Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

2-2009

Programmer Friendly Refactoring Tools

Emerson Murphy-Hill
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

Murphy-Hill, Emerson, "Programmer Friendly Refactoring Tools" (2009). *Dissertations and Theses.* Paper 2672.
https://doi.org/10.15760/etd.2671

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

DISSERTATION APPROVAL

The abstract and dissertation of Emerson Murphy-Hill for the Doctor of Philosophy in Computer Science were presented on February 26, 2009, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

____
Andrew P. Black, Chair

____
Stéphane Ducasse

____
Mark Jones

____
Susan Palmiter

____
Suresh Singh

____
Douglas Hall
Representative of the
Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:

____
Wu-chi Feng, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Emerson Murphy-Hill for the Doctor of Philosophy in Computer Science presented February 26, 2009.

Title: Programmer Friendly Refactoring Tools

Tools that perform semi-automated refactoring are currently under-utilized by programmers. If more programmers adopted refactoring tools, software projects could make enormous productivity gains. However, as more advanced refactoring tools are designed, a great chasm widens between how the tools must be used and how programmers want to use them. This dissertation begins to bridge this chasm by exposing usability guidelines to direct the design of the next generation of programmer-friendly refactoring tools, so that refactoring tools fit the way programmers behave, not vice-versa.

PROGRAMMER FRIENDLY REFACTORING TOOLS

by

EMERSON MURPHY-HILL

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
2009

To Tetey

Acknowledgements

This research could not have been accomplished without the help of countless others. First and foremost, thanks to my advisor, Andrew P. Black, for always providing enlightening guidance and advice. Thanks to the members of my thesis committee, each of whom contributed to this work: Stéphane Ducasse, Doug Hall, Mark Jones, Susan Palmiter, and Suresh Singh. Thanks to the National Science Foundation for partially funding this research under grant CCF-0520346. Thanks to Ken Brown at the Portland State Bookstore for donating gift cards as rewards for experiment participants. Thanks to the Computer Science department's staff for the continuous support and encouragement: Shiva Gudeti, Beth Holmes, Kathi Lee, Rene Remillard, and Barbara Sabath. Thanks to my research colleagues Chris Parnin and Danny Dig for their hard work during parts of this research. Thanks to Gail Murphy, Mik Kersten, Leah Findlater, Markus Keller, and Peter Weißgerber for use of their data. Thanks for Sergio Antoy, Andrew Black, Mark Jones, and Len Shapiro for inviting their students to participate in my experiments. Thanks to Robert Bauer, Paul Berry, Dan Brown, Cynthia Brown, Christian Bird, Tim Chevalier, Rob DeLine, Iavor Diatchki, Akshay Dua, Rafael Fernández-Moctezuma, Shiva Gudeti, Tom Harke, Anthony Hornof, Brian Huffman, Ed Kaiser, Rashawn Knapp, Jim Larson, Chuan-kai Lin, Ralph London, Bart Massey, Kathryn Mohror, Andrew McCreight, David Novick, Nick Pilkington, Philip Quitslund, Claudia Rocha, Suresh Singh, Tim Sheard, Jeremy Steinhauer, Aravind Subhash, Kal Toth, Eric Wheeler, Candy Yiu, and many anonymous reviewers for detailed, insightful criticism. Thanks to Barry Anderson, Robert Bow-

didge, Margaret Burnett, Jonathan Edwards, Joshua Kerievsky, Gregor Kiczales, Bill Opdyke, Bill Pugh, Jacek Ratzinger, and Vineet Sinha, Mathieu Verbaere, for their suggestions. Thanks to the participants of the Software Engineering seminar at UIUC for their suggestions. Special thanks to participants of my studies and interviews, without whom this research would have been impossible.

Contents

Acknowledgements **ii**

Contents **iv**

List of Tables **xi**

List of Figures **xiv**

1 A Roadmap **1**

2 Refactoring Theory **3**

2.1 Contributions 3

2.2 What is Refactoring? 3

2.3 When Should Programmers Refactor? 6

2.4 Refactoring Tools 7

2.5 A Model of How Programmers Use Refactoring Tools 13

2.6 The Structure of this Dissertation 15

3 Refactoring Practice **18**

3.1 Introduction 18

3.2 Contributions 19

3.3 The Data that We Analyzed 20

3.4 Findings on Refactoring Behavior 22

3.4.1 Toolsmiths and Users Differ . . . 22
3.4.2 Programmers Repeat Refactorings . . . 24
3.4.3 Programmers Often Do Not Configure Refactoring Tools . . 27
3.4.4 Commit Messages Do Not Predict Refactoring . . . 29
3.4.5 Floss Refactoring is Common . . . 33
3.4.6 Refactorings are Frequent . . . 35
3.4.7 Refactoring Tools are Underused . . . 36
3.4.8 Different Refactorings are Performed with and without Tools 40
3.5 Discussion . . . 41
3.5.1 Tool-Usage Behavior . . . 41
3.5.2 Detecting Refactoring . . . 42
3.5.3 Refactoring Practice . . . 43
3.5.4 Limitations of this Study . . . 44
3.5.5 Study Details . . . 45
3.6 Conclusions . . . 45

4 A Problem with Refactoring Tools **46**
4.1 Contributions . . . 46
4.2 Usability, Guidelines, and the Value of Guidelines Specific to Refactoring . . . 47
4.3 Why Usability is Important to Refactoring Tools . . . 48
4.4 Related Work . . . 49
4.5 An Exploratory Study of Refactoring . . . 50
4.5.1 The Extract Method Refactoring . . . 50
4.5.2 Methodology . . . 52
4.5.3 Results . . . 53
4.6 A Survey about Refactoring Behavior . . . 55
4.7 Usable Floss Refactoring Tools . . . 55

4.7.1 Principles for Tools that Support Floss Refactoring 56
4.7.2 Tools for Floss Refactoring 57

5 The Identification Step 60

5.1 Contributions 62
5.2 Guidelines and Related Work 63
5.2.1 Visualizations 63
5.2.2 Editor Annotations 67
5.3 Tool Description 70
5.3.1 Ambient View 70
5.3.2 Active View 73
5.3.3 Explanation View 74
5.3.4 Details of Stench Blossom 75
5.4 Evaluation 76
5.4.1 Subjects 77
5.4.2 Methodology 78
5.4.3 Results 80
5.4.3.1 Quantitative Results 80
5.4.3.2 How Smells were Identified *without* Stench Blossom 83
5.4.3.3 How Smells were Identified *with* Stench Blossom . 86
5.4.3.4 Suggestions for Tool Improvements 87
5.4.4 Threats to Validity 88
5.4.5 Discussion 91
5.5 Future Work 92
5.6 Conclusions 92

6 The Selection Step 93

6.1 Contributions 93

6.2 Tool Description 94
6.2.1 Selection Assist 94
6.2.2 Box View 95
6.3 Evaluation 96
6.3.1 Subjects 96
6.3.2 Methodology 97
6.3.3 Results 97
6.3.4 Threats to Validity 100
6.3.5 Discussion 101
6.4 Guidelines 102
6.5 Related Work: Alternative Selection Techniques 103
6.6 Generalization to Other Refactorings 104
6.6.1 A Running Example 105
6.6.2 Two More Selection Guidelines 105
6.6.3 Tool Description: Refactoring Cues 106
6.7 Conclusions 110

7 The Initiation Step **111**
7.1 Contributions 111
7.2 Guidelines 111
7.3 Related Work: Alternative Tool Initiation Techniques 114
7.4 Tool Description 114
7.5 Evaluation 117
7.5.1 Previous Studies: Pie Menus *vs.* Linear Menus 117
7.5.2 Memorability Study: Pie Menus with and without Placement Rules 119
7.5.2.1 Methodology 119
7.5.2.2 Subjects 122

7.5.2.3 Results 122
7.5.2.4 Threats to Validity 123
7.5.2.5 Discussion 124
7.5.3 Summary: A Comparison 124
7.6 Future Work 125
7.7 Conclusions 126

8 The Configuration Step 127

8.1 Contributions 127
8.2 Guidelines 128
8.3 Related Work: Alternative Configuration Techniques 130
8.4 Tool Description 130
8.5 Evaluations 131
8.5.1 Analytical Study: Refactoring Cues vs. Traditional Tools . . 131
8.5.1.1 Analysis by Stepwise Comparison 132
8.5.1.2 Threats to Validity 135
8.5.1.3 Discussion 136
8.5.2 Opinion Study: Pie Menus, Refactoring Cues, Hotkeys, and Linear Menus 136
8.5.2.1 Methodology 136
8.5.2.2 Subjects 138
8.5.2.3 Results 139
8.5.2.4 Threats to Validity 140
8.5.2.5 Discussion 140
8.5.3 Summary: A Comparison 140
8.6 Future Work 140
8.7 Conclusions 142

9 The Error Interpretation Step 143

9.1 Contributions 143
9.2 Tool Description 144
9.3 Evaluation 148
9.3.1 Subjects 148
9.3.2 Methodology 148
9.3.3 Results 149
9.3.4 Threats to Validity 151
9.3.5 Discussion 151
9.4 Guidelines 152
9.5 Related Work: Existing Research on Refactoring Errors 154
9.6 Generalization to Other Refactorings 155
9.6.1 A Taxonomy of Refactoring Preconditions 155
9.6.1.1 Methodology for Deriving a Precondition Taxonomy 156
9.6.1.2 Taxonomy Description 157
9.6.1.3 Application of the Remaining Guidelines to the Taxonomy 185
9.6.2 Evaluation 187
9.6.2.1 Subjects 188
9.6.2.2 Methodology 188
9.6.2.3 Example Experiment Run 192
9.6.2.4 Results 194
9.6.2.5 Discussion 198
9.6.2.6 Threats to Validity 199
9.7 Future Work 200
9.8 Conclusions 201

10 Conclusion 202

10.1 Summary of Contributions 202
10.2 Limitations 203
10.3 Future Work 204
10.4 The Thesis Statement 205

References **210**

List of Tables

3.1 Refactoring tool usage in Eclipse. Some tool logging began in the middle of the *Toolsmiths* data collection (shown in light grey) and after the *Users* data collection (denoted with a *). 23

3.2 The number and percentage of explicitly batched refactorings, for all Eclipse tool-based refactorings that support explicit batches. Some tool logging began in the middle of the *Toolsmiths* data collection (shown in light grey). 26

3.3 Refactoring tool configuration in Eclipse from *Toolsmiths*. 28

3.4 Refactoring between commits in *Eclipse CVS*. Plain numbers count commits in the given category; tuples contain the number of refactorings in each commit. 30

4.1 Preconditions to the EXTRACT METHOD refactoring, based on Opdyke's preconditions [58]. I have omitted preconditions that were not encountered during the formative study. 51

5.1 Some smell names and descriptions 61

5.2 Programming experience of subjects. 77

5.3 Post-experiment results regarding guidelines. 82

6.1 Total number of correctly selected and mis-selected `if` statements over all subjects for each tool. 98

6.2 Mean correct selection time over all subjects for each tool. 98

6.3 The number of times subjects used each tool to select `if` statements in each code set. 101

7.1 How refactorings can be initiated using Eclipse 3.3 and my current implementation of pie menus, in the order in which each refactoring appears on the system menu (Figure 7.1 on page 112); for pie menus, the direction in which the menu item appears is shown in the third column. I implemented the last three refactorings specifically for pie menus. . . 118

