back to the method that the change originated from. In contrast, Bob's previous workflow hid the information about which change impacted which location, and the compiler errors might even have prevented the use of navigational tools like Go To Declaration.

Bob can invoke this refactoring by deleting the parameter in each of the eight methods with the special functionality in the class U.java. For each of the eight methods, he can use the Impact View to manage his changes and to automatically apply the in-class changes. He can do this without closing any of the eight refactorings. A side panel in his editor can provide an overview of all ongoing refactorings and the steps that are still incomplete. When all the steps that remain are located in the test-class, he can use the Impact View to navigate to the callers located there and inspect the test code. He can edit or delete the individual test callers, and the tool will update the list of remaining impacts to indicate if any callers are remains to be addressed. During this process, the

tool continues to help him keep track of which impact belongs to which initial change.

Note that Bob can still interact with the compiler errors like before. They are still present in the code (5) since he actually did delete the parameter. He could choose to follow the same workflow as before, relying solely on compiler errors, but still benefit from the information and navigation provided by the tool. Rather than trapping Bob in an all-or-nothing situation, the tool lets him try out simple functionality before he must trust it to change his code. By navigating to each location that will be changed, the tool provides the automated refactoring as a sequence of stepwise local changes. Thereby, Bob benefits from the automated refactoring tool without deviating from his preferred workflow.

6.3.2 Alice and the Stepwise Approach

Alice began the task by invoking `change-signature` from the refactoring menu. This time, the tool presents her with a simple interface that overlays her code as seen in Figure 6.5. The view indicates the ongoing refactoring ("Change Signature") and marks the impacted locations as the caller on line 21 and the method's implementation on line 24-35. At this point, she has not yet indicated that she wants to remove a parameter.

The view shown in Figure 6.5 corresponds to the configuration view with which Alice previously interacted. She can select the parameter in the Refactoring Editor (the method declaration in the overlay interface) and delete the parameter either with her keypad or by invoking a menu that provides her with the possible actions. This corresponds to using the traditional Configuration View in the `change-signature` to indicate the `remove-parameter` refactoring.

Once she indicates the *initial change*—the parameter-removal—the tool provides her with feedback on the initial change and indicates the impact in the source code, as seen in Figure 6.6. Rather than separating "problems" in a Problem View from "changes" in a Preview View (Chapter 2.1), this presents her with information about all impacts in a single, comprehensive view. The tool provides additional changes for the impacts that the tool is able to reason about, such as impacted callers.

In Alice's previous workflow, she chose to cancel the refactoring. She then had to manually locate the problems and remember which errors the tool had shown her. Here, she can use the Impact View to navigate to the impacted locations while the problems and impacts are still visible to her and can even edit the source code while the view remains visible. For example, she may choose to remove the overloaded method and the usage

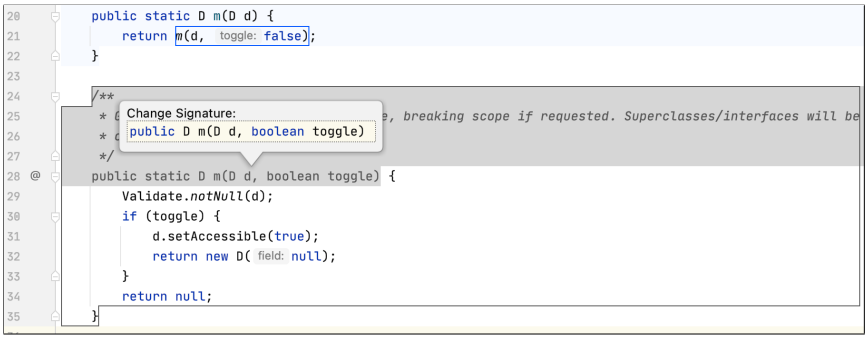


Figure 6.5: Alice invokes the stepwise refactoring tool through the refactoring menu. The tool presents an initial interface where no changes have been made. However, Alice's intent to change the method's signature has been captured and used to indicate potential impacted areas.

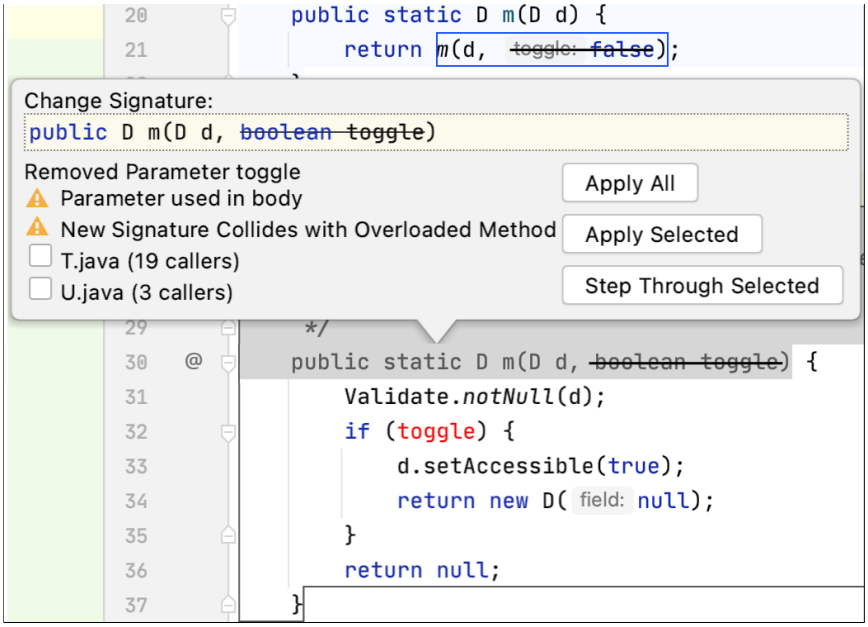


Figure 6.6: Alice has indicated that she wants to remove the `toggle` parameter and the tool indicates the impacts. Since Alice has not yet removed the parameter from the source code, there are no compiler errors yet. The proof-of-concept tool that I implemented depends on the removal of the parameter from the source code but this figure shows the parameter still present to better illustrate Alice's workflow.

```
@Test(expected = IllegalArgumentException.class)
public void testGetFieldIllegalArgumentException3() { U.m(new D()); }

@Test(expected = IllegalArgumentException.class)
public void testGetFieldIllegalArgumentException4() { U.m(new D()); }

@Test
public void testMToggle() {
    assertEquals(PublicChild.class, U.m(new D()));
    assertEquals(parentClass, U.m(new D()));
    assertEquals(parentClass, U.m(new D()));
    assertEquals(parentClass, U.m(new D()));
    assertEquals(parentClass, U.m(new D()));
```

Figure 6.7: The callers in the first two tests used to call the wrapper method and the callers in the last test used to call the method with the extra parameter. After the wrapper method is removed and the `remove-parameter` refactoring is applied, these bind to the same method and are indistinguishable. Here, the callers that had their argument changed are still indicated in blue and can be distinguished from the ones that did not.

of the parameter on line 32. In response, the tool updates the Impact View to indicate that the problems have been resolved and indicate instead that additional callers will be captured. At this point, Alice can continue the refactoring without needing to re-invoke and re-configure the tool.

At this point, she can apply the additional changes. However, this view lets her select which changes she wants to apply, either step-by-step, or as a group of changes. She selects U.java to indicate that she wants to apply the changes in the class U.java for now. She then clicks the Apply Selected button and the tool automatically updates those callers and indicates to her that only the callers in T.java remain. At this point, she can leave the refactoring incomplete and continue to the next method, repeating the workflow there.

The ongoing refactorings are indicated in a panel next to her editor to provide her with an overview and a kind of *checklist* of her ongoing changes. The panel indicates which additional changes have been applied or not. In this way, she can invoke the refactoring in all eight methods before completing any of them. Once she completes her work in this class, she can move to T.java and inspect the test code. If she chooses to remove the test callers that uses the extra parameter, the refactoring panel will update to indicate that no callers remain in T.java and that all eight refactorings are completed.

This workflow might depend on Alice's foresight that she should change one file at a time. However, consider instead if she selects Apply All and updates all the callers in the tests as well. Will she have the same problem as in her previous workflow? Figure 6.7 shows that the impact is still indicated by the blue impact-boxes. This shows her which

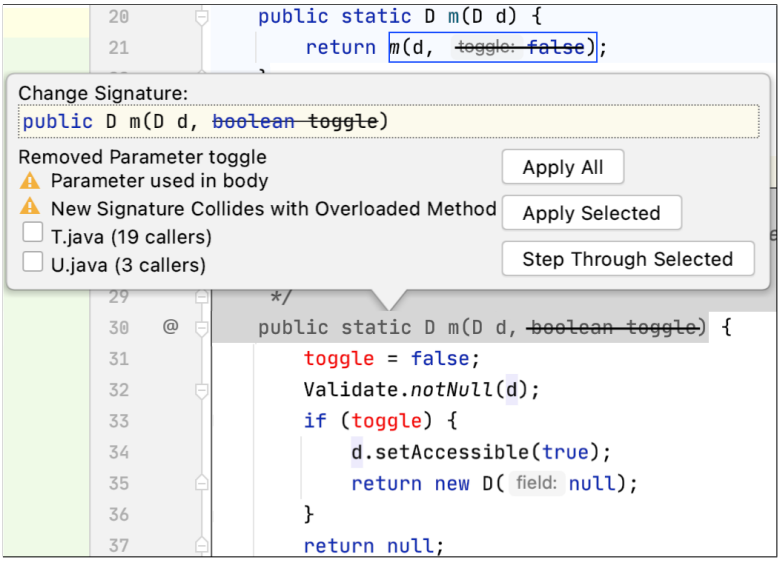


Figure 6.8: Charlie has invoked the refactoring and indicated that he wants to remove the parameter but has not actually deleted the code yet. There are no compiler errors yet and he can edit the code and run the tests. He assigns the value `false` to the parameter on line 31 and runs the tests. Later, he can continue the refactoring. The proof-of-concept tool that I implemented depends on the removal of the parameter from the source code but this figure shows the parameter still present to better illustrate Charlie's workflow.

callers used to call the method with the additional parameter and which used to call the wrapper method. If she needs to see what parameter value was used, she can use the refactoring panel to temporarily revert the additional changes in this location.

This example workflow indicates that a developer that wants to interact with a stepwise refactoring tool similarly to a traditional refactoring tool can do so, and gets access to the same information and configuration options as offered by traditional tools. In addition, this stepwise tool offers more provisional changes and better control over when the developer considers the various choices they need to make to determine the best way to alter the source code. This tool lets Alice both benefit from automated navigation and editing and lets her delay the choice about how to handle the callers in the other class until a time where she was ready to consider the necessary information.

6.3.3 Charlie and the Stepwise Approach

Charlie also began the task by trying to use the `change-signature` refactoring, but reverted his approach upon realizing the impact it would have on later steps in his

workflow. In contrast to Alice, Charlie wanted to temporarily alter the code locally, run the tests, then navigate to and edit the impacted tests. Only then did he want to apply the refactoring.

With the stepwise refactoring tool, Charlie can invoke and configure the tool similarly to Alice and interact with a view like the one shown in Figure 6.6. However, he can *also* edit the source code, as shown in Figure 6.8, without cancelling or completing the refactoring. This lets him prepare the refactoring—that is, to make a provisional change—and then run and edit the tests, similarly to before. By applying this pattern to all eight methods, that is, by *initiating* the refactoring in each of the eight methods without *applying* any of them *yet*, he is able to realize the promise of the refactoring tools: to save him from repetitive navigation and editing. Once he finishes editing the tests, he can then complete the initiated refactorings by applying all the remaining steps from all eight refactorings through a side panel rather than revisiting all the locations in the source code.

6.4 Discussion and Future Work

The usability problems that these personas encountered here are representative of the problems that the developers that I studied (Chapter 3) had experienced. In Chapter 4, I identified four usability *themes* that emerged from these participants' experiences. The four themes are listed in Table 6.2, and provide insight into which factors impact developers' experience of usability of refactoring tools during software change tasks. Here, I discuss how a stepwise refactoring tool addresses each of these four themes.

6.4.1 Tool communicates capabilities

A stepwise refactoring tool can be invoked both from a traditional refactoring menu and through a manual editing of source code. While the traditional refactoring menu approach poses well-known problems, such as developers' inability to remember refactoring names and their inability to select the right source code (described in Chapter 2.6 and Chapter 3), some developers might still prefer this mode of invocation. However, by expanding the invocation mechanism to include the recognition of manual editing patterns, usability problems that relate to the initial communication between the tool and the developer can be mitigated.