7.2 A comparison of initiation mechanisms for refactorings tools. 124

8.1 Advantages and disadvantages of pie menus and refactoring cues enumerated by the interviewer, labeled with + for advantage and – for disadvantage. 138

8.2 A comparison of selection and configuration mechanisms for refactoring tools. 141

9.1 The number and type of mistakes when finding problems during the EXTRACT METHOD refactoring over all subjects, for each tool, and the mean time to correctly identify all violated preconditions. Subjects diagnosed errors in a total of 64 refactorings with each tool. Smaller numbers indicate better performance. 149

9.2 A precondition taxonomy (left column), with counts of error messages in each taxonomy category for each refactoring tool (right columns). . . 159

9.3 In which order the four different groups of subjects used the two refactoring tools over the two code sets. 189

9.4 Refactorings and precondition violations used in the experiment. 190

9.5 The number and type of mistakes when diagnosing violations of refactoring preconditions, for each tool. The right-most column lists the total mean amount of time subjects spent diagnosing preconditions for all 8 refactorings. The asterisk (*) indicates that a timing was not obtained for one subject, so I could not include it in the mean. Subjects diagnosed errors in a total of 80 refactorings with each tool. Smaller numbers indicate better performance. 194

10.1 The guidelines postulated in this dissertation. **Step** indicates a step in the refactoring process (Section 2.5). **Guideline** states a postulated guideline and the page number where it was motivated. **Tools** lists my tools that implement that guideline and the page number where the tool was evaluated. 209

List of Figures

2.1 A stream class hierarchy in `java.io` (top, black) and a refactored version of the same hierarchy (bottom, black). In grey, an equivalent change is made in each version. 5
2.2 Selected code to be refactored in Eclipse. 9
2.3 A context menu in Eclipse. The next step is to select Extract Method... in the menu. 10
2.4 A configuration dialog asks you to enter information. The next step is to type "isSubnormal" into the Method name text box, after which the Preview > and OK buttons will become active. 11
2.5 A preview of the changes that will be made to the code. At the top, you can see a summary of the changes. The original code is on the left, and the refactored code on the right. You press OK to have the changes applied. 12
2.6 A model of how programmers use conventional refactoring tools. Steps outlined in black are the focus of this dissertation. 14

3.1 Percentage of refactorings that appear in batches as a function of batch threshold, in seconds. 60-seconds, the batch size used in Table 3.1 on page 23, is drawn in green. 26
3.2 Refactorings over 40 intervals. 32

3.3 Uses of Eclipse refactoring tools by 41 developers. Each column is labeled with the name of a refactorings performed using a tool in Eclipse, and the number of programmers that used that tool. Each row represents an individual programmer. Each box is labeled by how many times that programmer used the refactoring tool. The darker pink the interior of a box, the more times the programmer used that tool. Data provided courtesy of Murphy and colleagues [47]. 39

4.1 A code selection (above, highlighted in blue) that a tool cannot extract into a new method. 51

4.2 At the top, a method in `java.lang.Long` in an X-develop editor. At the bottom, the code immediately after the completion of the EXTRACT METHOD refactoring. The name of the new method is `m`, but the cursor is positioned to facilitate an immediate RENAME refactoring. 58

5.1 Examples of a smell visualization in Noseprints [62]. On the left, information about LONG METHOD for 3 classes, and on the right, information about LARGE CLASS for 3 other classes. This visualization appears inside of a window when the programmer asks the Visual Studio programming environment to find smells in a code base. Screenshots provided courtesy of Chris Parnin. 64

5.2 A compilation warning in Eclipse, shown as a squiggly line underneath program code. This line, for example, calls attention to the fact that this expression is being TYPECAST. 64

5.3 *Ambient View*, displaying the severity of several smells at the right of the editor. 71

5.4 *Active View*, where the programmer has placed the mouse cursor over a petal representing FEATURE ENVY to reveal the name of the smell and a clickable [+] to allow the programmer to transition to *Explanation View*. 73
5.5 *Explanation View*, showing details about the smell named in Figure 5.4 on page 73. 74

6.1 The Selection Assist tool in the Eclipse environment, shown covering the entire `if` statement, in green. The user's selection is partially overlaid, darker. 94
6.2 Box View tool in the Eclipse environment, to the left of the program code. 95
6.3 Mean time in seconds to select `if` statements using the mouse and keyboard versus Selection Assist (left) and Box View (right). Each subject is represented as a whole or partial X. The distance between the bottom legs represents the number of mis-selections using the mouse and keyboard. The distance between the top arms represents the number of mis-selections using Selection Assist (left) or Box View (right). Points without arms or legs represent subjects who did not make mistakes with either tool. 99
6.4 The several-step process of using refactoring cues. 108
6.5 Targeting several cues (the pink rectangles) at once using a single selection; the programmer's selection is shown by the grey overlay. 109

7.1 Initializing a refactoring from a system menu in Eclipse, with hotkeys displayed for some refactorings. 112
7.2 Two pie menus for refactoring, showing applicable refactorings for a method and temporary variable, respectively. 115

7.3 A sample training page (top) and a sample recall page (bottom). The refactorings (left, as program code before-and-after refactoring) are the same on both pages. Subjects were instructed to put a check mark in the appropriate direction on the recall page. 120
7.4 A histogram of the results of the pie menu experiment. Each subject is overlaid as one stick figure. Subjects from the experimental group who correctly guessed the refactoring that they did not see during training are denoted with a dashed oval. 123
7.5 A pie menu for refactoring with distance-from-center indicating what kind of configuration to perform. 125

8.1 Configuration gets in the way: an Eclipse configuration wizard obscures program code. 128
8.2 The user begins refactoring by selecting the `4` in X-develop, as usual (top). After initiating EXTRACT LOCAL VARIABLE (middle), the user types "ghostC" (bottom), using a linked in-line RENAME refactoring tool. 130
8.3 NGOMSL methods for conventional refactoring tools (top) and refactoring cues (bottom). 132

9.1 Refactoring Annotations overlaid on program code. The programmer has selected two lines of code (between the dotted lines) to extract. Here, Refactoring Annotations show how the variable will be used: `front` and `rear` will be parameters, as indicated by the arrows into the code to be extracted, and `trued` will be returned, as indicated by the arrow out of the code to be extracted. 144

9.2 Refactoring Annotations display an instance of a violation of refactoring precondition 1 (`goOnVacation`), precondition 2 (`curbHop`), and precondition 3 (`goForRide`), described in Table 4.1 on page 51. 146
9.3 For each subject, mean time to identify precondition violations correctly using error messages versus Refactoring Annotations. Each subject is represented as an X, where the distance between the bottom legs represents the number of imperfect identifications using the error messages and the distance between the top arms represents the number of imperfect identifications using Refactoring Annotations. 150
9.4 Illegal name violations, displayed normally in Eclipse (at left), and how such violations would be implemented following the guidelines (at right). The green violation indicators at right indicate that two invalid characters were typed into the new name text field. 166
9.5 Eclipse offering a quick-assist of all available return types in a refactoring dialog. 168
9.6 A mockup of how the guidelines inform the display of control unbinding (top and bottom left) and data unbinding (bottom right) for an attempted MOVE METHOD refactoring. The purple top annotation indicates that `isAttributeValueSupported(...)` calls this method, which is a problem because this method would not be visible outside in the destination. The `initMedia()` annotations indicate that this method calls the `initMedia()` method, which would not be visible from the destination. The `mediaPrintables` annotations indicate that this method uses the `mediaPrintables` field, which would not be visible from the destination. 173

9.7 A mockup of how the guidelines inform the display of name unbinding (in purple) and inheritance unbinding (in green) for an attempted MOVE METHOD refactoring, where the destination class is the class of `this_mon`. The purple `tranferQueue` annotations indicate that this method relies on a class `transferQueue`, which will not be accessible in the destination. The green `lookupTransferQueue` annotations indicate that the current method overrides a superclass method (top) and some subclass method (bottom), so the method cannot be moved. 175

9.8 A mockup of how the guidelines inform the display of control clash for an attempted RENAME METHOD refactoring, where the method at bottom has just been renamed to `isValid()` using Eclipse's in-line rename refactoring tool. At top, the existing method that the newly renamed method conflicts with, in a floating editor that can be used to perform recursive refactorings, such as renaming the original `isValid()` method. 177

9.9 A mockup of how the guidelines inform the display of context for an attempted MOVE METHOD refactoring, pointing out that the method `modalityPopped(...)` cannot be moved because interface methods cannot be moved. The original Eclipse modal error message states "Members in interfaces cannot be moved." 179

9.10 A mockup of how the guidelines inform the display of structure for an attempted CONVERT LOCAL TO FIELD refactoring, pointing out that the selected variable `originating_contact` is a parameter, which cannot be inlined. The original Eclipse modal error message states "Cannot convert method parameters to fields." 181

9.11 A mockup of how the guidelines inform the display of structure for an attempted INLINE CONSTANT refactoring, pointing out that the selected constant `theEnvironment` is blank, meaning that it is not assigned to at its declaration. The original Eclipse modal error message states “Inline Constant cannot inline blank finals.” 182

9.12 A mockup of how the non-local violations can be displayed in the program editor. Here, the variable `site_prefix` is referenced somewhere further down the editor. 184

9.13 An example of an experiment run. The experiment participant (at left), considers where to place a sticky note on the code responsible for the violation. The experiment administrator (at right), records observations about the participant’s reasoning regarding where he places the note. . . 193

Chapter 1

A Roadmap

Refactoring — the process of changing the structure of software without changing the way that it behaves — has been practiced by programmers for many years. More recently, tools that semi-automate the process of refactoring have emerged in various programming environments. These tools have promised to increase the speed at which programmers can write and maintain code while decreasing the likelihood that programmers will introduce new bugs. However, this promise remains largely unfulfilled, because programmers do not use the tools as much as they could. In this dissertation, I argue that one reason for this underuse is poor usability, meaning that the user interface of existing refactoring tools is sometimes too slow, too error-prone, and too unpleasant. I also take several steps to address the usability problem, guided by the following thesis statement:

> Applying a specified set of user-interface guidelines can help build more usable refactoring tools.

In this dissertation I explore the formation of those guidelines and the rationale behind them, as well as evaluate the effect that they have on refactoring tools' usability.

In Chapter 2, I introduce the concept of refactoring. In Chapter 3, I discuss how refactoring is actually practiced in the wild. In Chapter 4, I introduce usability, make the case that poor usability a problem with refactoring tools, and break down the

process of refactoring into individual steps. In Chapters 5 through 9, I look at five of these steps; I propose usability guidelines for each, reify those guidelines in the form of several novel user interfaces, and evaluate those user interfaces (and, indirectly, the guidelines that inspired them). Taken as a whole, I hope these new usability guidelines and tools will inform the next generation of refactoring tools, which will in turn more completely fulfill the tools' original promise.