In essence, the developer can *show*—not *tell*—the tool what they want to change, thereby

removing the initial hurdle of understanding how to communicate to the tool.

In response to an invocation, the stepwise refactoring tool communicates back to the developer what it has recognized as an initial change. In Figure 6.5, the tool has been invoked from the menu and already indicates which refactoring is invoked and the areas that could be impacted. In doing so, it communicates that it is capable of detecting impacts in these areas. Consider how this compares to the refactoring tool workflow described in Chapter 2.1, in which the developer must configure the refactoring tool before learning about its ability to update impacted locations.

In Figure 6.3, the tool has been invoked through a manual code change. The Impact View both communicates to the developer what it detects as an initial change and immediately also shows its ability to detect problems and impacts and produce additional changes. The additional code changes are indicated by highlighting the argument removals, thereby showing the change on the element rather than describing it in a separate view such as the Configuration View or the Preview View. This communicates to the developer the capabilities of the tool to automatically perform the additional changes associated with the `remove-parameter` refactoring and does so using terms (i.e., overlays over the actual code elements rather than descriptions of the code elements) that are familiar to the developer.

6.4.2 Tool communicates change

The way a stepwise refactoring tool communicating the changes it makes is a continuation of its communication of its capabilities. The change that will happen is indicated in the code and requires minimal effort from the developer to imagine. In contrast, traditional tools separate the change (e.g. a problem in a Problem View or a change in a Preview View) from the elements that will be change (i.e., the source code elements in the editor). This design puts higher demands on the cognitive effort required from the developer to "map" the change that a certain view communicates back onto the source code elements. Here, the change is shown next to, or on top of, the impacted elements, as illustrated by Figure 6.3 and Figure 6.4.

6.4.3 Developer guides tool

The stepwise refactoring tool addresses this theme by providing the developer with the ability to step through the individual changes. Rather than enforcing a complete configuration upfront, the developer can guide the tool by controlling individual steps (Fig-

Table 6.2: The Usability Themes that emerged in the meta-card sort in Chapter 4. These four themes were found to be important for developers' experience of a refactoring tool's usability during software change tasks. The rightmost five columns show how these themes related to the five usability factors, listed below.

Theme	Description	P	Efe	S	Efi	T
Tool communicates capabilities	A tool's user interface must communicate clearly and directly to a developer in terms familiar to a developer. The tool should guide a developer in its use, including providing intelligible error messages.	✓		✓		
Tool communicates change	A tool should make the changes it will make to code clear before the tool is executed. It should be possible for a developer to inspect the changes that the tool has made.	✓	✓	✓		✓
Developer guides tool	Developers wish to guide how a tool executes in several ways: applying an operation to many elements easily, excluding the application to some locations of code, and altering how particular code is changed.	✓	✓	✓	✓	✓
Developer cost-benefit analysis	Developers assess the cost of applying a tool before they invoke the tool. Developers want to assess whether it is better to proceed with a tool or manually at the start of considering using a tool. The tool should help them with this assessment.	✓	✓		✓	

P = Predictable Efe = Effective S = Satisfying Efi = Efficient T = Trustworthy

ure 6.4) or groups of steps. Increasing the developer's control over how small or large of a code transformation is applied at once lets the developer ensure that the tool is applied in a way that helps them progress in their task.

This chapter has presented examples of usability problems related to this theme. Alice and Charlie were unable to guide the tool in their initial workflows. With the stepwise refactoring tool, they were able to control the timing and locations of the changes. Bob was also able to guide the tool, however, in his case, the tool also guided Bob in his changes. What these examples all have in common is that the stepwise refactoring tool provide the developer with more control and information than a traditional tool.

6.4.4 Developer cost-benefit analysis

This theme is difficult to address without knowing what constitutes the costs and benefits of using a refactoring tool, from a developers' perspective. One could attempt to quantify the benefits of using a tool through source code metrics or reduction of bugs, and the cost by indicating to the developer how long a tool would take. However, these might not be the right measures of cost and benefit. Instead, in the stepwise refactoring tool, I take a different approach. I aim to lower the cost of *invoking* a tool, and to provide the developer with information about the changes that will be made *early* in the process.

By providing this information at an early point in the workflow, the developer can *pay* the initial low cost associated with invoking the tool, and *benefit* by collecting information about the impacts and additional changes. Then, they can make a more informed decision about whether they want to continue the process with the tool or to cancel the tool and complete it manually. By providing this information at an early point in the tool's workflow, the developer does not waste much time and effort to learn about the impacts, and might invoke the tool more often even though they might not always choose to use it to complete the change. In comparison, a traditional refactoring tool requires a complete configuration before the developer can inspect the impacts and additional changes, and consequently, the price of admission to access this information is comparatively higher.

In this way, I aid the developer's assessment of the costs and benefits associated with using the tool as early as possible. This places particular importance on the tool's *efficiency* at the cost of it's *trustworthiness*. According to the usability profiles presented in Chapter 5, this is a tradeoff developers are willing to accept.

6.4.5 Implementation

To explore the feasibility of, and the implementation challenges associated with, a stepwise refactoring tool, I created a stepwise tool for the `remove-parameter` refactoring that was used in this task. I implemented the support as a plugin for IntelliJ. Implementing a refactoring tool from scratch is a significant task; to reduce the effort of this proof-of-concept tool, I reused some of the implementation of `change-signature`[2] and some of the code base for `rename`.

While this reuse simplified some parts of the implementation, it complicated others. The `change-signature` implementation in IntelliJ is highly reliant on the "all-or-nothing" workflow depicted in Chapter 2.1. This is common in refactoring tools. I also attempted to create a plugin for Eclipse by reusing code from the corresponding Eclipse refactoring codebase[3] and observed a similar problem.

The reliance on the workflow is beneficial for ensuring correctness. Consider, as described in Chapter 2, a refactoring operation as a pattern of code changes and the importance of ensuring that these changes will not alter the program's behavior. This is achieved through condition checks. In these IDEs, invoking a refactoring operation triggers a number of condition checks. If these conditions hold, they will not be checked again; instead, they must remain valid until the refactoring is applied. This is achieved by "trapping" the developer in a sequence of views while the code changes are prepared and preventing code changes from occurring between the invocation of the tool and the application of the changes. This is illustrated in Figure 6.2. In this design, increased correctness is achieved by as a tradeoff against increased rigidity of the tool.

The refactoring implementations in these two IDEs follows the principle of one code transformation per invocation and there is no way to apply only parts of the refactoring while retaining the remainder for a later application, nor to edit the code between the invocation and the completion of the refactoring. This meant that, in my implementation, I was unable to reuse any code that prepared and applied the code change. Instead, I implemented my own rules of determining the impacted PSI-elements and "manually" updated the PSI according to those rules.

Figure 6.3 shows the main mode of interaction with the stepwise refactoring tool. The navigation works as described and it is possible to select only one class and step through those changes or to apply all at once. If individual steps are applied, the view that is shown in Figure 6.4 is used to navigate around the codebase. There is currently no

[2] `https://www.jetbrains.com/help/idea/change-signature.html`

[3] `https://help.eclipse.org/2021-06/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm?cp=1_4_6_0`

support for marking locations to be handled later; at each step, the location can either be "applied" (which will remove the parameter), be "skipped" (which will exclude it from that change and move to the next location), or be manually edited.

I previously described how the tool recognizes Bob's parameter deletion as an initial change and invokes the `remove-parameter` refactoring in response. This is the way that the tool currently recognizes an invocation. This might be difficult to scale: I only implemented the recognition of a single *initial change*, while in practice, there may be many different kinds of code changes that could be used as initial changes. Some work has previously looked into using editing patterns to invoke refactorings, such as the tool BeneFactor [50] and WitchDoctor [45]. Future research should investigate more editing patterns that can be used for this purpose.

The liveliness of this tool is challenging to implement, and I did this by updating the refactoring on regular intervals and comparing the method's start state (its declared signature) with the current one, to determine the changes that had been initiated. Because the logic in my implementation depends on this comparison between the start state and the current state, and the hover view currently does not support editing, this tool does not currently support the workflow that Alice and Charlie used without compiler errors. However, this could easily be rectified by making the hover view editable and introducing an additional comparison between the source signature and the signature that is shown in the hover view.

It is also challenging to maintain references to the impacts in a way that is resilient to code changes. Normally, a refactoring tool operates on a parsed representation of the source code, represented as an abstract syntax tree (AST) or a similar data structure. If the source code is changed, then the pointers into the data structure that the tool retains in memory might no longer be valid. For example, in Alice's stepwise approach when she was stepping through the `remove-parameter` refactoring, consider a case in which she decides to `rename` that same method between two steps. If the AST elements are identified by their text, then the refactoring tool's pointers to the callers will no longer be valid. If she reorders the methods, and they are identified by location, then they will also no longer be valid. Or, if she extracts a new class that contains some of the callers, then they will still bind to the same method, but the pointers that the refactoring tool has to them are no longer valid. In order to correctly continue the refactoring at this point, another search for callers must be made. Alternatively, the data structure used to represent the program code must be handle such code changes.

In this proof-of-concept tool, I used the data structure provided by IntelliJ called Program Source Interface (PSI) [59]. This structure remains valid through many types of

program changes. For example, considers an example in which Alice wants to apply a step of the `remove-parameter` refactoring and remove the argument of a method call on some line 20. Before doing so, she inserts a new and unrelated line of code on line 19. In most ASTs, this would change the position of node that represents the call on line 20 (now 21) to a different location in the tree. However, in the PSI, the reference to that location remains valid. However, some changes will render the PSI invalid as well. In those cases, I attempt to use text highlighting elements to locate the new PST element. I also perform frequent additional searches for callers and references throughout the refactoring steps to ensure that the refactoring tool's "knowledge" is up to date. This slows down the execution. I also limit the changes that can be made between steps. For example, code can be removed and changed, but cannot easily be moved between files. Future work should consider how to handle more complicated intermittent changes.

While this tool supports interleaving refactoring steps with other source code edits, it does not update the Impact View with any further information, such as the additionally captured callers. To do so would require answering the question of what constitutes an impact. This chapter has argued for rethinking the refactoring tool. Similarly, the *impact* of a refactoring has to be rethought. What does a developer consider the impact to be? How much information can be presented, and what are the most important ones? A hover view like this can easily become cluttered with too much information. Further research should investigate what impacts developers need to know about. A realistic tool should provide the developer with an ability to configure this and to define their own types of impacts.

Finally, a benefit of this tool for the developer personas was the ability to invoke multiple instances of the refactorings and use a side panel to manage them. Unfortunately, the reuse of IntelliJ source code complicated the creation of multiple instances of their refactoring, and my tool does not currently support this. This is a simple consequence of relying on IntelliJ's singleton pattern for parts of the codebase; consequently, the tool does support applications of certain other refactorings between steps. For example, Figure 6.9 shows an application of `change-signature` interleaved with the `rename` refactoring. A future version of this tool should support multiple invocations of the same refactoring instance and implement a side panel to keep track of the ongoing instances.