Chapter 2

Refactoring Theory: Techniques and Tools [1]

In this chapter, I introduce previous work on the practice of refactoring and tools that perform refactoring semi-automatically. I also introduce my own distinction between two different tactics for refactoring — *floss* and *root canal refactoring*. I then propose five principles that characterize successful floss refactoring tools, five principles that can help programmers to choose the most appropriate refactoring tools and also help toolsmiths to design tools that fit the programmer's purpose.

2.1 Contributions

The major contributions of this chapter are:

- The distinction between, and description of, floss and root canal refactoring (Section 2.3), and
- A model of how programmers use conventional refactoring tools (Section 2.5).

2.2 What is Refactoring?

Refactoring is the process of changing the structure of software while preserving its external behavior, a practice described in early research by Opdyke and Johnson [59]

[1] Parts of this chapter appeared as part of a journal paper in *IEEE Software* [50].

and Griswold [23]. Later, it was popularized by Martin Fowler's book [22], but refactoring has been practiced for as long as programmers have been writing programs. Fowler's book is largely a catalog of refactorings; each refactoring captures a structural change that has been observed repeatedly in various programming languages and application domains.

Some refactorings make localized changes to a program, while others make more global changes. As an example of a localized change, when you perform Fowler's INLINE TEMP refactoring, you replace each occurrence of a temporary variable with its value. Taking a method from `java.lang.Long`,

```
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

you might apply the INLINE TEMP refactoring to the variable `offset`. Here is the result:

```
public static Long valueOf(long l) {
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + 128];
    }
    return new Long(l);
}
```

The inverse operation, in which you take the second of these methods and introduce a new temporary variable to represent 128, is also a refactoring, which Fowler calls INTRODUCE EXPLAINING VARIABLE. Whether the version of the code with or without the temporary variable is better depends on the context. The first version would be better if you were about to change the code so that `offset` appeared a second time; the second version might be better if you prefer more concise code.

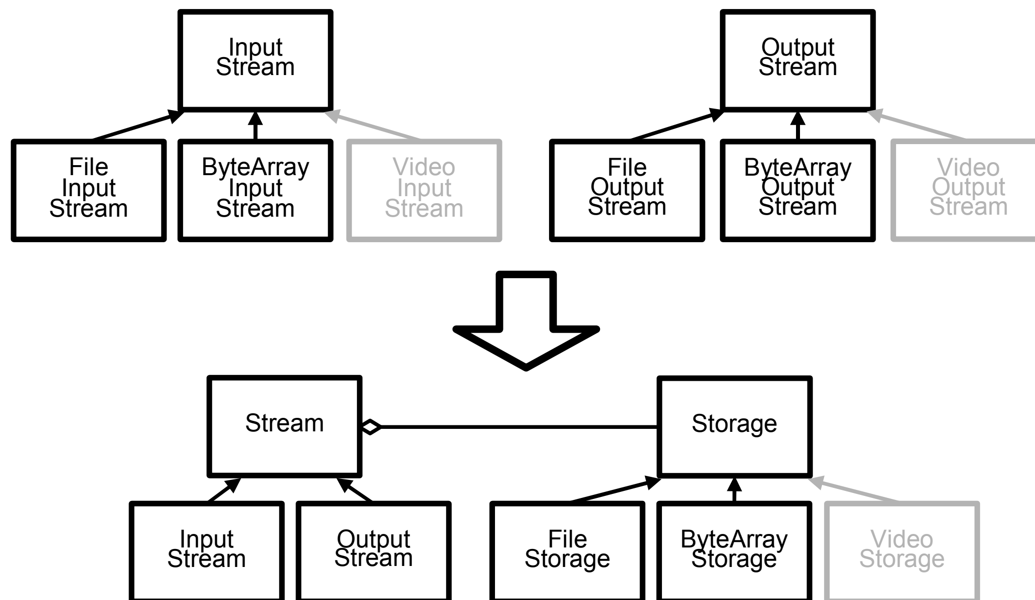


Figure 2.1: A stream class hierarchy in `java.io` (top, black) and a refactored version of the same hierarchy (bottom, black). In grey, an equivalent change is made in each version.

So, whether a refactoring improves your code depends on the context: you must still exercise good judgement.

Refactoring is an important technique because it helps you prepare to make semantic changes to your program. For example, to motivate a more global refactoring, suppose that you want to add the ability to read and write to a video stream to `java.io`. The relevant existing classes are shown in black at the top of Figure 2.1. Unfortunately, this top class hierarchy confounds two concerns: the direction of the stream (input or output) and the kind of storage that the stream works over (file or byte array). It would be difficult to add video streaming to the original `java.io` because you would have to add two new classes, `VideoInputStream` and `VideoOutputStream`, as shown by the grey boxes at the top of Figure 2.1. You would inevitably be forced to duplicate code between these two classes because their functionality would be similar.

Fortunately, you can separate these concerns by applying Fowler's TEASE APART INHERITANCE refactoring to produce the two separate stream and storage hierarchies shown in black at the bottom of Figure 2.1 on the preceding page. It is easier to add video streaming in the refactored version: all that you need do is add a class `VideoStorage` as a subclass of `Storage`, as shown by the grey box at the bottom of Figure 2.1 on the previous page. Because it enables software change, "Refactoring helps you develop code more quickly" [22, p. 57].

2.3 When Should Programmers Refactor?

On one hand, some experts have recommended refactoring in small steps, interleaving refactoring and writing code. For instance, Fowler states:

> In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. [22, p. 58]

Agile consultant Jim Shore has given similar advice:

> Avoid the temptation to stop work and refactor for several weeks. Even the most disciplined team inadvertently takes on design debt, so eliminating debt needs to be an ongoing activity. Have your team get used to refactoring as part of their daily work. [72]

On the other hand, the literature has also described a more heavyweight kind of refactoring, where programmers set aside specific time for refactoring planning and execution:

> Here, we want to use refactoring to improve a code base that has gone astray for several man-years without any noticeable rework in between!. . . This paper presented the results of a 5 months case study trying

> to imrove [sic] the quality of a commercial, medium size code base by refactoring. [63]

I call the first tactic **floss refactoring**, because the intent is to maintain healthy software by frequent refactoring, intermingled with other kinds of program changes. In contrast, I call the second tactic **root canal refactoring**. This is characterized by infrequent, protracted periods of refactoring, during which programmers perform few if any other kinds of program changes. You perform floss refactoring to maintain healthy code, and you perform root canal refactoring to correct unhealthy code. When I talk about refactoring tactics, I am referring to the choices that you make about how to mix refactoring with your other programing tasks, and about how frequently you choose to refactor.

I use the dental metaphor because, for many people, flossing one's teeth every day is a practice they know that they should follow, but which they sometimes put off. Neglecting to floss can lead to tooth decay, which can be corrected with a painful and expensive trip to the dentist for a root canal procedure. Likewise, a program that is refactored frequently and dutifully may be healthier and less expensive in the long run than a program whose refactoring is deferred until the most recent bug cannot be fixed or the next feature cannot be added. Like delaying dental flossing, the decision to delay refactoring may initially save time, but eventually may have painful consequences.

2.4 Refactoring Tools

Refactoring tools automate refactorings that you would otherwise perform with an editor.[2] Many popular development environments for a variety of languages — such as Eclipse [18], Microsoft Visual Studio [46], Xcode [31], and Squeak [21] — now include refactoring tools.

[2]When I say "editor," I mean a user-interface component with which you edit program text.

For example, suppose that you are the developer of the class `java.lang.Float`, and want to use refactoring tools in Eclipse to refactor code in that class. First, you choose the code you want refactored, typically by selecting it in an editor. In this example, you will choose the conditional expression in an `if` statement (Figure 2.2 on the following page) that checks to make sure that `f` is in subnormal form. Suppose that you want to put this condition into its own method so that you can give it an intention-revealing name and so that you can reuse it elsewhere in the `Float` class. After selecting the expression, you choose the desired refactoring from a menu. The refactoring that you want is labeled EXTRACT METHOD (Figure 2.3 on page 10).

The menu selection starts the refactoring tool, which brings up a dialog asking you to supply configuration options (Figure 2.4 on page 11). You have to provide a name for the new method: you will call it `isSubnormal`. You can also select some other options. You then have the choice of clicking OK, which would perform the refactoring immediately, or Preview >.

The preview page (Figure 2.5 on page 12) shows the differences between the original code and the refactored version. If you like what you see, you can click OK to have the tool apply the transformation. The tool then returns you to the editor, where you can resume your previous task.

Of course, you could have performed the same refactoring by hand: you could have used the editor to make a new method called `isSubnormal`, cutting-and-pasting the desired expression into the new method, and editing the `if` statement so that it uses the new method name. However, using a refactoring tool can have two advantages.

1. The tool is less likely to make a mistake than is a programmer refactoring by hand. In the example, the tool correctly inferred the necessary argument and return types for the newly created method, as well as deducing that the method should be static. When refactoring by hand, you can easily make mistakes on

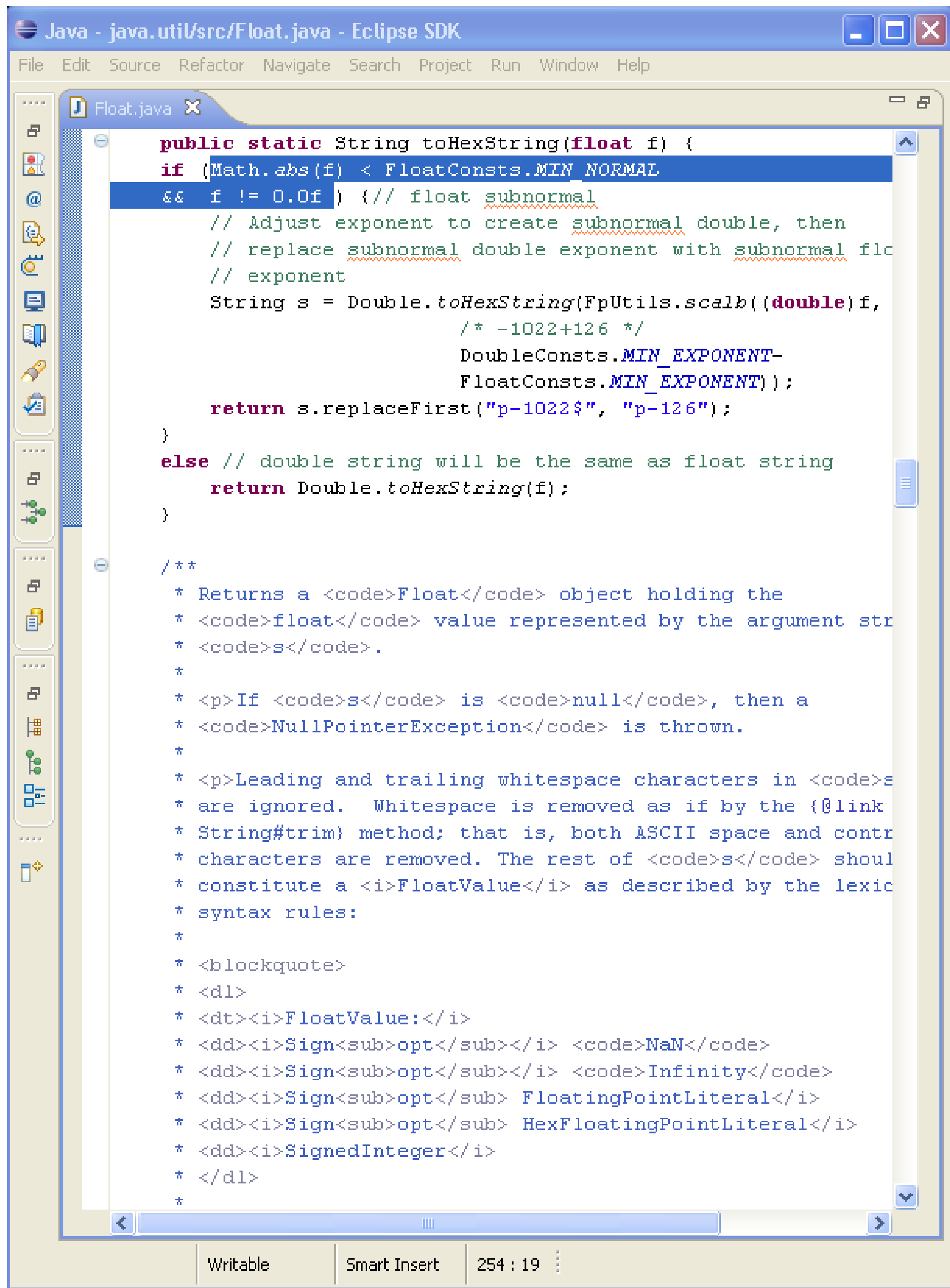


Figure 2.2: Selected code to be refactored in Eclipse.

Figure 2.3: A context menu in Eclipse. The next step is to select Extract Method... in the menu.

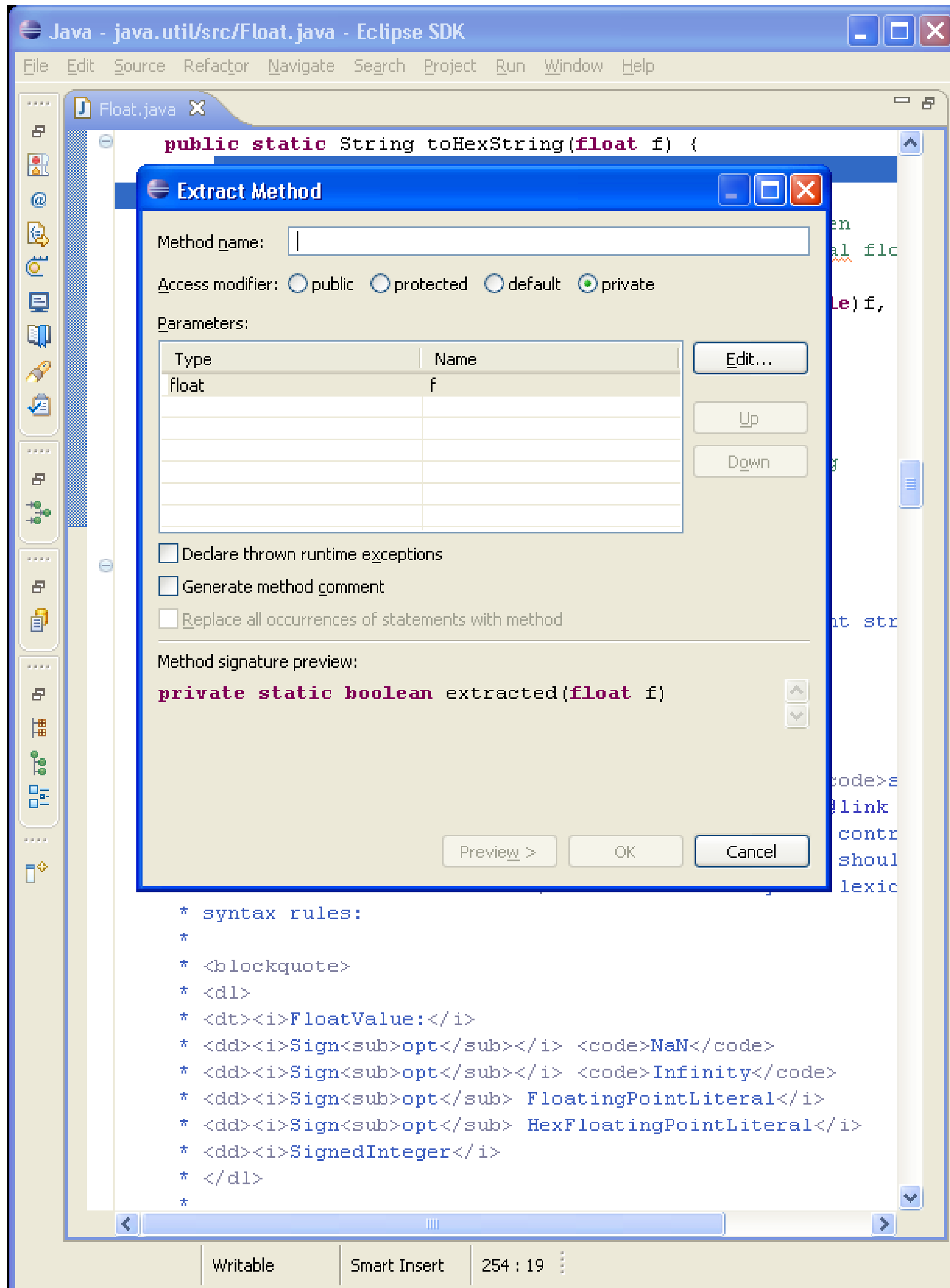


Figure 2.4: A configuration dialog asks you to enter information. The next step is to type "isSubnormal" into the Method name text box, after which the Preview > and OK buttons will become active.

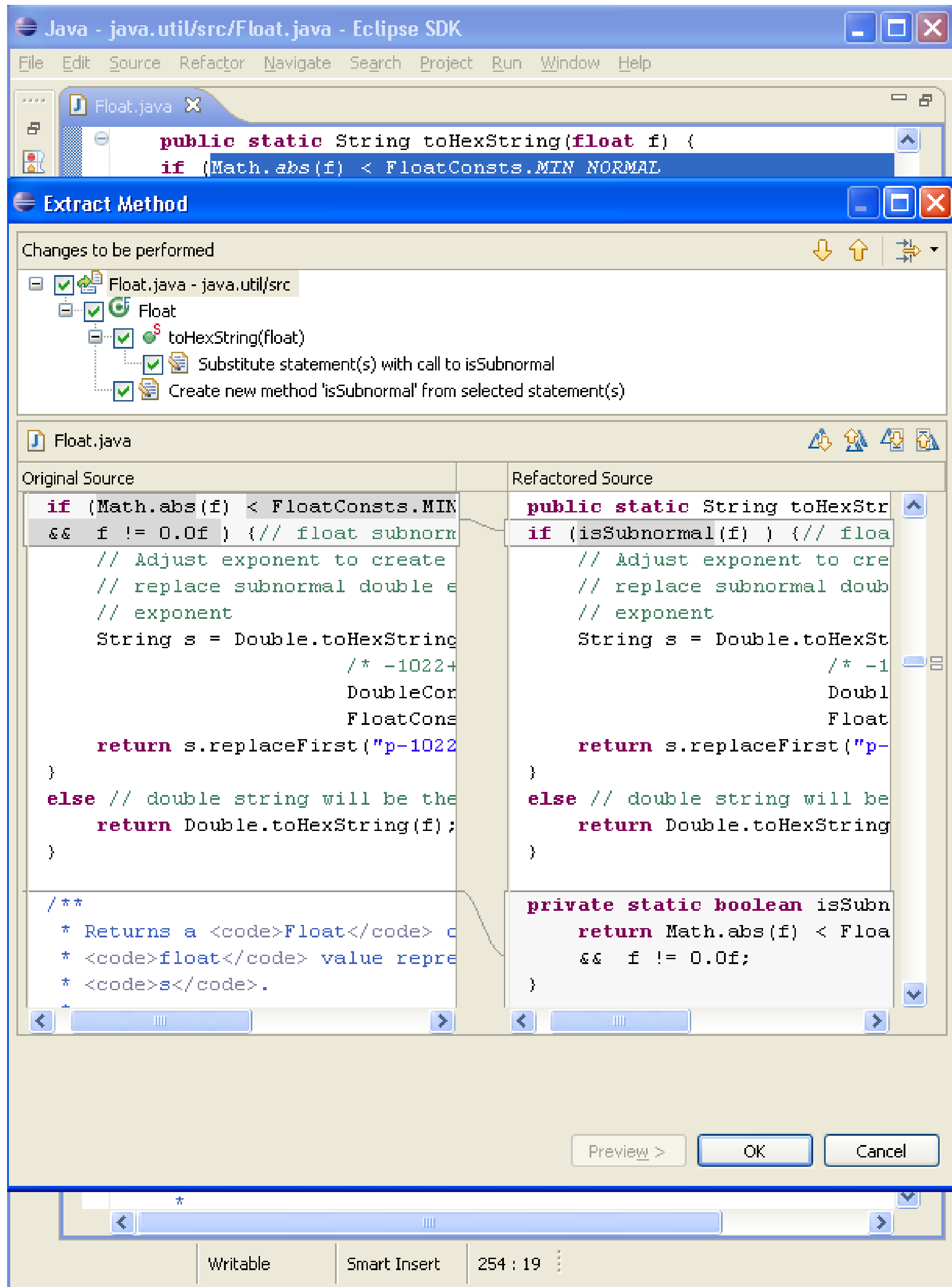


Figure 2.5: A preview of the changes that will be made to the code. At the top, you can see a summary of the changes. The original code is on the left, and the refactored code on the right. You press OK to have the changes applied.

such details.

2. The tool is faster than refactoring by hand. Doing it by hand, you would have to take time to make sure that you got the details right, whereas a tool can make the transformation almost instantly. Furthermore, refactorings that affect many locations throughout the source code, such as renaming a class, can be quite time-consuming to perform manually. They can be accomplished almost instantly by a refactoring tool.

In short, refactoring tools allow you to program faster and with fewer mistakes — but only if you choose to use them. Unfortunately, refactoring tools are not being used as much as they could be; the evidence for this claim is set out in Chapter 3. My goal is to make tools that programmers will choose to use more often. As a first step towards that goal, I next describe a model that I will use throughout this dissertation to speak more generally about how programmers use refactoring tools, without having to refer to specific tools or specific refactorings.

2.5 A Model of How Programmers Use Refactoring Tools

Figure 2.6 on the following page shows my model of how programmers use conventional refactoring tools. I started by examining Mealy and colleagues' 4-step model [45], Kataoka and colleagues' 3-step model [35], Fowler's description of small refactorings [22], and Lippert's description of large refactorings [38]. I expanded these simpler models into my new model by adding finer-grained steps, and the possibility of a recursive workflow, based my own observations of programmers refactoring. I have found this model useful both for reasoning about how programmers use refactoring tools and for improving the usability of those tools. However, while the model is meant to cover the most common refactoring tools, new tools are not com-

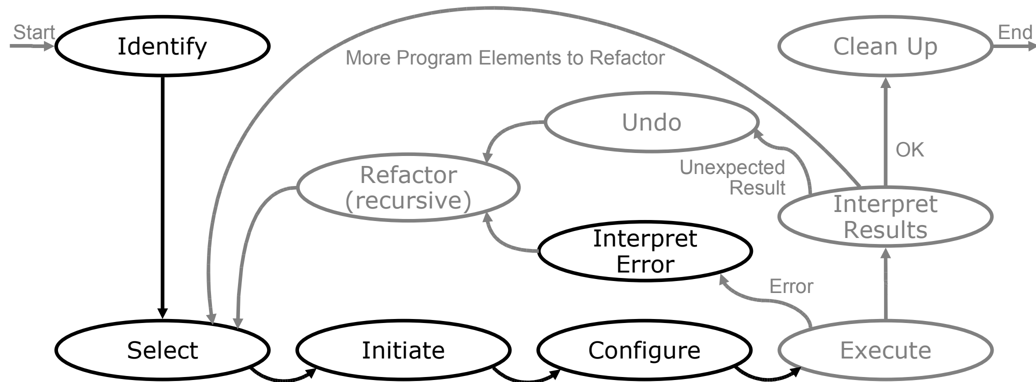


Figure 2.6: A model of how programmers use conventional refactoring tools. Steps outlined in black are the focus of this dissertation.

pelled to follow it; indeed, as I will show in Section 8.5, reordering or eliminating some of the steps can be beneficial.

I will explain the model by applying it to a simple refactoring. You begin by finding code that should be refactored (the Identify step). Then, you tell the tool which program element to refactor (Select), often by selecting code in an editor. You initiate the refactoring tool (Initiate), often by choosing the desired refactoring from a menu. You then give the tool some configuration information (Configure), such as by typing a new name into a dialog box. You signal the tool to actually transform the program (Execute), often by clicking an "OK" button in the dialog. You make sure that the tool performed the refactoring that you were expecting (Interpret Results). Finally, you may choose to perform some Clean Up refactorings. While not explicitly shown, you may abandon using the tool at any point, which corresponds to transitioning to a failure state from any step in the model.