6.5 Summary

The usability of refactoring tools can be improved by addressing usability themes that emerge from the experiences that developers have with these tools. In this chapter, I

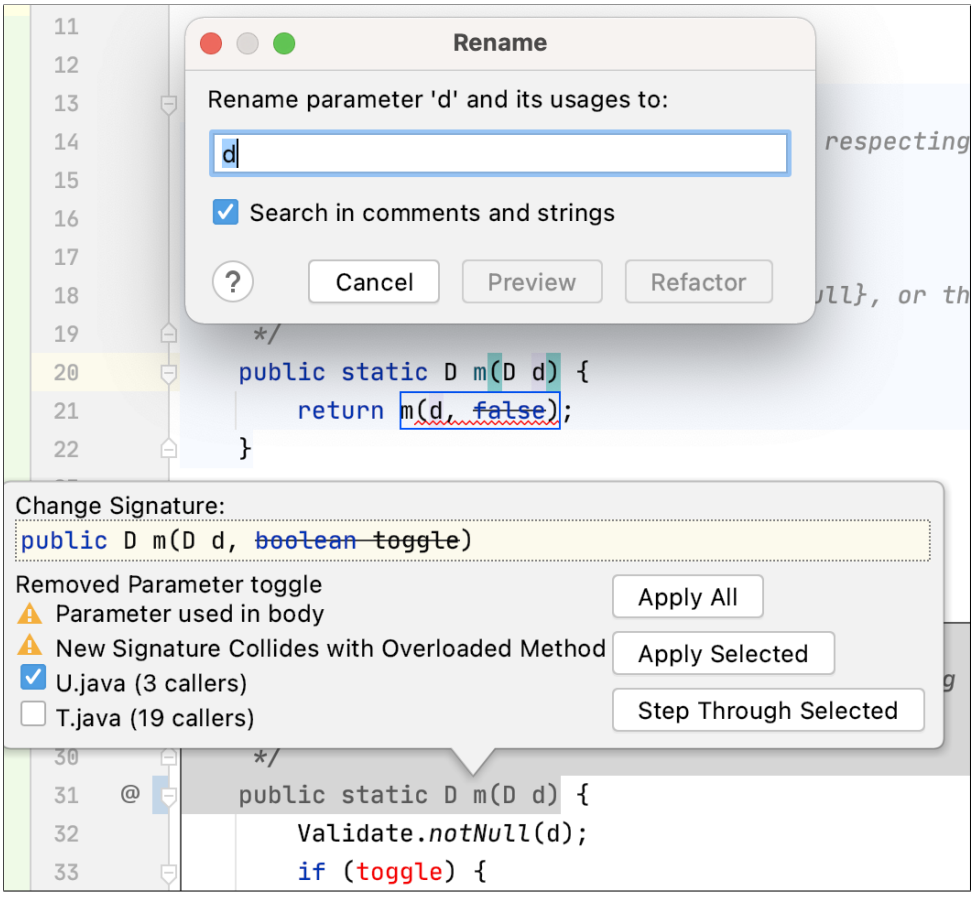


Figure 6.9: During the application of the stepwise `change-signature`, a developer might realize the need to rename another code element. The stepwise interaction model allows the developer to invoke and apply the `rename` refactoring before continuing the `change-signature` refactoring.

used three developer personas to illustrate how the interaction model of traditional refactoring tools hindered them from benefitting from these tools in three different workflow scenarios. I then introduced the interaction model of a stepwise refactoring tool that addresses the four usability themes and illustrated how it enabled the personas to utilize the stepwise refactoring tool during their workflows. This illustrates the unrealized potential of current refactoring tools and ways in which toolmakers can realize the promise of refactoring tools—to help developers change code faster and with fewer errors—by adopting a stepwise interaction model.

Chapter 7

Discussion and Future Work

> There is surely nothing quite so useless as doing with great efficiency what should not be done at all.
>
> — *Peter Drucker, Managing for Business Effectivenes, Harvard Business Review*

With this dissertation, I have made contributions to the understanding of refactoring tool usability. My research spanned from studies of developers to a theoretical investigation of usability, and to the implementation and design challenges associated with tool implementations. Yet, many questions remain. This chapter reviews the outstanding questions in these areas by discussing the potential for future research.

7.1 Studying Developers' Refactoring Experiences

Throughout this dissertation, I have argued for the need to study how developers use and experience refactoring tools. With the laboratory study described in Chapter 3, I looked in depth at the experiences of a small group of developers in realistic scenarios. The survey in Chapter 5 let me triangulate their experiences with another, larger, group of developers. This incremental investigation let me iteratively refine the questions that I investigated as I uncovered new insights about existing refactoring tools and the ways in which they supported or hindered developers in their workflows.

Still, many questions remain. More research is needed to address shortcomings of the studies presented here and to collect further data. The studies that I undertook were

limited in scope and scale. They focused on only a small set of refactoring operations and a limited set of software changes. Future research in this area should study a higher number of developers, a wider variety of refactoring operations and tools, and more varied software changes. Developers' applications of refactorings should also be studied during their normal work, either through field studies or by collecting data from their programming environments. These study methods could help both to identify the contexts in which refactoring operations are frequently applied and the ways in which developers use these tools.

More study is also needed on the usability to refactoring tools as they exist today. For example, the developers that I studied found the Preview View to be insufficient support for predicting how the refactoring would impact subsequent steps in their workflow. Future work should investigate whether Preview Views are effective tools for this and consider if such views need to be improved or if other mechanisms have to be employed altogether.

7.2 Refactoring Tool Usability

Studying developers as they work enables insight into their activities and experiences with refactoring tools, but it is difficult to act on that insight without a deep understanding of usability. Without this understanding, efforts aimed at improving one experience might worsen another. For example, a refactoring tool's trustworthiness and predictability might be improved by limiting the accepted invocations to programs that can be easily analyzed and transformed but in doing so, the tool becomes less flexible for the developer to use. Or, the tool's effectiveness and efficiency might be increased by accepting potentially erroneous invocations but this reduces its trustworthiness because the code changes might occasionally introduce errors. Creating usable refactoring tools requires the toolmaker to determine the optimal (or acceptable) tradeoffs between such considerations.

What constitutes the optimal tradeoff depends on the user, their goals and capabilities, and on the context of use. I approached this topic through the framework put forth in ISO 9241-11 [5] and as described in Chapter 4. The resulting theory of usability for refactoring tools, and the usability factors identified in the same chapter, helped shape the investigations into the collected data. Through this lens, I identified four usability themes and approached these with the design of the stepwise refactoring tool described in Chapter 6. The proposed tool makes certain design tradeoffs according to the themes that emerged from the study data. However, these tradeoffs may not be optimal for all

demographics of developers.

Additional studies of developers could shed light on the variety of the needs from various developer demographics. The findings presented in Chapter 5 indicate that such varieties exist between the users of various IDEs; similarly, there might be varieties between developers who work in different types of teams, on certain kinds of projects, or in particular cultures. Future research that looks at a larger demographic of developers could add to the understanding of such variances which, in turn, help toolmakers define and target the demographic of their tools.

Additional research is also needed to understand whether such variety is best addressed by a one-size-fits-all approach that aims at designing for the larger demographic, or whether such considerations can be included in the communication between the refactoring tool and the developer. For example, researchers could investigate the value of enabling configuring the refactoring tool to specify the developers' risk-tolerance, certain no-go areas in the code, or other relevant factors.

Future work could also investigate whether developers' experiences of refactoring tools are impacted by the degree of familiarity that a developer has with the code base. For an example of this, consider investigating whether a developer might have a need for *predictability* and *trust* when using a refactoring tool in an unfamiliar codebase, and a higher need for *efficiency* than trust in a highly familiar codebase. In the latter case, the developer might be able to quickly identify changes, while in an unfamiliar codebase, they might risk overlooking changes that the tool made. Such insights could bring valuable insights to future tool designers but were outside of the scope of this dissertation. Future studies are needed.

Any communication between the developer and the tool must employ terminology that is either understood or learned by the developer. Further studies could be used to identify current and common terminology that can be employed in the tools. Again, a balance must be struck between using terminology that is expressive enough to capture the information that is necessary to convey while simple enough that the intended demographic can understand it.

Alternatively, additional education of developers can mitigate difficulties with terminology but might be unrealistic at scale. One interesting topic to investigate in this direction is the educational impact of integrating refactoring information into git diff views. Tools like the RefactoringInsight plugin [70] can detect refactorings that occur in commits and enrich the git diff view with information about these refactorings. This might aid developers in learning about refactorings that they can later invoke and use. For the design of a tool to be predictable for a developer, it should share similarities

with other designs that the developer has encountered before or somehow conform to their mental model [36]. If tools like the RefactoringInsight plugin can enrich developers' experience with the git diff tool to include information and terminology that aligns with the available refactoring tools, the usability of these tools might be improved for those developers. Future research should investigate the effect of such tools and ensure that the two designs are not dissimilar.

7.3 Refactoring Tool Support

Chapter 6 introduced a stepwise interaction model for refactoring tools. This model addresses many of the usability themes that emerged from the study data presented in this dissertation. Many of the usability problems arose from hurdles in the communication between the developer and the tool, and a resulting experience of lack of control for the developer. The stepwise refactoring tool address these hurdles by presenting the developer with an alternative workflow, in which the refactoring is performed stepwise, the developer can interact with each step, and individual steps can be interleaved with other changes.

This represents a particular tradeoff in its design: by prioritizing the communication between the developer and the tool, its usability during software change tasks is improved, at a potential cost of correctness. Some might express concerns about an interaction model for refactorings that is not behavior preserving. If this interaction model is used in a manner where the developer Apply All the transformation steps, and does so only if there are no warnings or problems, then behavior will be preserved. If the stepwise refactoring approach is used otherwise, behavior may not be preserved, but this is no different than the situation that exists now for refactoring tools. If a developer uses an atomic refactoring tool and chooses to proceed even if problems are identified by the tool, or to exclude some parts of the refactoring in a preview view, the refactoring will not be behavior preserving. A stepwise refactoring tool might not be able to guarantee behavior-preservation, but can instead provide the developer with more visibility about the impact of the transformations it will apply than is available in current refactoring tools. Future work should employ user studies to evaluate the impact that this tradeoff has for developers.

From this tradeoff follows another challenge: to generate and apply the individual code transformations that correspond to refactoring steps. This requires both determining which of the code changes in a refactoring constitute steps that are meaningful for a developer, and to support their initiation and execution as separate transformations.

There are well-established sources of typical steps in a refactoring [47] but additional research is needed to evaluate the utility of these steps, to identify additional steps, and to determine how to meaningfully represent these steps in a developer-friendly user interface.

The ability to use an initial change to invoke the tools solves multiple usability problems, such as developers' inability to remember refactoring names, to select code, and to invoke the refactoring on the right code element. A number of tools have previously focused on detecting and completing manual refactoring edits. GhostFactor [49], used a notification system similar to "quick fixes" to alert the developer about missing changes in refactorings that are applied manually. WitchDoctor [45] and BeneFactor [50] detect code editing patterns of a manual refactoring and invokes the appropriate Eclipse refactoring tool to complete it. ReSynth [100] goes further by generating sequences of refactorings that can complete the change. These tools provide an alternative invocation method for accessing mainstream refactoring tools. Constraint-based refactoring tools can also generate additional code changes to restore a program's semantics [116, 117]. However, current mainstream tools do not support this invocation mechanism. One reason might be the difficulty associated with determining which refactoring the developer wants to invoke. Future work should aim to identify more invocation patterns of refactorings that can be used for this invocation style.

Another reason that mainstream tools do not utilize this form for invocation might be the risk of false positives, such as initiating or applying unintentional changes. Consider if the wrong refactoring tool is applied in response to a manual change. The developer would need to close the tool or revert its application, and potentially redo the manual changes. In fact, several of the usability problems that I observed in the laboratory study would be worsened by such an experience, such as lack of control and difficulties with integrating the refactoring into their workflow. This illustrates the type of design tradeoff prevalent in refactoring tool design. The stepwise interaction model might enable this mode of invocation in a more seamless manner. By employing the model of an *initial change*, *impacts*, and *additional changes*, and by enabling the developer to adjust and delay the remaining steps of the refactoring to an appropriate time, the cost of erroneous invocations is reduced. Future work should investigate how the experiences that developers have with this type of invocation are impacted by the overall interaction model of the refactoring tool and whether the usability profiles described in Chapter 5 can support toolmakers in making such tradeoffs.

In terms of implementation, it is useful to identify areas where existing refactoring tools can be reused. Chapter 6 detailed some implementation problems that I encountered in the attempt to reuse existing source code from the refactoring tools in mainstream IDEs

for this tool. Most refactoring tools have ad-hoc implementations that are tightly bound to the workflow described in Chapter 2.1. This means that implementing a stepwise refactoring tool is not done simply by making minor adjustments to existing refactoring tools. However, the use of enriched tree-structures for representing program code, such as the PSI used in IntelliJ [59] has great potential for reuse.

In fact, it might be beneficial to think about a stepwise interaction model as an interaction model for a tool that can perform impact analysis and stepwise code propagation where refactoring is only one type of change propagation (Chapter 2.7). Examples of such tools are JRipples, which uses dependency analysis to detect locations that might be impacted by code changes [27], and Chianti, which indicate impacted locations in tests based on a code change [102]. However, these tools have limited ability to understand the intention behind the changes, and consequently, they might produce overly many false positives, that is, locations which are somehow dependent on the changed location but do not need to be updated. Moreover, from this analysis, a tool can not easily determine the additional changes that are necessary to propagate an initial change to impacted locations. The stepwise refactoring tool described in Chapter 6 instead bases the detection of impacted locations on the definitions of the refactorings [47, 94]. This ensures that the proposed locations are relevant for the developer and enables the identification of additional changes. This is somewhat similar to tools that support partial application of refactorings on API clients, such as CatchUp! [57] and Trident [62]. Future work should investigate how these types of tools can be combined and improved to better support developers' software changes.

A developer's experience with the stepwise refactoring tool will have some similarities to the experience of using so-called "quick-fixes". Modern IDEs often present the developer with small suggestions for code changes that the IDE can automate, such as fixing compiler warnings or errors, applying simple bug fixes, or applying custom changes that are *learned* from the developer's own change patterns [80, 48]. Interestingly, developers can experience similar usability problems as the ones described in this dissertation when they are applying these fixes, such as lack of control and predictability. For example, in a study of developers' experiences with using static analysis tools (which include quick-fixes) to fix bugs [60], one participant described their experience with applying quick-fixes to resolve bugs, and proposed a design change that would improve the tools' usability for them. They proposed a so-called "three option dialog box": *"apply the entire fix (default option), do not apply the fix or step by step apply the solution allowing the developer to decide which parts of the solution they would like to keep."* [61, p.6]. This design closely resembles that of the stepwise refactoring tool, which was developed separately from their study. Future research could investigate whether this "three option dialog box" interface can be considered a usability improvement that addresses developers' lack of

trust and predictability in generic tools that automatically change code.

Perhaps it would be useful to combine the three types of tools discussed here: "quick-fix" tools, refactoring tools, and code propagation tools, into one kind that can guide the developer's manual changes, step through additional changes on impacted locations, and preview and apply code changes such as refactorings. One approach could be to consider a stepwise refactoring tool as a specialized change propagation tool that uses a "three option dialog box" version of the "quick-fix".

As previously mentioned, traditional change propagation tools are limited in their ability to determine the intent of a change. The limited scope of refactoring tools enables a more accurate interpretation of the developer's intent from an initial change and the possibility of using mainstream sources [47] to generate additional changes. In contrast to "quick-fix" tools which generate code changes, stepwise refactoring tools can retain a state that prevents "nonsense" proposals. For example, in Bob's stepwise approach (Chapter 6), a "quick-fix" tool might propose to remove the compiler error in a caller by introducing a boolean parameter in the method declaration. Knowing that Bob had just removed the boolean parameter, this proposal is not useful. A combined tool could retain his intent to remove the parameter, and propose the correct "quick-fix" or refactoring step. Future research should systematically investigate the relationship between these kinds of tools and how they can be combined to better assist developers.

To illustrate how the boundaries between such seemingly different types of tools can be blurred to benefit the developer, I refer to the idea of the Programmer's Apprentice [103] as put forth by Rich and Waters in 1988. They envisioned automated support for programmers that work on creation and modification of programs, and describe this support as follows [103, p.2]:

> *"We think of the Apprentice as a new agent in the software process (...) rather than as a tool. It should interact with the programmer like a human assistant. Furthermore, both the programmer and the Apprentice have access to all the facilities of the existing programming environment. The key to making this approach work is communication between the programmer and the Apprentice: It must be based on a substantial body of shared knowledge of programming technique. If the programmer has to explain everything to the Apprentice from first principles, it would be easier to do the work alone."*

While Rich and Waters envisioned an intelligent, human-like interface that was controlled through short commands, the stepwise refactoring tool can instead detect some types of code change intent based on a developer's editing activities. Then, it can retain

this intent in memory and use it to provide the developer with relevant automatic code changes that complete their intended change.

Taking inspiration from the Apprentice, one could imagine that the utility of a stepwise refactoring tool could be greatly improved by integrating into it a capability to reason about how these changes will impact the software design, beyond that of the program's behavior. Future work should investigate how such a tool can incorporate knowledge about the software development context outside of the target language, such as design documents, security requirements, runtime requirements, certain no-go areas in the code, and so on, to better assist the developer.

Chapter 8

Conclusion

> In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that it is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.
>
> — *C. Alexander, A Pattern Language* [20]

Refactoring tools have a long and rich history of research. The early ideas of this field were formulated in the 1990s and were motivated by the promise of refactoring tools to help developers change programs faster and with fewer errors. Toolmakers and researchers have put effort into improving the tools to support more refactoring operations and to increase the speed and correctness of the condition checks and the transformations that these tools perform. In doing so, toolmakers and researchers have made significant improvements to the technical aspects of refactoring tools. This can be seen in mainstream IDEs like Eclipse and IntelliJ whose standard installations contain support for around 40 refactoring operations each. However, despite this extensive support, the promise of refactoring tools has not been fully realized, because developers avoid using these tools during software change tasks and instead refactor manually.

In this dissertation, I have addressed this *disuse* as a design problem. I found that developers approached software change tasks that are amenable to refactorings with one out of three strategies and that these strategies impacted their ability to benefit from refactoring tools (Chapter 3). I also defined a theory of usability that let me identify a number of usability themes that impacted these developers' experiences (Chapter 4)

and usability profiles that could potentially help developers design tools that better fit into their workflows (Chapter 5).

By studying usability problems that developers face when they attempt to integrate these tools into their workflows, I determined a number of ways in which the design of refactoring tools can be adjusted to better support developers in their software changes. In Chapter 6, I proposed an interaction model for a stepwise refactoring tool that illustrates tradeoffs in its design. With this, I illustrated how multiple usability problems with traditional tools could be addressed. A tool that implements such tradeoffs may no longer be aligned with our expectations from refactoring tools. However, the software development process that the original refactoring tools were developed to support, has changed significantly in the past 30 years. So have the other tools in development environments, like test suites, "quick-fixes", and version control diff tools. Even some of the code changes that developers make today are different from when the earliest tools were designed.

Refactoring tools must fit into the software development processes that they occur as part of, they must integrate with other tool support for changes that surround their use, and they must be aligned with the code changes that developers want to make. By updating the design of refactoring tools to better integrate with these contexts, their promise—supporting developers in making faster and safer code changes—can be realized. With this dissertation, I hope to have helped identify how this can be done.

Appendix A

Refactoring Patterns

This chapter briefly describes the refactoring patterns that this dissertation relies upon. For a full overview of refactorings, refer to related literature [47].

A.1 Remove Parameter

This refactoring removes a parameter from a method declaration and updates all callers by removing the corresponding argument from the calls. Listing A.1 and A.2 shows a simple example in which the second parameter of a method, `foo`, is removed.

Listing A.1: A method foo with two parameters and one caller.

```
60 public static void main(String[] args){
61    System.out.println(foo("MyString", 10));
62 }
63 public static boolean foo(String s, int i){
64    return s;
65 }
```

Listing A.2: The second parameter is removed and the caller is updated accordingly.

```
60 public static void main(String[] args){
61    System.out.println(foo("MyString"));
62 }
63 public static boolean foo(String s, int i){
64    return s;
65 }
```

The method `foo` has one caller on line 61. That caller passes the value `10` as an argument to the parameter that is removed, `i`. That argument must also be removed. s

In IntelliJ and Eclipse, this refactoring is accessed by invoking the refactoring menu on a method declaration and selecting the "Change Signature" menu option.

A.1.1 Conditions

This refactoring has a number of conditions that must be true for the refactoring to be correct.

The parameter can not be in use in the method. If the parameter is used in the method from which it is removed, then the result code will have compiler errors and behavior might not be preserved. Listing A.3 shows an example in which the parameter can not be removed without introducing errors.

Listing A.3: A method foo with two parameters and one caller.

```
60 public static void main(String[] args){
61    System.out.println(foo("MyString", 10));
62 }
63 public static boolean foo(String s, int i){
64    if(i>5)
65       return s;
66    else
67       return "prefix" + s;
68 }
```

The method must not be overriding a method from a superclass or interface. If the method that is refactored overrides another method, then the resulting program might have its semantics changed. If the method overrides a method from a superclass and has callers, then those callers will now bind to the superclass method which might alter the behavior of the program. If the method overrides an interface method, then the class no longer implements the interface, which might alter the behavior of the program and lead to compiler errors.

The new (target) signature of the method must not collide with existing methods in the same scope. If it does, the program might not compile or any

callers to either method might alter their bindings. For example, if another method in the same type has the target signature, then the program will not compile. If another method in a subclass or superclass has the target signature, then the program might compile, but the meaning of the program might change.

A.2 Move Method

This refactoring moves a method from one type (source) to another (target). The method is removed from the source type and a corresponding declaration is made in the target type. Listing A.4 and A.5 shows a simple example of moving a method from one class to another.

Listing A.4: A method foo with one caller is declared in C.

```
60 class C{
61    public static void main(String[] args){
62       System.out.println(foo("MyString"));
63    }
64    public static boolean foo(String s){
65       return s;
66    }
67 }
68 class B{
69 }
```

Listing A.5: The method foo is moved to B and the caller is updated.

```
60 class C{
61    public static void main(String[] args){
62       System.out.println(B.foo("MyString"));
63    }
64 }
65 class B{
66    public static boolean foo(String s){
67       return s;
68    }
69 }
```

If there are callers to the method, they must be updated to access the new declaration. If

the method is static this is typically simpler; instance methods require a correspondence between the source and target call objects (`this`) [94]. If the method's body depends on other code elements, these must potentially be updated according to the new location. If any additional values are needed from the source context, they can be passed as arguments. In that case, parameters must be added to the method declaration.

In IntelliJ and Eclipse, this refactoring is accessed by invoking the refactoring menu on a method and invoking the "Move" menu option.

For a list of conditions, please refer to [47].

A.3 Inline Method

This refactoring replaces calls to a method with the body of that method. Usually, the method declaration is also removed. Listing A.6 and A.7 shows a simple example of this refactoring. The call on line 61 has been replaced with the method's body with the parameter reference `s` replaced with the argument value "MyString".

Listing A.6: A method foo with one caller is declared in C.

```
60 public static void main(String[] args){
61    System.out.println(foo("MyString"));
62 }
63 public static boolean foo(String s){
64    return "prefix"+s;
65 }
```

Listing A.7: The method foo is moved to B and the caller is updated.

```
60 public static void main(String[] args){
61    System.out.println("prefix"+"MyString");
62 }
```

A condition for this refactoring is that the method body can be adjusted to fit the new context. For example, if any references from the method body is not visible in its new location, this refactoring can result in compiler errors.

In IntelliJ and Eclipse this refactoring can be accessed by invoking the refactoring menu on a method or a caller and select the "Inline" menu option. For a list of conditions, refer to [47].

Appendix B

Survey Questions

The following pages contain the complete survey that was used in the study described in Chapter 5.

Landing Page

THE UNIVERSITY OF BRITISH COLUMBIA
Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Tool Use in Software Change Tasks

Do you wish it was easier to change software? So do we. And we want to make that happen.

First, we need to know how developers use the tools that already exist. The purpose of this survey is to understand developers' experiences with mainstream tools. We focus on development tools for Java and C# programs, because they are popular, mainstream, and allows us to build on others' work.

The survey is expected to take no more than 15 minutes to complete. You participate by answering questions in an anonymous, web-based questionnaire that you can start, pause, and complete at your convenience. Your responses are anonymous: we do not collect identifying information.

If you choose to participate, you will be asked to share your experiences with a few mainstream tools. You will also be asked to answer questions about tool use in the context of a software change scenario that has occurred in an open-source codebase.

In addition to being a contributor to our research goal, you may find it interesting to learn about alternative ways to solve real software change problems and to reflect on your own tool usage.

This study is conducted by Dr. Gail Murphy, Dr. Anya Bagge, and graduate student Anna Eilertsen. It is funded by the Research Council of Norway under grant number 250683 (Co-Evo).

For more information, continue to the consent form on the next page.

Informed Consent

THE UNIVERSITY OF BRITISH COLUMBIA
Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Tool Use in Software Change Tasks

Welcome to our study on tool use in software change tasks. This first page provides survey information and asks for your consent to participate. If you consent, you will be taken to a set of questions that ensure eligibility to participate. If you are eligible, you will be taken to the survey.

Principal Investigator:
Dr. Gail Murphy, Dept. of Computer Science, University of British Columbia (murphy@cs.ubc.ca, +1 604 822 5169)

Co-Investigators:
Anna Maria Eilertsen, graduate student, Institute for Informatics, University of Bergen (anna.eilertsen@uib.no, +47 401 03 368)

Other Investigators:
Dr. Anya Bagge, Institute for Informatics, University of Bergen (anya.bagge@uib.no, +47 482 71 775)

Study Purpose:
The purpose of this survey is to learn about how software developers experience tools that can help them make software changes. We want to gather data about the different tools that developers use when changing software and how they fit into developers' workflows. The data resulting from this study will be used to design tools that better support software evolution and maintenance activities.

What you will be asked to do:

This is a web-based survey. You will be asked to read and respond to a series of questions regarding your experience with a number of tools that developers use when changing software. The survey is estimated to take 15 minutes or less. You will be asked to read a description of a software change scenario and answer questions about the steps and tools that you would approach it with. You will be asked qualifying questions to ensure eligibility. You will be asked non-identifying demographic questions.