The model also captures more complicated refactorings. When a precondition is violated, you typically must interpret an error message and choose an appropriate course of action (Interpret Error). When an unexpected result is encountered, you

may revert the program to its original state (Undo). You may recursively perform a sub-refactoring (Refactor) in order to make the desired refactoring successful. When you want to refactor several program elements at once, such as renaming several related variables, you must repeat the Select, Initiate, Configure, and Execute steps.

This model is a generalization: it describes how refactoring tools are typically used, but some programmers and specific tools may diverge from it in at least three ways. First, different tools provide different levels of support at each step. For instance, only a few tools help identify candidates for refactoring. Second, although the model defines a recursive refactoring strategy, a linear refactoring strategy is also possible. In a linear strategy, you perform sub-refactorings first, and avoid errors before they occur. I do not favor a strictly linear refactoring strategy because it requires foresight about what the tool *will* do, which I consider an unnecessary burden on programmers. In Section 4.5.3, I observe that such foresight — guessing what error messages a tool might produce — can lead programmers to avoid using a refactoring tool altogether. Third, some steps can be reordered or skipped entirely; for example, some tools provide a refactoring preview so that you may interpret the results of a refactoring before it is executed.

2.6 The Structure of this Dissertation

I have introduced refactoring and refactoring tools in this chapter, providing the necessary background to understand the remainder of the dissertation.

In Chapter 3, I will describe how programmers refactor in practice, based on data from programmers using existing refactoring tools, and on inspection of a code base where refactoring took place. Chapter 3 will lay the foundation of data for later propositions on how to improve refactoring tools. A central finding is that refactoring tools are underused, which means that the potential of refactoring tools is as yet unfulfilled.

In Chapter 4, I will argue that poor usability of current refactoring tools is a significant cause of underuse. This argument is based on existing research and on my own data on how programmers use — and do not use — refactoring tools.

Rather than finding and correcting a single usability problem with refactoring tools, I take a divide-and-conquer approach. Specifically, in each of the remaining chapters, I propose usability guidelines and new refactoring tool user interfaces for individual steps in my refactoring model (Section 2.5):

- In Chapter 5, I present how tools can more effectively help programmers *identify* code suitable for refactoring.
- In Chapter 6, I present how program elements can be more easily *selected* for refactoring.
- In Chapter 7, I present how the programmer can more easily *initiate* the refactoring she wants to perform.
- In Chapter 8, I present how *configuration* of refactoring tools can be made optional for the programmer.
- In Chapter 9, I present how the representation of refactoring *errors* can be improved.

Each of these Chapters 5–9 has a common set of components:

- In each chapter, I discuss related approaches and user interfaces for that refactoring step.
- I postulate new user interface guidelines to guide the construction of new refactoring tools that align with how programmers typically refactor.

- I describe a new user interface designed either (a) to address specific usability problems, or (b) to fit the postulated usability guidelines. Although my prototypes have been built for the Java programming language in the Eclipse environment, the techniques embodied in these interfaces should apply to other object-oriented and imperative programming languages and environments.
- Finally, I describe an evaluation of the proposed user interface, which forms an indirect evaluation of the guidelines embodied in the tool.

In Chapters 6 and 9, I first describe the tools that I created and then describe the guidelines that make them different from previous tools, whereas in Chapters 5, 7, and 8, I first postulate guidelines and then discuss how I implemented tools based on those guidelines. Ideally, I have learned, the latter ordering is preferable from a scientific standpoint; you have a hypothesis about what makes tools good, and then you test that hypothesis. I learned this halfway through the research described in this dissertation, and thus I describe orderings because that is the way my research was conducted.

The goal of this dissertation is to improve usability of refactoring tools by proposing usability guidelines combined with novel refactoring tool user interfaces, with the hope of increasing refactoring tool adoption and thus fulfilling the original productivity promise of refactoring tools.

Chapter 3

Refactoring Practice: What We Know and How We Know It [1]

In the last chapter, I discussed how refactoring has been prescribed by experts. In this chapter, I describe how my colleague Chris Parnin and I examined four data sets spanning more than 13 000 developers, 240 000 tool-assisted refactorings, 2500 developer hours, and 3400 version control commits. Using these data, I cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, I find that programmers frequently *do not* indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, I was able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes.

3.1 Introduction

In his book on refactoring, Fowler claims that refactoring produces significant benefits based on his own experience: it can help programmers to prepare to add functionality, fix bugs, and understand software [22, pp. 55-57]. Indeed, case studies have demonstrated that refactoring is a common practice [85] and that it can improve code metrics [5].

However, conclusions drawn from a single case study may not hold in general.

[1] Parts of this chapter are scheduled to appear as part of the *Proceedings of the 2009 International Conference on Software Engineering* [52].

Studies that investigate a phenomenon using a single research method also may not hold. To see why, one particular example that uses a single research method is Weißgerber and Diehl's study of three open source projects [84]. Their research method was to apply a tool to the version history of each project to detect high-level refactorings such as RENAME METHOD and MOVE CLASS. Low- and medium-level refactorings, such as RENAME LOCAL VARIABLE and EXTRACT METHOD, were classified as *non*-refactoring code changes. One of their findings was that, on every day on which refactoring took place, non-refactoring code changes also took place. What you can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If the latter are scarce, you can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then you cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this chapter, I use both of these methods to confirm — and disconfirm — several conclusions that have been published in the refactoring literature.

3.2 Contributions

In Section 3.3 I characterize the data that I used for this work. My experimental method takes data from four different sources (described in Section 3.3) and applies several different refactoring-detection strategies to them. I use this data to test eight hypotheses about refactoring. The contributions of my work lie in both the experimental method used when testing these hypotheses, and in the observations that I make about refactoring:

- The RENAME refactoring tool is used more frequently by tool-users than by toolsmiths (Section 3.4.1).
- About 40% of refactorings performed using a tool occur in batches (Section 3.4.2).
- About 90% of configuration defaults of refactoring tools remain unchanged when programmers use the tools (Section 3.4.3).
- Messages written by programmers in version histories are unreliable indicators of refactoring (Section 3.4.4).
- Floss refactoring, in which refactoring is interleaved with other types of programming activity, is used frequently (Section 3.4.5).
- Refactorings are performed frequently (Section 3.4.6).
- Almost 90% of refactorings are performed manually, without the help of tools (Section 3.4.7).
- The kind of refactoring performed with tools differs from the kind performed manually (Section 3.4.8).

In Section 3.5 I discuss the interaction between these conclusions and the assumptions and conclusions of other researchers.

3.3 The Data that We Analyzed

The work described in this chapter is based on four sets of data. The first set, which I will call *Users*, was originally collected in the latter half of 2005 by Murphy and colleagues [47], who used the Mylyn Monitor tool to capture and analyze fine-grained usage data from 41 volunteer programmers in the wild using the Eclipse development environment [18]. These data capture an average of 66 hours of development

time per programmer; about 95 percent of the programmers wrote in Java. The data include information on which Eclipse commands were executed, and at what time. Murphy and colleagues originally used these data to characterize the way programmers used Eclipse, including a coarse-grained analysis of which refactoring tools were used most often.

The second set of data, which I will call *Everyone*, is publicly available from the Eclipse Usage Collector [78], and includes data requested from every user of the Eclipse Ganymede release who consented to an automated request to send the data back to the Eclipse Foundation. These data aggregate activity from over 13 000 Java developers between April 2008 and January 2009, but also include non-Java developers. The data count how many programmers have used each Eclipse command, including refactoring commands, and how many times each command was executed. I know of no other research that has used these data for characterizing programmer behavior.