Known Risks:
The main risk is the time required to answer the survey. This amounts to the time it takes you to respond to the questions in this web-based survey. We are mitigating this risk by distributing the survey such that you may respond at a time that is convenient for you. Also, you can terminate your participation in the survey at any point in time without providing any reason. Otherwise, the risks involved in this study are minimal and are those commonly associated with the use of computers, such as potential eye or wrist strain.

Potential benefits:
You may find it beneficial to learn about different tools for software changes. You may also find it interesting to reflect on your own tool use. Direct benefits can arise if our findings lead to automated tools that can better support you in your daily software evolution and maintenance tasks.

Compensation:
You will not be compensated for participating in this survey.

Data, Storage & Confidentiality:

Survey responses will be stored on password-protected and encrypted devices. No personal information will be collected and as a result all data is anonymous. This means that once you complete and submit your survey responses, we can no longer remove data if you choose to withdraw your consent.

You will be identified by number or pseudonyms in any internal or academic research publication or presentation. If we choose to use some of your comments, they will be attributed to a participant number or pseudonym. At no point in time will your employer have access to the identifying information.

The anonymous data may be seen by other researchers for educational purposes or the application of further scientific methods. Papers published on this data may also require the processed data to be submitted to an online repository or database where other researchers and members of the public will be able to access it. Note that you will not identifiable in this data.

The survey responses will be stored for at least five years and may exist longer, in open repositories, to enable other researchers to benefit from the data. All of the data collected and stored is anonymous.

Use of the Data:

The results of this study will potentially appear in both internal and external academic research presentations and publications, such as academic journals and conference proceedings. Note that you will not identifiable in this data.

The data collected in this study may be useful for designing better tools for software developers or benchmarking tools. We may wish to use the collected data in the future for this purpose. Please note that this data does not require any information that can be traced back to you or your personal data.

Contact for information about the study:

If you have any questions or desire further information with respect to the study, you may contact Dr. Gail Murphy (murphy@cs.ubc.ca, +1 604-822-5169).

Who can you contact if you have complaints or concerns about the study?

If you have any concerns or complaints about your rights as a research participant and/or your experiences while participating in this study, contact the Research Participant Complaint Line in the UBC Office of Research Ethics at 604-822-8598 or in long distance email RSIL@ors.ubc.ca or call toll free 1-877-822-8598.

Study Ethics ID H20-03787

Consent

Your participation in this survey is entirely voluntary. You are free to withdraw your participation at any point without providing any reason. Any information you contribute up to your withdrawal will be retained and included in the dataset unless you request otherwise.

By clicking "I consent" below and providing your name and the date, you confirm that you:

1. *understand what is required based on reading the information provided above,*
2. *understand that your participation is voluntary and you are free to withdraw at any time,*
3. *understand the provisions of confidentiality,*

4. *consent to participate in the study.*

- ○ I consent and wish to participate
- ○ I do not consent, I do not wish to participate

DeclinedBlock

You declined the consent form.

You are not eligible to participate in this survey. In order to ensure that the collected information is relevant and up to date, we require participants to have two years of professional experience with Java or C# programming from within the last ten years.

Thank you for your consideration. Click the "next" button to end this survey.

Background Questions

Background Questions

On this page, we present you with the terminology we use in this survey. Please read it carefully.

> **Software change tasks** refer to activities where developers apply changes to existing software for purposes such as adding or refining a feature, correcting a mistake, or improving code quality.
>
> When applying changes, developers may also need to take actions like inspecting or validating changes they have already made and actions that help them understand the code, such as navigating or reading source code, test code, and documentation.
>
> We use the term **tools** to refer to programs that help support or automate the types of software change activities previously outlined: applying changes, inspecting changes, validating changes, and comprehending source code, test code, and documentation.
>
> Developers typically perform one or more of these activities in an integrated development environment (IDE) or a text editor. We refer to these collectively as **programming environments**. Tools may be accessed both from the IDE or editor or from other sources such as, e.g., a console window.

Now we will ask a few background questions. Your responses will help us ensure that your experiences are within the scope of this study and will be kept confidential.

How many years have you worked in the software industry?

0 4 8 12 16 20 24 28 32 36 40

The following questions will refer to "changing software in Java or C#". You may answer every question based on the language you have used the most or are most proficient with. For example, if you have spent 1 year maintaining software in Java and 10 other years in C#, answer 11 years. If you spent 1 year maintaining both Java and C# software, answer 1 year.

How many years have you spent developing or maintaining software in Java or C# professionally?

0 4 8 12 16 20 24 28 32 36 40

How many of **the last ten years** (2010-2020) have you spent developing or maintaining software in Java or C# professionally?

0 1 2 3 4 5 6 7 8 9 10

For questions relating to programming environments or tools, you may answer based on both your Java and C# experience. For example, if you use the environment *foo* for Java code and *bar* for C# code, you may select both *foo* and *bar*.

Which of the following programming environments do you use when working on software in Java or C#?

- ☐ JetBrains IntelliJ

- ☐ Eclipse
- ☐ Netbeans
- ☐ Visual Studio
- ☐ Visual Studio Code
- ☐ JetBrains Rider
- ☐ Vim
- ☐ EMACS
- ☐ Other, please specify

Please rank your general proficiency with changing software in Java and C#.
(One star indicates no proficiency and five stars indicate expert proficiency.)

Java

C#

Demographic Questions

Demographic Questions

Based on your answers to the background questions, we welcome your participation in this study, **Tool Use in Software Change Tasks**.

On this page, we ask a few demographic questions. Your answers will allow us to compare data across different groups of respondents and will be kept confidential.

Which job title best represents your responsibilities right now?

- ○ Programmer / Software Developer / Software Engineer
- ○ System Administrator / Network Engineer
- ○ Project Manager
- ○ Technical Lead / Team Leader
- ○ Researcher
- ○ Other, please specify
- ○ Prefer not to answer

What is the highest level of school you have completed or the highest degree you have received?

- ○ Less than high school degree
- ○ High school graduate (high school diploma or equivalent including GED)
- ○ Some college or university but no degree
- ○ Bachelor's degree
- ○ Master's degree
- ○ Doctoral degree
- ○ Professional degree (JD, MD)
- ○ Prefer not to answer

What is your age?

- ○ Under 18
- ○ 18 - 24
- ○ 25 - 34
- ○ 35 - 44
- ○ 45 - 54
- ○ 55 - 64
- ○ 65 - 74
- ○ 75 - 84

- ○ 85 or older
- ○ Prefer not to answer

What is your gender?

- ○ Male
- ○ Female
- ○ Other
- ○ Prefer not to answer

Part 1 - Tool Experience

Tool Experiences

We will now ask about your experiences with using tools when changing software written in Java and C#.

We briefly reiterate the survey terminology.

> **Software change tasks** refer to applying changes to existing software for purposes such as adding or refining a feature, correcting a mistake, or improving code quality.
>
> We use the term **tools** to refer to programs that help support or automate the types of software change activities previously outlined: applying changes, inspecting changes, validating changes, and comprehending source code, test code, and documentation.

Please answer the following questions based on your experiences within the last ten years of changing software written in Java or C#.

The following is a list of tools that software developers may find useful during software change tasks. Please mark all the tools that you find useful during software change tasks.

- ☐ Test suites (e.g. JUnit, test errors)
- ☐ Version Control Systems "diffs" (e.g. git diff, et.c.)
- ☐ Structural navigation (e.g. find references, go to declaration)
- ☐ Copy/cut and paste code
- ☐ Debugging tools
- ☐ Simple Refactoring tools (e.g. Rename, Extract Constant, Inline Method)
- ☐ Compiler output (e.g. compiler errors)
- ☐ Complex Refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)
- ☐ Textual search (e.g. grep, find and replace)

Part 1 - Tool Experience - ranking

Here we list some statements that may or may not be true about the different tools.

For each tool, please mark **the statements that are necessary for that tool to be useful**.

You should select statements that are necessary, regardless of whether they are currently true or not. The following list contains a description of each statement.

- The tool is effective if it is reasonable to locate and apply for a desired intent.
- The tool saves time or effort if it is reasonable in terms of time and effort required to use.
- The tool is satisfying to use if it adds value to the development process.
- The tool is trustworthy if it feels safe or reliable to use, for example by rarely or never making errors or mistakes.
- The tool is predictable if it behaves consistently and predictably each time you use it.

	The tool is effective	The tool saves time or effort	The tool is satisfying to use	The tool is trustworthy	The tool is predictable
» Simple Refactoring tools (e.g. Rename, Extract Constant, Inline Method)	☐	☐	☐	☐	☐
» Complex Refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)	☐	☐	☐	☐	☐

	The tool is effective	The tool saves time or effort	The tool is satisfying to use	The tool is trustworthy	The tool is predictable
» Version Control Systems "diffs" (e.g. git diff, et.c.)	☐	☐	☐	☐	☐
» Compiler output (e.g. compiler errors)	☐	☐	☐	☐	☐
» Textual search (e.g. grep, find and replace)	☐	☐	☐	☐	☐
» Structural navigation (e.g. find references, go to declaration)	☐	☐	☐	☐	☐
» Copy/cut and paste code	☐	☐	☐	☐	☐
» Test suites (e.g. JUnit, test errors)	☐	☐	☐	☐	☐
» Debugging tools	☐	☐	☐	☐	☐

Part 2 - Scenario - intro

You will now be presented with one **software change scenario**.

You will be given a description of a **task** you need to solve, three **approaches** you may want to use, and a question about **tool use** in this task.

Please select the approach that is most similar to the approach you are likely yo take. If none of the approaches apply to you, select None of the above, and give a textual brief description of what you would do.

For the purpose of the scenario, put yourself in the following position.

You are acting as one of three developers of a utility library for Java core types. The source code is a Maven project comprising 78K lines of Java code and 1335 JUnit tests. It mainly consists of classes with static utility methods.

All public methods have one or more tests that specify their behavior. When considering the tasks you should consider changes to source code and test code but you may assume that appropriate release notes or other client-related actions has been handled.

For example, if your task is to remove a method **foo** from the library and replace it with a method **bar** that has somewhat different behavior, you need to remove **foo**, add **bar**, and update any internal references to foo, including JUnit tests like **testFoo**. You do NOT need to deprecate **foo**, consider release documents, client code, or other artifacts as long as other code in the library functions as before. You will be given descriptions about internal usages as test code as part of the task or approach descriptions.

If you wish to look at the source code or even attempt any of the tasks yourself, you may browse or download it from a GitHub repository here. It will open in a new window. **It is not necessary to do so in order to solve the tasks.**

Part 2 - Scenario - Remove Methods

Remove methods scenario

In this task, you should remove a few methods that are no longer useful. These methods are declared in a file of 8000 lines consisting solely of static methods and test methods are declared in its own (JUnit) test file.

Each method has one in-class caller and one test method. They all follow this pattern. Here we present an excerpt of the functional code to be changed (left) and the tests to be changed (right). In this example, isAnyNotEmpty should be removed.

Functional Code: StringUtils.java

```
public static boolean isAnyNotEmpty(final CharSequence... css) {
        if (ArrayUtils.isEmpty(css)) {
                return false;
        }
        for (final CharSequence cs : css) {
                if (isNotEmpty(cs)) {
                        return true;
                }
        }
        return false;
}
...
public static boolean isAllEmpty(final CharSequence... css) {
        return !isAnyNotEmpty(css);
}
```

Test Code: StringUtilsTest.java

```
@Test
public void testIsAnyNotEmpty() {
    assertFalse(StringUtils.isAnyNotEmpty((String) null));
    assertFalse(StringUtils.isAnyNotEmpty((String[]) null));
    assertTrue(StringUtils.isAnyNotEmpty(null, "foo"));
    assertTrue(StringUtils.isAnyNotEmpty("", "bar"));
    assertTrue(StringUtils.isAnyNotEmpty("bob", ""));
    assertTrue(StringUtils.isAnyNotEmpty("  bob  ", null));
    assertTrue(StringUtils.isAnyNotEmpty(" ", "bar"));
    assertTrue(StringUtils.isAnyNotEmpty("foo", "bar"));
    assertFalse(StringUtils.isAnyNotEmpty("", null));
}
```

```
@Test
public void testIsAllEmpty() {
    assertTrue(StringUtils.isAllEmpty());
    assertTrue(StringUtils.isAllEmpty(new String[]{}));
    assertTrue(StringUtils.isAllEmpty((String) null));
    assertTrue(StringUtils.isAllEmpty((String[]) null));
    assertFalse(StringUtils.isAllEmpty(null, "foo"));
    assertFalse(StringUtils.isAllEmpty("", "bar"));
    assertFalse(StringUtils.isAllEmpty("bob", ""));
    assertFalse(StringUtils.isAllEmpty("  bob  ", null));
    assertFalse(StringUtils.isAllEmpty(" ", "bar"));
    assertFalse(StringUtils.isAllEmpty("foo", "bar"));
    assertTrue(StringUtils.isAllEmpty("", null));
}
```

If you wish to see all the code in detail, click anywhere on the code or here to open it in a new window as a GitHub repository.

In this example, isAnyNotEmpty should be removed and isAllEmpty should stay. Then, you need to perform the same change to other methods.

Please select the workflow **is most similar to what you would do** in your day-to-day work to approach this scenario.

- ○ **I would:**
  1. Start by removing isAnyNotEmpty (e.g. by deleting or commenting out) to make compiler errors appear.
  2. Use the compiler errors to navigate to the code that is impacted (i.e., testIsAnyNotEmpy and isAllEmpty).
  3. Fix compiler errors (e.g. by removing testIsAnyNotEmpty and implementing isAllEmpty by undoing and moving code).
  4. Validate the changes by running tests.
- ○ **I would:**
  1. I would start by introducing a temporary new method that isAnyNotEmpty delegate to.
  2. I would validate this change by running tests.
  3. Then I would replace the call to isAnyNotEmpty to the temporary method. (e.g. return !isAnyNotEmpty(..) is replaced by return !tempMethod(..))
  4. I would validate this change by running tests.
  5. If the tests pass, I would remove testIsAnyNotEmpty and isAnyNotEmpty (e.g. by deleting or commenting out)
  6. Then I would validate the change by running tests.
  7. If the tests pass, I would inline the temporary method into isAllEmpty and remove it.
- ○ **I would:**
  1. Start by moving the implementation of isAnyNotEmpty into isAllEmpty.
  2. Validate this change by running testIsAllEmpty.
  3. If the test pass, I will remove testIsAnyNotEmpty and isAnyNotEmpty (e.g. by deleting or commenting out).
  4. Finally, I validate the changes by running tests.
- ○ **None of the above**, please specify

You chose ${q://QID84/ChoiceGroup/SelectedChoices}

You chose ${q://QID95/ChoiceTextEntryValue/10}

Which of the following tools would you use in this scenario?

- ☐ Safe Delete
- ☐ Extract Class
- ☐ Inline Constant
- ☐ Remove Parameter

- [ ] Test suite (e.g. test failures)
- [ ] Textual search (e.g. grep, find and replace)
- [ ] Change Signature
- [ ] Extract Constant
- [ ] Rename
- [ ] Inline Method
- [ ] Structural navigation (e.g. find references, go to declaration)
- [ ] Copy/cut and paste code
- [ ] Version Control Systems "diffs" (e.g. git diff, et.c.)
- [ ] Move Method
- [ ] Compiler output (e.g. compiler errors)
- [ ] Extract Method
- [ ] Other, please specify

Please select all the the statements that best match your reason for choosing this approach.

If none apply to you, please specify your reason.

- [ ] This approach lets me make the change stepwise.
- [ ] This approach lets me automate as much as possible.
- [ ] This approach lets me rely on the compiler to show me the steps.
- [ ] This approach lets me keep the tests running so I can validate changes.
- [ ] None of the above, please specify..

One way to solve this task would be to use the refactoring tool Inline Method to move the implementation of isAnyNotEmpty to isAllEmpty.

You did not select the Inline Method refactoring. Why would you not use the Inline Method refactoring in this scenario?

- [ ] The tool is not effective (it is not reasonable to locate and apply for a desired intent)
- [ ] The tool does not saves time or effort (it is not reasonable in terms of time and effort required to use)
- [ ] The tool is not satisfying to use (it does not add value to the development process)
- [ ] The tool is not trustworthy (it does not feel safe or reliable to use)
- [ ] The tool is not predictable (it does not behave consistently and predictably each time you use it)
- [ ] Other, please specify...

Part 2 - Scenario - Reorganize Methods

Reorganize test methods scenario

There is a large class (StringUtils, ~8000 LOC) with static helper methods for strings (e.g. isAllEmpty, isAllBlank). Each method in this file has one or more JUnit tests (e.g. testIsAllEmpty, testIsAllBlank), distributed across two test files: StringUtilsTest and StringUtilStrimEmptyTest.

Your task is to reorganize the tests such that test methods related to a particular functionality (Empty and Blank) is found in its own class such that it is easier to locate and run tests related to only this functionality.

You need to take around 8 methods from one class (StringUtilsTest) and 4 methods from another test class (StringUtilStrimEmptyTest) and put them into a new test file (new file, e.g. StringUtilsEmptyBlankTest). If there is any setup, variables, et.c., they should be brought along.

Please select the workflow **is most similar to what you would do** in your day-to-day work to approach this scenario.

- ○ **I would:**
  1. Start with creating a new empty class.
  2. Then I would move all the methods into the new class.
  3. I would fix any compiler errors (e.g. from any additional code).
  4. Then I would validate the change.
- ○ **I would:**
  1. Start with extracting the code I want and any necessary additional code from one of the files into a new class.
  2. Then I would extract the code I want and any necessary additional code from the other file into the new class.
  3. Then I would validate the change.
- ○ **I would:**
  1. Start with duplicating one of the classes (e.g. by copying and pasting).
  2. Then I would validate this change.
  3. Then I would remove the appropriate code from the original and the duplicate, so they each contain only the right code elements.
  4. Then I would validate this change.
  5. Once one class is finished, I would duplicate code from the other file into the new class (e.g. by copying and pasting).
  6. Then I would validate this change.
  7. Finally, I would remove code until all classes contain only the code I want and validate continously.
- ○ **None of the above**, please specify

You chose ${q://QID95/ChoiceGroup/SelectedChoices}

You chose ${q://QID95/ChoiceTextEntryValue/10}

Which of the following tools would you use in this scenario?

- ☐ Compiler output (e.g. compiler errors)
- ☐ Structural navigation (e.g. find references, go to declaration)
- ☐ Extract Class
- ☐ Textual search (e.g. grep, find and replace)
- ☐ Extract Method
- ☐ Change Signature
- ☐ Move Method
- ☐ Rename
- ☐ Version Control Systems "diffs" (e.g. git diff, et.c.)
- ☐ Inline Constant
- ☐ Extract Constant
- ☐ Test suite (e.g. test failures)
- ☐ Inline Method
- ☐ Safe Delete
- ☐ Copy/cut and paste code
- ☐ Remove Parameter
- ☐ Other, please specify

Please select all the the statements that best match your reason for choosing this approach. If none apply to you, please specify your reason.

- [ ] This approach lets me make the change stepwise.
- [ ] This approach lets me keep the tests running so I can validate changes.
- [ ] This approach lets me automate as much as possible.
- [ ] This approach lets me rely on the compiler to show me the steps.
- [ ] None of the above, please specify..

One way to solve this task would be to use the refactoring tool Move Method to move methods from one class to another. You did not select the Move Method refactoring. Why would you not use the Move Method refactoring in this scenario?

- [ ] The tool is not effective (it is not reasonable to locate and apply for a desired intent)
- [ ] The tool does not saves time or effort (it is not reasonable in terms of time and effort required to use)
- [ ] The tool is not satisfying to use (it does not add value to the development process)
- [ ] The tool is not trustworthy (it does not feel safe or reliable to use)
- [ ] The tool is not predictable (it does not behave consistently and predictably each time you use it)
- [ ] Other, please specify...

Part 2 - Scenario - Parameter Removal

Parameter Removal Scenario

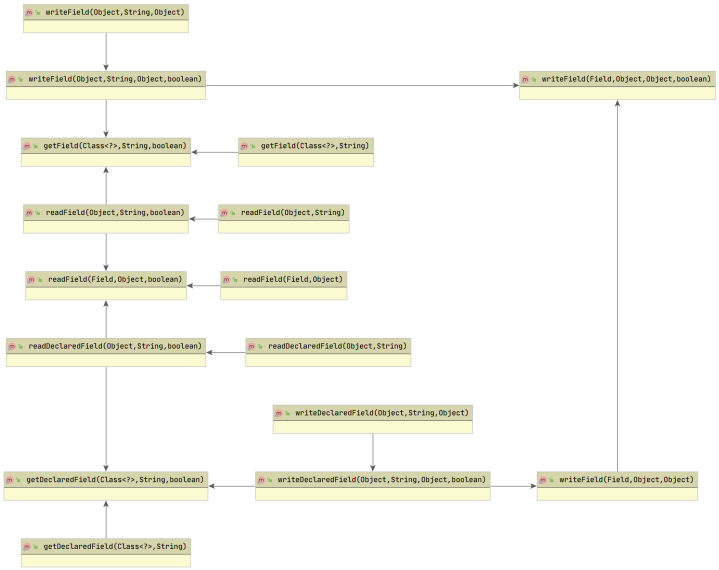
In this task, you need to make changes to a class FieldUtils, comprising 16 static helper methods for reading and writing fields, and to their test methods. Each method in the class has 1 or more JUnit tests. All methods in the class are only called by other methods in the class or by the test code.

The class consists of 8 method "pairs". Where the method without the forceAccess parameter, or flag exposes a default behavior, and the method with the parameter allows to access a special-case behavior. The default method delegates to the special-case method by calling it with the parameter set to false, as can be seen below.

```
public static Object readField(final Field field, final Object target) throws IllegalAccessException {
        return readField(field, target, false);
}
```

Your task is to remove the parameter. In order to do so, **all methods should be declared without the forceAccess parameter. Consequently, in-class callers must be updated, as must the tests**. (There are no callers outside the class).

In addition to the delegating calls between method "pairs", there are dependencies between the methods in the class, as can be seen in the following diagram.



The ability to force access should be removed. Consequently, the forceAccess parameter representing this flag should be removed and all callers and overloaded methods should be updated accordingly. During this process all 16 methods must be changed at least once

Please select the workflow **is most similar to what you would do** in your day-to-day work to approach this scenario.

- **I would:**
  1. Start by exploring the code structure and get an overview of callers and parameter references.
  2. Change one method and its callers at a time, attempting to resolve any compiler errors that appear, (e.g. by removing tests, or delete the default method or update code that references forceAccess, or consolidate the overloaded methods) before moving to the next method.
  3. Repeat for each method until all 8 methods are updated.
  4. Validate the change by running tests.
- **I would:**
  1. Go through the file top-down and move the implementation from each forceAccess-method into the default method (i.e. for all 8 methods before addressing compiler errors).
  2. Go through the file top-down again, and remove all usages of forceAccess in the method body (e.g. by locating compiler errors)
  3. Then I would update all callers and tests (e.g. by relying on compiler errors)
  4. Finally, I would validate the change by running the tests.
- **I would:**
  1. Start by assigning the default value of false to all forceAccess parameters inside all the special-case methods, so the class only performs the default functionality.
  2. Then I would run the tests to locate the ones that break and fix them (e.g. by deleting the ones that are no longer necessary).
  3. Once all tests pass, I would remove the forceAccess functionality (that is now unused) from all 8 methods.
  4. Then I validate the change using tests.
  5. Finally, I move all implementations into the default methods' bodies.
  6. Then I validate the change using tests.

○ **None of the above**, please specify

You chose ${q://QID108/ChoiceGroup/SelectedChoices}

You chose ${q://QID108/ChoiceTextEntryValue/10}

Which of the following tools would you use in this scenario?

- ☐ Compiler output (e.g. compiler errors)
- ☐ Remove Parameter
- ☐ Move Method
- ☐ Version Control Systems "diffs" (e.g. git diff, et.c.)
- ☐ Inline Method
- ☐ Test suite (e.g. test failures)
- ☐ Safe Delete
- ☐ Inline Constant
- ☐ Extract Constant
- ☐ Change Signature
- ☐ Copy/cut and paste code
- ☐ Rename
- ☐ Extract Method
- ☐ Extract Class
- ☐ Structural navigation (e.g. find references, go to declaration)
- ☐ Textual search (e.g. grep, find and replace)
- ☐ Other, please specify

Please select all the the statements that best match your reason for choosing this approach. If none apply to you, please specify your reason.

- ☐ This approach lets me make the change stepwise.
- ☐ This approach lets me keep the tests running so I can validate changes.
- ☐ This approach lets me rely on the compiler to show me the steps.
- ☐ This approach lets me automate as much as possible.
- ☐ None of the above, please specify..

One way to solve this task would be to use the refactoring tool Change Signature to remove the forceAccess parameter. You did not select the Change Signature refactoring. Why would you not use the Change Signature refactoring in this scenario?