The third set of data, which I will call *Toolsmiths*, includes refactoring histories from four developers who maintain Eclipse's refactoring tools. These data include detailed histories of which refactorings were executed, when they were performed, and with what configuration parameters. These data include all the information necessary to recreate the usage of a refactoring tool, assuming that the original source code is also available. These data were collected between December 2005 and August 2007, although the date ranges are different for each developer. This data set is not publicly available and has not previously been described in the literature. The only study that I know of using similar data was published by Robbes [68]; it reports on refactoring tool usage by Robbes himself and one other developer.

The fourth set of data I will call *Eclipse CVS*, because it is the version history of the Eclipse and JUnit (http://junit.org) code bases as extracted from their Concurrent Versioning System (CVS) repositories. Specifically, Chris Parnin and I randomly

sampled from about 3400 source file commits (Section 3.4.4) that correspond to the same time period, the same projects, and the same developers represented in *Toolsmiths*. Using these data, we inferred which refactorings were performed by comparing adjacent commits manually. While many authors have mined software repositories automatically for refactorings (for example, Weißgerber and Diehl [84]), I know of no other research that compares refactoring tool logs with code histories.

3.4 Findings on Refactoring Behavior

In each of the following subsections, I describe a hypothesis about refactoring behavior; discuss why I suspect that the hypothesis is true; describe the results of an experiment that tests the hypothesis, using one or more of the data sets; and state the main limitations of the experiment. Each subsection heading briefly summarizes the subsection's findings.

3.4.1 Toolsmiths and Users Differ

I hypothesize that the refactoring behavior of the programmers who develop the Eclipse refactoring tools differs from that of the programmers who use them. Toleman and Welsh assume a variant of this hypothesis — that the designers of software tools erroneously consider themselves typical tool users — and argue that the usability of software tools should be evaluated objectively [81]. However, as far as I know, no previous research has tested this hypothesis, at least not in the context of refactoring tools. To do so, I compared the refactoring tool usage in the *Toolsmiths* data set against the tool usage in the *User* and *Everyone* data sets.

In Table 3.1 on the next page, the "Uses" columns indicate the total number of times each refactoring tool was invoked in that data set. The "Use %" column presents the same measure as a percentage of the total number of refactorings. Notice that while the rank order of each tool is similar across the three data sets —

Refactoring Tool	Toolsmiths				Users				Everyone	
	Uses	Use %	Batched	Batched %	Uses	Use %	Batched	Batched %	Uses	Use %
Rename	670	28.7%	283	42.2%	1862	61.5%	1009	54.2%	179871	74.8%
Extract Local Variable	568	24.4%	127	22.4%	322	10.6%	106	32.9%	13523	5.6%
Inline	349	15.0%	132	37.8%	137	4.5%	52	38.0%	4102	1.7%
Extract Method	280	12.0%	28	10.0%	259	8.6%	57	22.0%	10581	4.4%
Move	147	6.3%	50	34.0%	171	5.6%	98	57.3%	13208	5.5%
Change Method Signature	93	4.0%	26	28.0%	55	1.8%	20	36.4%	4764	2.0%
Convert Local To Field	92	3.9%	12	13.0%	27	0.9%	10	37.0%	1603	0.7%
Introduce Parameter	41	1.8%	20	48.8%	16	0.5%	11	68.8%	416	0.2%
Extract Constant	22	0.9%	6	27.3%	81	2.7%	48	59.3%	3363	1.4%
Convert Anonymous To Nested	18	0.8%	0	0.0%	19	0.6%	7	36.8%	269	0.1%
Move Member Type to New File	15	0.6%	0	0.0%	12	0.4%	5	41.7%	838	0.3%
Pull Up	12	0.5%	0	0.0%	36	1.2%	4	11.1%	1134	0.5%
Encapsulate Field	11	0.5%	8	72.7%	4	0.1%	2	50.0%	1739	0.7%
Extract Interface	2	0.1%	0	0.0%	15	0.5%	0	0.0%	1612	0.7%
Generalize Declared Type	2	0.1%	0	0.0%	4	0.1%	2	50.0%	173	0.1%
Push Down	1	0.0%	0	0.0%	1	0.0%	0	0.0%	279	0.1%
Infer Generic Type Arguments	0	0.0%	0	-	3	0.1%	0	0.0%	703	0.3%
Use Supertype Where Possible	0	0.0%	0	-	2	0.1%	0	0.0%	143	0.1%
Introduce Factory	0	0.0%	0	-	1	0.0%	0	0.0%	121	0.1%
Extract Superclass	7	0.3%	0	0.0%	*	-	*	*	558	0.2%
Extract Class	1	0.0%	0	0.0%	*	-	*	*	983	0.4%
Introduce Parameter Object	0	0.0%	0	-	*	-	*	*	208	0.1%
Introduce Indirection	0	0.0%	0	-	*	-	*	*	145	0.1%
Total	2331	100%	692	29.7%	3027	100%	1431	47.3%	240336	100%

Table 3.1: Refactoring tool usage in Eclipse. Some tool logging began in the middle of the *Toolsmiths* data collection (shown in light grey) and after the *Users* data collection (denoted with a *).

RENAME, for example, always ranks first — the proportion of occurrence of the individual refactorings varies widely between *Toolsmiths* and *Users/Everyone.* In *Toolsmiths*, RENAME accounts for about 29% of all refactorings, whereas in *Users* it accounts for about 62% and in *Everyone* for about 75%. I suspect that this difference is not because *Users* and *Everyone* perform more RENAMES than *Toolsmiths*, but because *Toolsmiths* are more frequent users of the other refactoring tools.

This analysis is limited in two ways. First, each data set was gathered over a different period of time, and the tools themselves may have changed between those periods. Second, the *Users* data include both Java and non-Java RENAME and MOVE refactorings, but the *Toolsmiths* and *Everyone* data report on just Java refactorings. This may inflate actual RENAME and MOVE percentages in *Users* relative to the other two data sets.

3.4.2 Programmers Repeat Refactorings

I hypothesize that when programmers perform a refactoring, they typically perform several refactorings of the same kind within a short time period. For instance, a programmer may perform several EXTRACT LOCAL VARIABLES in preparation for a single EXTRACT METHOD, or may RENAME several related instance variables at once. Based on personal experience and anecdotes from programmers, I suspect that programmers often refactor several pieces of code because several related program elements may need to be refactored in order to perform a composite refactoring. In Section 6.6.3, I describe a tool that allows the programmer to select several program elements at once, something that is not possible with traditional tools.

To determine how often programmers do repeat refactorings, I used the *Toolsmiths* and the *Users* data to measure the temporal proximity of refactorings to one another. I say that refactorings of the same kind that execute within 60 seconds of each another form a *batch*. From my personal experience, I think that 60 seconds is

long enough for a programmer to complete a typical Eclipse wizard-based refactoring, yet short enough to exclude refactorings that are not part of the same conceptual group. Additionally, a few refactoring tools, such as PULL UP in Eclipse, can refactor multiple program elements, so a single application of such a tool can be an explicit batch of related refactorings. For such tools, I counted the total number of tool uses that refactored only one program element (not an explicit batch of refactorings) and the number of tool uses that refactored more than one program element (an explicit batch of refactorings) in *Toolsmiths*.

In Table 3.1 on page 23, each "Batched" column indicates the number of refactorings that appeared as part of a batch, while each "Batched %" column indicates the percentage of refactorings appearing as part of a batch. Overall, you can see that certain refactorings, such as RENAME, INTRODUCE PARAMETER, and ENCAPSULATE FIELD, are more likely to appear as part of a batch for both *Toolsmiths* and *Users*, while others, such as EXTRACT METHOD and PULL UP, are less likely to appear in a batch. In total, you see that 30% of *Toolsmiths* refactorings and 47% of *Users* refactorings appear as part of a batch.[2] For comparison, Figure 3.1 on the next page displays the percentage of batched refactorings for several different batch thresholds.

In *Toolsmiths*, the number of explicit batches varied between tools (Table 3.2 on the following page). Although the total number of uses of these refactoring tools is fairly small, Table 3.2 suggests refactorings are batched about 25% of the time for tools that can refactor several program elements.

This analysis has two main limitations. First, while I wished to measure how often several related refactorings are performed in sequence, I instead used a 60-second heuristic. It is almost certain that some related refactorings occur outside my 60-second window, and that some unrelated refactorings occur inside the window.

[2] I suspect that the difference in percentages arises partially because the *Toolsmiths* data set counts the number of *completed* refactorings while *Users* counts the number of *initiated* refactorings. I have observed that programmers occasionally initiate a refactoring tool on some code, cancel the refactoring, and then re-initiate the same refactoring shortly thereafter (Section 4.5.3).

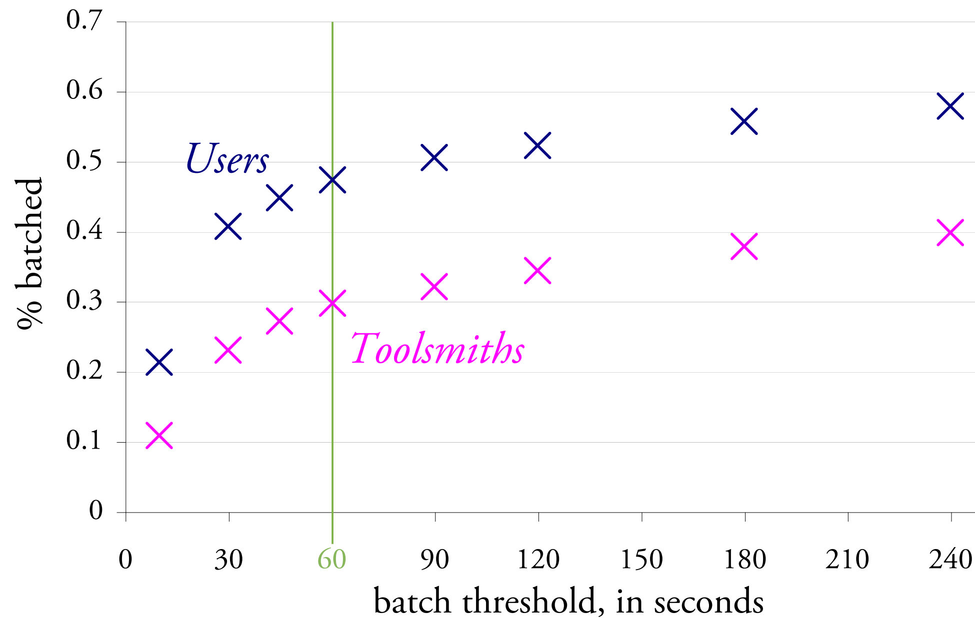


Figure 3.1: Percentage of refactorings that appear in batches as a function of batch threshold, in seconds. 60-seconds, the batch size used in Table 3.1 on page 23, is drawn in green.

Refactoring Tool	Uses	Explicitly Batched	Explicitly Batched %
MOVE	147	22	15.0%
PULL UP	12	11	91.6%
EXTRACT SUPERCLASS	7	6	85.7%
EXTRACT INTERFACE	2	1	50.0%
PUSH DOWN	1	1	100.0%
Total	169	42	24.8%

Table 3.2: The number and percentage of explicitly batched refactorings, for all Eclipse tool-based refactorings that support explicit batches. Some tool logging began in the middle of the *Toolsmiths* data collection (shown in light grey).

Other metrics for detecting batches should be investigated in the future. As a consequence, the percentage of refactorings that appear as part of a group is a *statistic* that only estimates the *population parameter* of interest: how often programmers repeat refactorings. Second, I could ascertain how often explicit batches are used in only the *Toolsmiths* data set: the other data sets are not sufficiently detailed.

3.4.3 Programmers Often Do Not Configure Refactoring Tools

Refactoring tools are typically of two kinds: either they force the programmer to provide configuration information, such as whether a newly created method should be `public` or `private`, or they perform a refactoring without allowing any configuration at all. Configurable refactoring tools are more common in some environments, such as Netbeans [53], whereas non-configurable tools are more common in others, such as X-develop [75]. Which interface is preferable depends on how often programmers configure refactoring tools. I hypothesize that programmers do not often configure refactoring tools. I suspect this because tweaking code manually after the refactoring may be easier than configuring the tool.