- ☐ The tool is not effective (it is not reasonable to locate and apply for a desired intent)
- ☐ The tool does not saves time or effort (it is not reasonable in terms of time and effort required to use)
- ☐ The tool is not satisfying to use (it does not add value to the development process)
- ☐ The tool is not trustworthy (it does not feel safe or reliable to use)

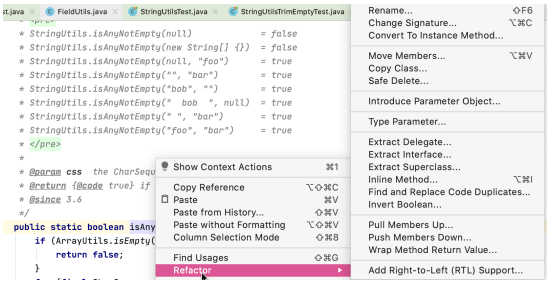
- [ ] The tool is not predictable (it does not behave consistently and predictably each time you use it)
- [ ] Other, please specify...

Refactoring Usability Issues

Refactoring Tool Usability

IDEs usually support numerous refactoring operations. On this page, we are interested in any problems you may have encountered when attempting to use such refactoring tools. We first provide a brief description of how they are invoked, before posing some problem statements that may or may not apply to your experience.

A refactoring operation can be accessed by selecting a *program element* (like a method, a variable, or a code selection) and invoking the refactoring either from the menu bar, by right-clicking on code, or using *hotkeys* (e.g. cmd+alt+R to Rename in Eclipse).



Once a refactoring operation is invoked on the program element, most IDEs will open an interactive wizard that lets you provide additional arguments (e.g. the name of an extracted method), *preview* the change (e.g. see all locations that will be changed), and a *problem* view that presents errors and warnings, and finally, a "*Continue*" or "Do Refactor" button that applies the refactoring operation to the source code.

With this workflow in mind, please consider the following statements describing usability problems that can occur during these steps. Please check any statements that describe situations you have encountered during your work.

- [ ] **The tool makes me lose control of my code or workflow.** This may be due to, for example, changing code you did not intend, making larger changes than you intended, hiding information that you will need later, etc.
- [ ] **The tool prevents me from validating the change.** This may be due to, for example, changing too many things at the same time, changing test code and source code simultaneously, not presenting a way to review changes, etc.
- [ ] **The tool does not help me find the next step.** This may be due to, for example, using terminology that you do not understand, not offering alternative steps when presenting problems or options, not communicating the implications of choices you make, etc.
- [ ] Prefer not to answer

If you use automated refactoring tools in any of the following programming environments, please rank your proficiency with the refactoring tools that are available in that programming environment.
(One star indicates no proficiency and five stars indicate expert proficiency.)

» JetBrains IntelliJ

» Eclipse

» Netbeans

» Visual Studio

» Visual Studio Code

» JetBrains Rider

» Vim

» EMACS

» Other, please specify

When you perform refactorings using an **automated tool**, which of the following tools do you use to **verify** that the change is correct? (Select all that applies. If you do nothing to verify, select "I do not verify the change")

- ☐ Version Control Systems "diffs" (e.g. git diff, et.c.)
- ☐ Compiler output (e.g. compiler errors)
- ☐ Textual search (e.g. grep, find and replace)
- ☐ Structural navigation (e.g. find references, go to declaration)
- ☐ Copy/cut and paste code
- ☐ Test suites (e.g. JUnit, test errors)
- ☐ Debugging tools
- ☐ Other, please specify..
- ☐ I do not verify the change

Please indicate the percentage of refactoring operations like the ones referred to in this study that you perform manually (e.g. using cut-and-paste or other simple tools) instead of using refactoring tools that are available to you?

	0	10	20	30	40	50	60	70	80	90	100
The percentage of % of refactoring operations performed manually instead of using available tools.											

When you choose to perform refactorings **manually** instead of using refactoring tools that are available to you, which of the following reasons impacts that choice?

- ☐ The tool is not effective (it is not reasonable to locate and apply for a desired intent)
- ☐ The tool does not saves time or effort (it is not reasonable in terms of time and effort required to use)
- ☐ The tool is not satisfying to use (it does not add value to the development process)
- ☐ The tool is not trustworthy (it does not feel safe or reliable to use)
- ☐ The tool is not predictable (it does not behave consistently and predictably each time you use it)
- ☐ Other, please specify...

When you perform refactorings **manually**, which of the following tools do you use to **verify** that the change is correct? (Select all that applies. If you do nothing to verify, select "I do not verify the change")

- ☐ Version Control Systems "diffs" (e.g. git diff, et.c.)
- ☐ Compiler output (e.g. compiler errors)
- ☐ Textual search (e.g. grep, find and replace)
- ☐ Structural navigation (e.g. find references, go to declaration)
- ☐ Copy/cut and paste code
- ☐ Test suites (e.g. JUnit, test errors)
- ☐ Debugging tools
- ☐ Other, please specify..
- ☐ I do not verify the change

When you were asked which statements was necessary for **Simple Refactoring tools (e.g. Rename, Extract Constant, Inline Method)** to be useful, these are the statements you selected.

Now, please select the statements **that you agree are true**. If you think none of these statements are true, select None.

None

	None
Simple Refactoring tools (e.g. Rename, Extract Constant, Inline Method)	☐

When you were asked which statements was necessary for **Complex Refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)** to be useful, these are the statements you selected.

Now, please select the statements **that you agree are true**. If you think none of these statements are true, select None.

	None
Complex Refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)	☐

If you have encountered any difficult situations when attempting to use refactoring tools in software change tasks, other than the ones we have asked about so far, please give a brief description here.

Prototype block

Here we present three types of functionality that could be added to refactoring tools.

For each type, please read the description and indicate whether it would make you more likely to use refactoring tools, it would have no change, or it would make you less likely to use tools.

When you apply the refactoring, the tool offers the ability to **step through** each change in the code editor as it happens, so that in each step you may view or edit the code in that location before continuing, somewhat akin to stepping through a program execution with a debugger tool.

- ○ This would make me **more** likely to use a refactoring tool.
- ○ This would make me **no more or less** likely to use a refactoring tool.
- ○ This would make me **less** likely to use a refactoring tool.

When you apply the refactoring, the tool offers the ability to **review a summary** after changing the code so that you may learn about the code that was changed and see why it was changed, somewhat akin to reviewing a project's version control history.

- ○ This would make me **more** likely to use a refactoring tool.
- ○ This would make me **no more or less** likely to use a refactoring tool.
- ○ This would make me **less** likely to use a refactoring tool.

When you apply the refactoring, the tool offers the ability to **revert or alter a refactoring you previously applied** at a later point in your workflow, even after applying other code changes. This is so that you may change, for example, configuration arguments or the type of refactoring to reflect information you garnered since applying it, somewhat akin to rebasing a project's version control history.

- ○ This would make me **more** likely to use a refactoring tool.
- ○ This would make me **no more or less** likely to use a refactoring tool.
- ○ This would make me **less** likely to use a refactoring tool.

When presented with the ability to **step through** each change in the code editor as it happens, you answered: ${q://QID139/ChoiceGroup/SelectedChoices} Please briefly describe why.

When presented with the ability to **review a summary after changing the code**, you answered: ${q://QID140/ChoiceGroup/SelectedChoices} Please briefly describe why.

When presented with the ability to **revert or alter a refactoring you previously applied at** a later point in your workflow, you answered: ${q://QID142/ChoiceGroup/SelectedChoices} Please briefly describe why.

End Questions

The questions on this page are optional. Once you click next, the survey will end.

Use this field to provide us with any comments or extra information that you think is relevant.

Use this field to provide us with feedback on your experience with this survey.

Bibliography

[1] Eclipse java refactoring. https://help.eclipse.org/2021-03/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm. [Online: accessed May 14th 2021].

[2] grep. https://www.gnu.org/software/grep/. [Online: accessed 12-October-2020].

[3] Intellij refactoring. https://www.jetbrains.com/help/idea/refactoring-source-code.html. [Online: accessed May 14th 2021].

[4] Software engineering — software life cycle processes — maintenance iso/iec 14764: 2006 (en), . URL https://www.iso.org/obp/ui/#iso:std:iso-iec:14764. (Accessed May 11th, 2021).

[5] Ergonomics of human–system interaction—part 11: Usability: Definitions and concepts iso 9241-11: 2018 (en), . URL https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-2:v1:en. (Accessed September 10th, 2020).

[6] Apache commons-lang. commons.apache.org/proper/commons-lang. [Online: accessed 25-August-2020].

[7] Maven. https://maven.apache.org/. [Online: accessed 28-August-2020].

[8] mlxtend library. https://rasbt.github.io/mlxtend. [Online: accessed 25-October-2020].

[9] Blinded replication package. Submitted to reviewers as additional files. The package will eventually be published online alongside the paper.

[10] Stack overflow survey 2020, professional developers demographic data. https://insights.stackoverflow.com/survey/2020#developer-profile-educational-attainment-professional-developers. [Online: Accessed March 25 2021].

[11] Apache commons-lang commit 0223a4d. https://github.com/apache/commons-lang/commit/0223a4d4cd127a1e209a04d8e1eff3296c0ed8c1, . [Online: accessed 24-July-2020].

[12] Apache commons-lang commit 3ce7f9e. https://github.com/apache/commons-lang/commit/3ce7f9eecfacbf3de716a8338ad4929371a66ca2, . [Online: accessed 24-July-2020].

[13] Apache commons-lang commit 3ce7f9e pull request. https://github.com/apache/commons-lang/pull/221, . [Online: accessed 24-July-2020].

[14] Quasar Commit 56d4b99. https://github.com/puniverse/quasar/commit/56d4b999e8be70be237049708f019c278c356e71, . [Online: accessed 24-July-2020].

[15] Trello. http://trello.com. [Online: accessed 25-October-2020].

[16] Visual studio refactoring documentation. https://docs.microsoft.com/en-us/visualstudio/ide/refactoring-in-visual-studio?view=vs-2019, note = "[Online: accessed 01-January-2021]".

[17] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig. 30 years of software refactoring research:a systematic literature review, 2020.

[18] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[19] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993. doi: 10.1145/170036.170072.

[20] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[21] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In D. N. Card, editor, *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*, pages 292–301. IEEE Computer Society, 1993. doi: 10.1109/ICSM.1993.366933.

[22] F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In *International Symposium on Formal Methods*, pages 492–507. Springer, 2006. doi: 10.1007/11813040_33.

[23] G. Bavota, A. De Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2010.11.918. URL `https://www.sciencedirect.com/science/article/pii/S0164121210003195`.

[24] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software*, 32(6):80–83, Nov 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.145.

[25] F. P. Brooks Jr. The mythical man-month (anniversary ed.), 1995.

[26] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[27] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. Jripples: A tool for program comprehension during incremental change. In *13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA*, pages 149–152. IEEE Computer Society, 2005. doi: 10.1109/WPC.2005.22.

[28] D. Campbell and M. Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-339-6. doi: 10.1145/1636642.1636651. URL `http://doi.acm.org/10.1145/1636642.1636651`.

[29] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, page 73–82, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342018. doi: 10.1145/2973839.2973848. URL `https://doi.org/10.1145/2973839.2973848`.

[30] M. O. Cinnéide and P. Nixon. Composite refactorings for java programs. In *Workshop on Formal Techniques for Java Programs, ECOOP*, pages 129–135. Citeseer, 2000.

[31] J. Corbin and A. Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[32] J. W. Cresswell and J. D. Creswell. *Research Design: Qualitative, Quantitative and Mixed Methods Approaches*. SAGE, 5 edition, 2018.

[33] L. Cruz, R. Abreu, and J.-N. Rouvignac. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In *Proceedings of the 4th International*

Conference on Mobile Software Engineering and Systems, pages 205–206. IEEE Press, 2017.

[34] P. DeGrace and L. H. Stahl. *Wicked problems, righteous solutions*. Yourdon Press, 1990.

[35] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006. ISSN 1532-0618. doi: 10.1002/smr.328.

[36] A. M. Eilertsen. Predictable, flexible or correct: Trading off refactoring design choices. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 330–333, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379632. doi: 10.1145/3387940.3392185. URL `https://doi.org/10.1145/3387940.3392185`.

[37] A. M. Eilertsen and G. C. Murphy. Data Package. `https://github.com/annaei/Replication-Data-for-The-Usability-or-Not-of-Refactoring-Tools/`. [Online: accessed 08-January-2021].