In the past, I have found some limited evidence that programmers perform only a small amount of configuration of refactoring tools. When I did a small survey in September 2007 at a Portland Java User's Group meeting, 8 programmers estimated that, on average, they supply configuration information only 25% of the time.

To validate this hypothesis, I analyzed the 5 most popular refactorings performed by *Toolsmiths* to see how often programmers used various configuration options. I skipped refactorings that did not have configuration options.

The results of the analysis are shown in Table 3.3 on the next page. "Configuration Option" refers to a configuration parameter that the user can change. "Default Value" refers to the default value that the tool assigns to that option. "Change %" refers to how often a user used a configuration option other than the default. The data suggest that refactoring tools are configured very little: the overall mean change percentage for these options is just under 10%. Although different configuration options are changed from defaults with varying percentages, all configuration options that I inspected were below the average configuration percentage predicted by the Portland Java User's Group survey.

This analysis has several limitations. First, I did not have detailed-enough infor-

Refactoring Tool	Configuration Option	Default Value	Change %
Extract Local Variable	Declare the local variable as ‘final’	false	5%
Extract Method	New method visibility	private	6%
	Declare thrown runtime exceptions	false	24%
	Generate method comment	false	9%
Rename Type	Update references	true	3%
	Update similarly named variables and methods	false	24%
	Update textual occurrences in comments and strings	false	15%
	Update fully qualified names in non-Java text files	true	7%
Rename Method	Update references	true	0%
	Keep original method as delegate to renamed method	false	1%
Inline Method	Delete method declaration	true	9%

Table 3.3: Refactoring tool configuration in Eclipse from *Toolsmiths*.

mation in the other data sets to cross-validate my results outside *Toolsmiths*. Second, I could not count how often certain configuration options were changed, such as how often parameters are reordered when EXTRACT METHOD is performed. Third, I examined only the 5 most-common refactorings; configuration may be more or less common for less popular refactorings.

3.4.4 Commit Messages Do Not Predict Refactoring

Several researchers have used messages attached to commits in a version control system, such as CVS, as indicators of refactoring activity [28, 66, 67, 76]. For example, if a programmer commits code to CVS and attaches the commit message "refactored class Foo," you might assume that the committed code contains more refactoring activity than if a programmer commits with a message that does not contain the word stem "refactor." However, I hypothesize that this assumption is false. I suspect this because refactoring may be an unconscious activity [9, p. 47], or because the programmer may consider it subordinate to some other activity, such as adding a feature [50].

In his dissertation, Ratzinger describes the most sophisticated strategy for finding refactoring messages of which I am aware [66]: searching for the occurrence of 13 keywords, such as "move" and "rename," and excluding "needs refactoring." Using two different project histories, the author randomly drew 100 file modifications from each project and classified each as either a refactoring or as some other change. He found that his keyword technique accurately classified modifications 95.5% of the time. Based on this technique, combined with a technique for finding bug fixes, Ratzinger and colleagues concluded that an increase in refactoring activity tends to be followed by a decrease in software defects [67].

Chris Parnin and I replicated Ratzinger's experiment for the Eclipse code base. Using the *Eclipse CVS* data, I grouped individual file revisions into global commits:

Change	Labeled	Unlabeled
Pure Whitespace	1	3
No Refactoring	8	11
Some Refactoring	5 (1,4,11,15,17)	6 (2,9,11,23,30,37)
Pure Refactoring	6 (1,1,2,3,3,5)	0
Total	20(63)	20(112)

Table 3.4: Refactoring between commits in *Eclipse CVS*. Plain numbers count commits in the given category; tuples contain the number of refactorings in each commit.

revisions were grouped if they were made by the same developer, had the same message, and were made within 60 seconds of each other. Henceforth, I use the word "revision" to refer to a particular version of a file, and the word "commit" to refer to one of these global commit groups. I then removed commits to CVS branches, which would have complicated my analysis, and commits that did not include a change to a Java file. Parnin and I also manually removed commits whose messages referred to changes to a refactoring tool (for example, "105654 [refactoring] CONVERT LOCAL VARIABLE TO FIELD has problems with arrays"), because such changes are false positives that occur only because the project is itself a refactoring tool project. Next, using Ratzinger's 13 keywords, I automatically classified the log messages for the remaining 2788 commits. 10% of these commits matched the keywords, which compares with Ratzinger's reported 11% and 13% for two other projects [66]. Next, we randomly drew 20 commits from the set that matched the keywords (which I will call "Labeled") and 20 from the set that did not match ("Unlabeled"). Without knowing whether a commit was in the Labeled or Unlabeled group, Parnin and I manually compared each committed version of Eclipse against the previous version, inferring how many and which refactorings were performed, and whether at least one non-refactoring change was made. Together, over about a 6 hour period, we did this comparison for the 40 commits using a single computer and the standard compare tool in Eclipse.

The results are shown in Table 3.4 on the preceding page. In the left column, the kind of Change is listed. “Pure Whitespace” means that the developer changed only whitespace or comments; “No Refactoring” means that the developer did not refactor but did change program behavior; “Some Refactoring” means that the developer both refactored and changed program behavior, and “Pure Refactoring” means the programmer refactored but did not change program behavior. The center column counts the number of Labeled commits with each kind of change, and the right column counts the number of Unlabeled commits. The parenthesized lists record the number of refactorings found in each commit. For instance, the Table shows that, in 5 out of 40 inspected commits, a programmer mentioned a refactoring keyword in the CVS commit message and made both functional and refactoring changes. The 5 commits contained 1, 4, 11, 15, and 17 refactorings.

These results suggest that classifying CVS commits by commit message does not provide a complete picture of refactoring activity. While all 6 pure-refactoring commits were identified by commit messages that contained one of the refactoring keywords, commits labeled with a refactoring keyword contained far fewer refactorings (63, or 36% of the total) than those not so labeled (112, or 64%). Figure 3.2 on the next page shows the variety of refactorings in Labeled (dark blue and purple) commits and Unlabeled (light blue and pink) commits.

There are several limitations to this analysis. First, while I tried to replicate Ratzinger’s experiment [66] as closely as was practicable, the original experiment was not completely specified, so I cannot say with certainty that the observed differences were not due to methodology. Likewise, observed differences may be due to differences in the projects studied. Indeed, after I completed this analysis, a personal communication with Ratzinger revealed that the original experiment included and excluded keywords specific to the projects being analyzed. Second, because the process of gathering and inspecting subsequent code revisions is labor intensive,

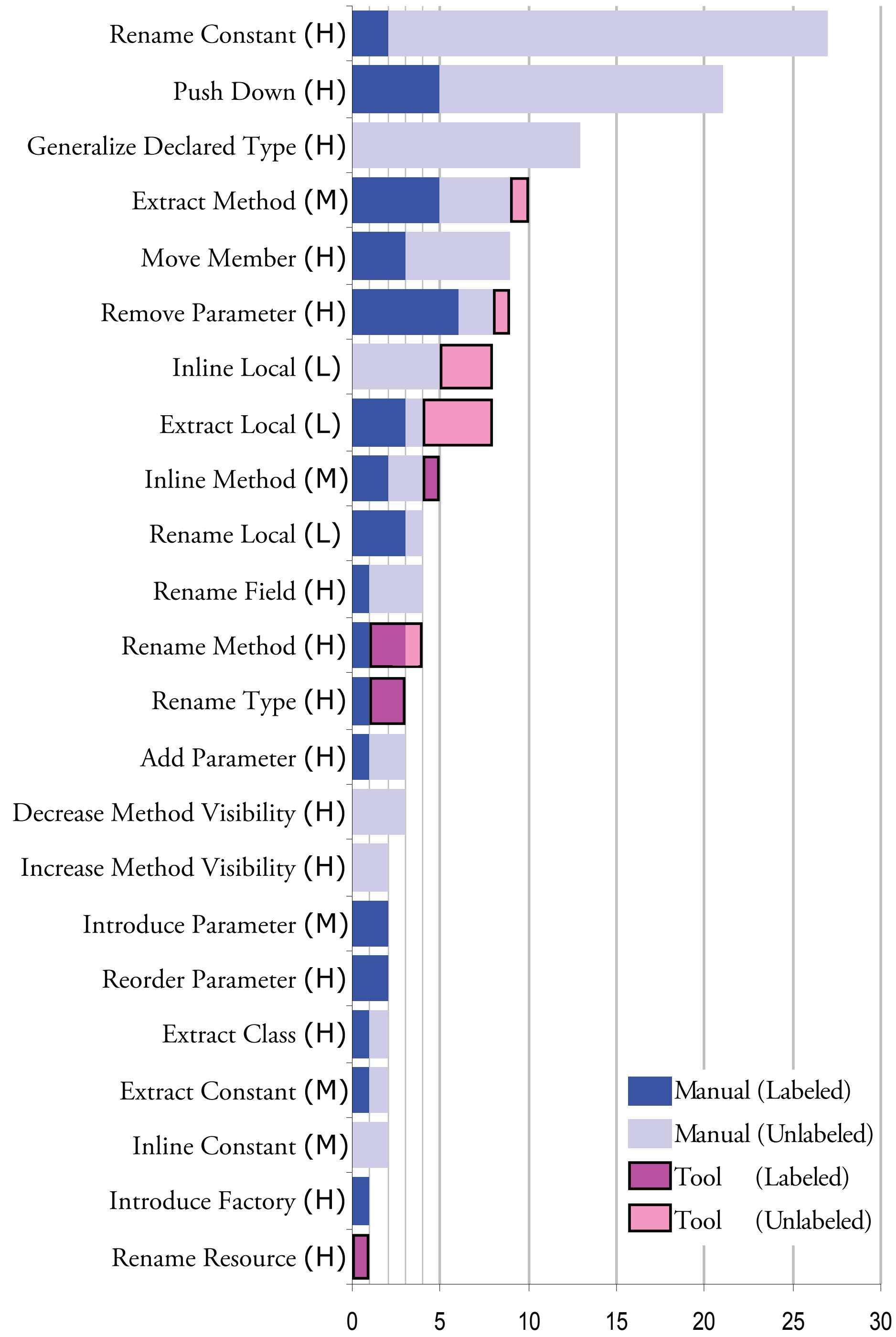


Figure 3.2: Refactorings over 40 intervals.

my sample size (40 commits in total) is smaller than would otherwise be desirable. Third, the classification of a code change as a refactoring is somewhat subjective. For example, if a developer removes code known to her to never be executed, then she may legitimately classify that activity as a refactoring, although to an outside observer it may appear to be the removal of a feature. Parnin and I tried to be conservative, classifying changes as refactorings only when we were confident that they preserved behavior. Moreover, because the comparison was blind, any bias introduced in classification would have applied equally to both Labeled and Unlabeled commit sets.

3.4.5 Floss Refactoring is Common

In Chapter 2.3, I introduced the distinction between floss and root canal refactoring. During floss refactoring, the programmer intersperses refactoring with other kinds of program changes to keep code healthy. Root-canal refactoring, in contrast, is used for correcting deteriorated code and involves a protracted process consisting of exclusive refactoring. A survey of the literature suggested that floss refactoring is the *recommended* tactic, but it did not provide evidence that it is the more *common* tactic.

Why does this matter? Case studies in the literature, for example those reported by Pizka [63] and by Bourqun and Keller [5], describe root-canal refactoring. However, inferences drawn from these studies will be generally applicable only if most refactorings are indeed root-canals.

I can estimate which refactoring tactic is used more frequently from the *Eclipse CVS* data. I first define behavioral indicators of floss and root-canal refactoring during programming intervals, which (in contrast to the intentional definitions given above) I can hope to recognize in the data. For convenience, let a programming interval be the period of time between consecutive commits to CVS by a single pro-

grammer. In a particular interval, if a programmer both refactors and makes a semantic change, then I say that that the programmer is floss refactoring. If a programmer refactors during an interval but does not change the semantics of the program, then I say that the programmer is root-canal refactoring. Note that a true root-canal refactoring must also last an extended period of time, or take place over several intervals. The above behavioral definitions relax this requirement and so will tend to over-estimate the number of root canals.