[38] A. M. Eilertsen and G. C. Murphy. Replication Data for: A Study of Refactorings During Software Change Tasks, 2021. URL `https://doi.org/10.18710/VTTNXM`.

[39] A. M. Eilertsen and G. C. Murphy. Stepwise refactoring tools. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 629–633. IEEE, 2021.

[40] A. M. Eilertsen and G. C. Murphy. The usability (or not) of refactoring tools. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 237–248. IEEE, 2021. doi: 10.1109/SANER50967.2021.00030.

[41] A. M. Eilertsen and G. C. Murphy. A study of refactorings during software change tasks. *Journal of Software: Evolution and Process*, Special Edition, 2021. doi: 10.1002/smr.2378. URL `https://doi.org/10.1002/smr.2378`.

[42] A. M. Eilertsen, A. H. Bagge, and V. Stolz. Safer refactorings. In *International Symposium on Leveraging Applications of Formal Methods*, pages 517–531. Springer, 2016. ISBN 978-3-319-47166-2. doi: 10.1007/978-3-319-47166-2_36.

[43] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):1–41, 2013.

[44] N. V. Flor and E. L. Hutchins. A case study of team programming during perfective software maintenance. In *Empirical studies of programmers: Fourth workshop*, volume 36. Intellect Books Norwood, NJ, 1991.

[45] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 222–232, June 2012. doi: 10.1109/ICSE.2012.6227191.

[46] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999. ISBN 0201485672.

[47] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 2018. ISBN 0134757599.

[48] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–30, 2020.

[49] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1095–1105, 2014.

[50] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337249`.

[51] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[52] L. A. Goodman. Snowball sampling. *The annals of mathematical statistics*, 32(1): 148–170, 1961. doi: 10.1214/aoms/1177705148.

[53] T. Green and A. Blackwell. Cognitive dimensions of information artefacts: a tutorial. In *BCS HCI Conference*, volume 98, pages 1–75, 1998.

[54] T. R. Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[55] W. G. Griswold. Program restructuring as an aid to software maintenance. 1992.

[56] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 284–293. IEEE Computer Society, 2004. doi: 10.1109/ICSM.2004.1357812.

[57] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 274–283. IEEE, 2005.

[58] A. Hindle, D. M. German, and R. Holt. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, page 99–108, 2008. doi: 10.1145/1370750.1370773.

[59] D. Jemerov. Implementing refactorings in intellij idea. In *Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–2, 2008.

[60] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[61] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606613. URL `https://doi.org/10.1109/ICSE.2013.6606613`.

[62] P. Kapur, B. Cossette, and R. J. Walker. *Refactoring References for Library Migration*, volume 45. ACM, New York, NY, USA, Oct. 2010. doi: 10.1145/1932682.1869518. URL `http://doi.acm.org/10.1145/1932682.1869518`.

[63] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 736–743, Nov 2001. doi: 10.1109/ICSM.2001.972794.

[64] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[65] B. W. Kernighan and P. J. Plauger. Software tools. *ACM SIGSOFT Software Engineering Notes*, 1(1):15–20, 1976.

[66] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving refactoring speed by 10x. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1145–1156. IEEE, 2016. doi: https://doi.org/10.1145/2884781.2884802.

[67] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393655. URL `http://doi.acm.org/10.1145/2393596.2393655`.

[68] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 126–135. ACM, 2005. doi: 10.1145/1062455.1062492. URL `https://doi.org/10.1145/1062455.1062492`.

[69] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Software Eng.*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116.

[70] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. *arXiv preprint arXiv:2108.11202*, 2021.

[71] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 23–32, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486792`.

[72] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A.-P. Tuovinen, and T. Männistö. Refactoring-a shot in the dark? *IEEE Software*, 32(6):62–70, 2015.

[73] H. Li and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 2:1–2:4, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-339-6. doi: 10.1145/1636642.1636644. URL `http://doi.acm.org/10.1145/1636642.1636644`.

[74] W. Liu and H. Liu. Major motivations for extract method refactorings: Analysis based on interviews and change histories. *Front. Comput. Sci.*, 10(4):644–656, Aug.

2016. ISSN 2095-2228. doi: 10.1007/s11704-016-5131-4. URL http://dx.doi.org/10.1007/s11704-016-5131-4.

[75] W. Maalej, R. Tiarks, T. Röhm, and R. Koschke. On the comprehension of program comprehension. In U. Aßmann, B. Demuth, T. Spitta, G. Püschel, and R. Kaiser, editors, *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, volume P-239 of *LNI*, pages 65–68. GI, 2015.

[76] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Australian Software Engineering Conference (ASWEC'06)*, pages 10–pp. IEEE, 2006.

[77] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth. Improving usability of software refactoring tools. In *2007 Australian Software Engineering Conference (ASWEC'07)*, pages 307–318. IEEE, 2007.

[78] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817.

[79] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *International Conference on Graph Transformation*, pages 286–301. Springer, 2002.

[80] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[81] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, May 2018. ISSN 0098-5589. doi: 10.1109/TSE.2017.2693982.

[82] E. Murphy-Hill. A model of refactoring tool use. *Proc. Wkshp. Refactoring Tools*, 2009. URL https://people.engr.ncsu.edu/ermurph3/papers/wrt09.pdf.

[83] E. Murphy-Hill. *Programmer Friendly Refactoring Tools*. PhD thesis, 2009.

[84] E. Murphy-Hill and A. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 421–430, May 2008. doi: 10.1145/1368088.1368146.

[85] E. Murphy-Hill and A. P. Black. High velocity refactorings in eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 1–5, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-015-9. doi: 10.1145/1328279.1328280. URL `http://doi.acm.org/10.1145/1328279.1328280`.

[86] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070529. URL `http://dx.doi.org/10.1109/ICSE.2009.5070529`.

[87] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. pages 287–297, 2009. doi: 10.1109/ICSE.2009.5070529. URL `http://dx.doi.org/10.1109/ICSE.2009.5070529`.

[88] E. Murphy-Hill, M. Ayazifar, and A. P. Black. Restructuring software with gestures. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 165–172, Sep. 2011. doi: 10.1109/VLHCC.2011.6070394.

[89] E. R. Murphy-Hill and A. P. Black. Why don't people use refactoring tools? In *Proc. of the 1st Workshop on Refactoring Tools*, pages 60–61, 2007.

[90] K. Narasimhan and C. Reichenbach. Copy and paste redeemed (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 630–640. IEEE, 2015.

[91] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*, pages 552–576. Springer, 2013. doi: 10.1007/978-3-642-39038-8_23.

[92] J. Oliveira, R. Gheyi, M. Mongiovi, G. Soares, M. Ribeiro, and A. Garcia. Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136 – 138, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.03.002. URL `http://www.sciencedirect.com/science/article/pii/S0950584919300461`.

[93] J. Oliveira, R. Gheyi, F. Pontes, M. Mongiovi, M. Ribeiro, and A. F. Garcia. Revisiting refactoring mechanics from tool developers' perspective. In G. Carvalho and V. Stolz, editors, *Formal Methods: Foundations and Applications - 23rd Brazilian Symposium, SBMF 2020, Ouro Preto, Brazil, November 25-27, 2020, Proceedings*, volume 12475 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2020. doi: 10.1007/978-3-030-63882-5_3.

[94] W. F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, 1992.

[95] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio. Behind the intents: An in-depth empirical study on software refactoring in modern code review. *17th MSR*, 2020.

[96] F. Palomba, A. Zaidman, R. Oliveto, and A. D. Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 176–185. IEEE Computer Society, 2017. doi: 10.1109/ICPC.2017.38. URL https://doi.org/10.1109/ICPC.2017.38.

[97] N. Pennington. Comprehension strategies in programming. In *Proc. of Second Workshop on the Empirical Studies of Programmers*, pages 100–112, 1987.

[98] P. Pirkelbauer, D. Dechev, and B. Stroustrup. Source code rejuvenation is not refactoring. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 639–650. Springer, 2010. doi: 10.1007/978-3-642-11266-9_53.

[99] V. Rajlich. Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 133–144. ACM, 2014. doi: 10.1145/2593882.2593893. URL https://doi.org/10.1145/2593882.2593893.

[100] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev. Refactoring with synthesis. In *ACM SIGPLAN Notices*, volume 48, pages 339–354. ACM, 2013.

[101] C. Reichenbach, D. Coughlin, and A. Diwan. Program metamorphosis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_18. URL http://dx.doi.org/10.1007/978-3-642-03013-0_18.

[102] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, Oct. 2004. ISSN 0362-1340. doi: 10.1145/1035292.1029012.

[103] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[104] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, Oct. 1997. ISSN 1074-3227. doi: 10.1002/(SICI)1096-9942(1997)3:4⟨253::AID-TAPO3⟩3.3.CO;2-I. URL http://dx.doi.org/10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.3.CO;2-I.

[105] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Champaign, IL, USA, 1999. AAI9944985.

[106] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869485. URL `http://doi.acm.org/10.1145/1869459.1869485`.

[107] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 277–294, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449787. URL `http://doi.acm.org/10.1145/1449764.1449787`.

[108] T. Sharma, G. Suryanarayana, and G. Samarthyam. Challenges to and solutions for refactoring adoption: An industrial perspective. *IEEE Software*, 32(6):44–51, Nov 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.105.

[109] J. Sillito, K. D. Volder, B. D. Fisher, and G. C. Murphy. Managing software change tasks: an exploratory study. In *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*, pages 23–32. IEEE Computer Society, 2005. doi: 10.1109/ISESE.2005.1541811. URL `https://doi.org/10.1109/ISESE.2005.1541811`.

[110] J. Sillito, K. D. Volder, B. D. Fisher, and G. C. Murphy. Managing software change tasks: an exploratory study. In *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*, pages 23–32. IEEE Computer Society, 2005.

[111] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006.

[112] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950305. URL `http://doi.acm.org/10.1145/2950290.2950305`.

[113] J. Singer, T. C. Lethbridge, N. G. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, Novem-*

ber 10-13, 1997, Toronto, Ontario, Canada, page 21. IBM, 1997. URL https://dl.acm.org/citation.cfm?id=782031.

[114] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984. doi: 10.1109/TSE.1984.5010283. URL https://doi.org/10.1109/TSE.1984.5010283.

[115] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 157–166. IEEE Computer Society, 2009. doi: 10.1109/ICSM.2009.5306335. URL https://doi.org/10.1109/ICSM.2009.5306335.

[116] F. Steimann. Constraint-based refactoring. *ACM Trans. Program. Lang. Syst.*, 40(1):2:1–2:40, Jan. 2018. ISSN 0164-0925. doi: 10.1145/3156016. URL http://doi.acm.org/10.1145/3156016.

[117] F. Steimann and J. von Pilgrim. Refactorings without names. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 290–293. IEEE, 2012.

[118] E. D. Tempero, T. Gorschek, and L. Angelis. Barriers to refactoring. *Commun. ACM*, 60(10):54–61, 2017. doi: 10.1145/3131873. URL https://doi.org/10.1145/3131873.

[119] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. *ACM SIGPLAN Notices*, 38(11):13–26, 2003.

[120] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. pages 174–181, Oct 1999. doi: 10.1109/ASE.1999.802203.

[121] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. pages 174–181, Oct 1999. doi: 10.1109/ASE.1999.802203.

[122] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[123] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, page 483–494. Association for Computing Machinery, 2018. ISBN 9781450356381. doi: 10.1145/3180155.3180206.

[124] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia. An empirical study on developer-related factors characterizing fix-inducing commits. *J. Softw. Evol. Process.*, 29(1), 2017. doi: 10.1002/smr.1797.

[125] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 31–38, 2011.

[126] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, June 2012. doi: 10.1109/ICSE.2012.6227190.

[127] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *European Conference on Object-Oriented Programming*, pages 527–551. Springer, 2013.

[128] M. Van Someren, Y. Barnard, and J. Sandberg. The think aloud method: a practical approach to modelling cognitive. *London: AcademicPress*, 1994.

[129] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995. doi: 10.1109/2.402076. URL `https://doi.org/10.1109/2.402076`.

[130] L. A. Wilson, Y. Senin, Y. Wang, and V. Rajlich. Empirical study of phased model of software change. *arXiv preprint arXiv:1904.05842*, 2019.

[131] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, COMPSAC 1978, 13-16 November, 1978, Chicago, Illinois, USA*, pages 60–65. IEEE, 1978. doi: 10.1109/CMPSAC.1978.810308.

Graphic design: Communication Division, UiB / Print: Skipnes Kommunikasjon AS

uib.no

ISBN: 9788230848067 (print)
9788230854334 (PDF)