Returning to Table 3.4 on page 30, you can see that “Some Refactoring”, indicative of floss refactoring, accounted for 28% of commits, while “Pure Refactoring”, indicative of root-canal refactoring, accounts for 15%. Normalizing for the relative frequency of commits labeled with refactoring keywords in *Eclipse CVS*, commits indicating floss refactoring would account for 30% of commits while commits indicating root-canal would account for only 3% of commits.

Also notice in Table 3.4 on page 30 that the “Some Refactoring” (floss) row tends to show more refactorings per commit than the “Pure Refactoring” (root-canal) row. Again normalizing for labeled commits, 98% of individual refactorings would occur as part of a “Some Refactoring” (floss) commit, while only 2% would occur as part of a “Pure Refactoring” (root-canal) commit.

Pure refactoring with tools is infrequent in the *Users* data set, suggesting that very little root-canal refactoring occurred in *Users* as well. I counted the number of refactorings performed using a tool during intervals in that data. In no more than 10 out of 2671 commits did programmers use a refactoring tool without *also* manually editing their program. In other words, in less that 0.4% of commits did I observe the possibility of root-canal refactoring using only refactoring tools.

My analysis of Table 3.4 on page 30 is subject to the same limitations described in Section 3.4.4. The analysis of the *Users* data set (but not the analysis of Table 3.4) is also limited in that I consider only those refactorings performed using tools. Some

refactorings may have been performed by hand; these would appear in the data as edits, thus possibly inflating the count of floss refactoring and reducing the count of root-canal refactoring.

3.4.6 Refactorings are Frequent

While the *concept* of refactoring is now popular, it is not entirely clear how commonly refactoring is *practiced*. In Xing and Stroulia's automated analysis of the Eclipse code base, the authors conclude that "indeed refactoring is a frequent practice" [85]. The authors make this claim largely based on observing a large number of structural changes, 70% of which are considered to be refactoring. However, this figure is based on manually excluding 75% of semantic changes — resulting in refactorings that account for 16% of all changes. Further, their automated approach suffers from several limitations, such as the failure to detect low-level refactorings, imprecision when distinguishing signature changes from semantic changes, and the limited window of granularity offered by CVS inspection.

To validate the hypothesis that refactoring is a frequent practice, Parnin and I characterized the occurrence of refactoring activity in the *Users* and *Toolsmiths* data. In order for refactoring activity to be defined as frequent, I sought to apply criteria that require refactoring to be habitual and occurring at regular intervals. For example, if refactoring activity occurs just before a software release, but not at other times, then I would not want to claim that refactoring is frequent. First, Parnin examined the *Toolsmiths* data to determine how refactoring activity was spread throughout development. Second, Parnin and I examined the *Users* data to determine how often refactoring occurred within a programming interval and whether there was significant variation among the population.

In the *Toolsmiths* data, Parnin found that refactoring activity occurred throughout the Eclipse development cycle. In 2006, an average of 30 refactorings took place

each week; in 2007, there were 46 refactorings per week. Only two weeks in 2006 did not have any refactoring activity, and one of these was a winter holiday week. In 2007, refactoring occurred every week.

In the *Users* data set, Parnin and I found refactoring activity distributed throughout the programming intervals. Overall, 41% of programming intervals contained refactoring activity. More interestingly, intervals that did not have refactoring activity contained an order of magnitude fewer edits than intervals with refactoring, on average. The intervals that contained refactoring also contained, on average, 71% of the total edits made by the programmer. This was consistent across the population: 22 of 31 programmers had an average greater than 72%, whereas the remaining 9 ranged from 0% to 63%. This analysis of the *Users* data suggests that, when programmers must make large changes to a code base, refactoring is a common way to prepare for those changes.

Inspecting refactorings performed using a tool does not have the limitations of automated analysis; it is independent of the granularity of commits and semantic changes, and captures all levels of refactoring activity. However, it has its own limitation: the exclusion of manual refactoring. Including manual refactorings can only increase the observed frequency of refactoring. Indeed, this is likely: as you will see in Section 3.4.7, many refactorings are in fact performed manually.

3.4.7 Refactoring Tools are Underused

A programmer may perform a refactoring manually, or may choose to use an automated refactoring tool if one is available for the refactoring that she needs to perform. Ideally, a programmer will always use a refactoring tool if one is available, because automated refactorings are theoretically faster and less error-prone than manual refactorings. However, several pieces of existing data suggest that programmers do not use refactoring tools as much as they could:

- From my own observations, it appears that few programmers in an academic setting use refactoring tools. Of the 16 students who participated in the experiment described in Section 4.5.2, only 2 reported having used refactoring tools, and even then only for 20% and 60% of the time[3]. Furthermore, between September 2006 and December 2007, of the 42 people who used Eclipse on networked college computers, only 6 had tried Eclipse's refactoring tools.

- Professional programmers also appear not to use refactoring tools as much as they could. I surveyed 112 people at the Agile Open Northwest 2007 conference. I found that, on average, when a refactoring tool is available for a refactoring that programmers want to perform, they choose to use the tool 68% of the time[3]; the rest of the time they refactor by hand. Because agile programmers are often enthusiastic about refactoring, tool use by conventional (i.e., non-agile) programmers may be lower.

- When I compared predicted usage rates of two refactorings against the usage rates of the corresponding refactoring tools observed in the field, I found a surprising discrepancy. In a small experiment, Mäntylä and Lassenius [44] demonstrated that programmers wanted to perform EXTRACT METHOD more urgently, and several-fold more often, than RENAME. However, Murphy and colleagues' study [47] of 41 professional software developers provided data that suggest that Eclipse's EXTRACT METHOD tool is used significantly *less* often and by fewer programmers than its RENAME tool (Figure 3.3 on page 39). Comparing these two studies, I infer that some refactoring tools — the EXTRACT METHOD tool in this case — may be underused because the refactoring that programmers most *want* to perform is EXTRACT METHOD,

[3] The question's wording was ambiguous, so it is unclear whether respondents interpreted it as a percentage of time spent refactoring or as a percentage of uses of a refactoring tool, versus refactoring by hand

but the refactoring that they most *perform with tools* is RENAME.

These estimates of usage are surprisingly low, but they are still only estimates. I hypothesize that programmers often do not use refactoring tools. I suspect this because existing tools may not have a sufficiently usable user-interface.

To validate this hypothesis, I correlated the refactorings that Parnin and I observed by manually inspecting *Eclipse CVS* commits with the refactoring tool usages in the *Toolsmiths* data set. A refactoring found by manual inspection can be correlated with the application of a refactoring tool by looking for tool applications between commits. For example, the *Toolsmiths* data provides sufficient detail (the new variable name and location) to correlate an EXTRACT LOCAL VARIABLE with an EXTRACT LOCAL VARIABLE observed by manually inspecting adjacent commits in *Eclipse CVS*.

After analysis, I was unable to link 89% of 145 observed refactorings that had tool support to any use of a refactoring tool (also 89% when normalized). This suggests that *Toolsmiths* primarily refactor manually. An unexpected finding was that 31 refactorings that were performed with tools were not visible by comparing revisions in CVS. It appeared that most of these refactorings occurred in methods or expressions that were later removed or in newly created code that had not yet been committed to CVS. Overall, the results support the hypothesis that programmers are manually refactoring in lieu of using tools, but actual tool usage was lower than the median estimate in the professional agile developer survey. This suggests that either programmers overestimate their tool usage (perhaps refactoring is often not a conscious activity) or that expert programmers prefer to refactor manually.

This analysis suffers from two main limitations. First, some tool usage data may be missing. If programmers used multiple computers during development, some of which were not included in the data set, this would result in under-reporting of tool usage. Given a single commit, I can be more certain that I have a record of *all*

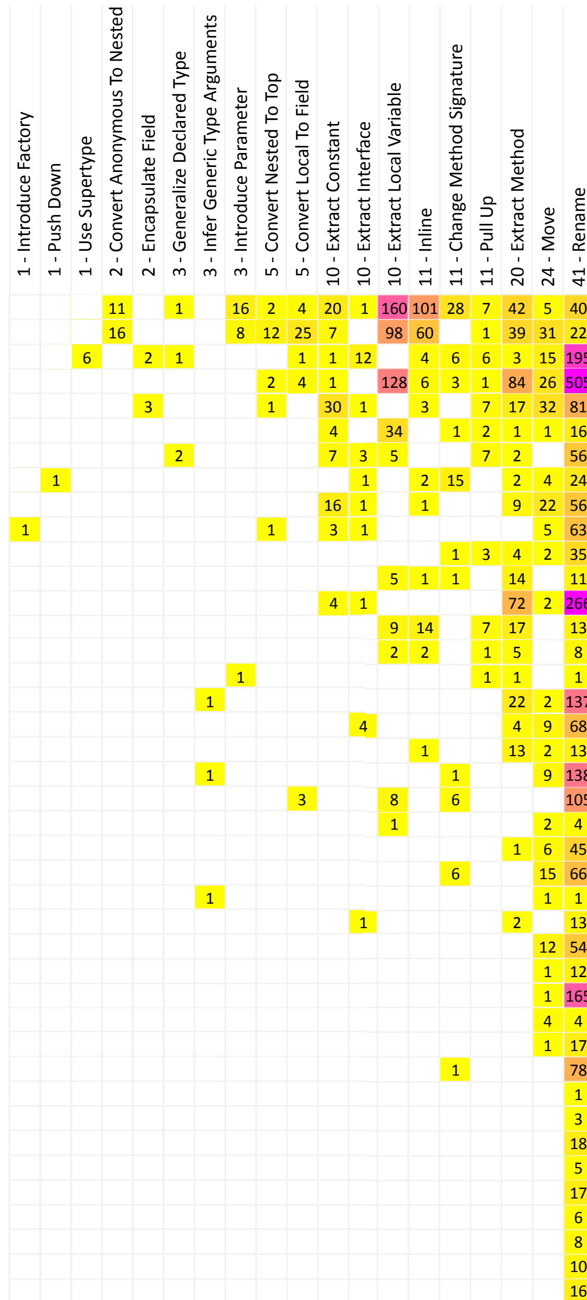


Figure 3.3: Uses of Eclipse refactoring tools by 41 developers. Each column is labeled with the name of a refactorings performed using a tool in Eclipse, and the number of programmers that used that tool. Each row represents an individual programmer. Each box is labeled by how many times that programmer used the refactoring tool. The darker pink the interior of a box, the more times the programmer used that tool. Data provided courtesy of Murphy and colleagues [47].

refactoring tool uses over code in that commit if there is a record of *at least one* refactoring tool use applied to that code since the previous commit. If I apply my analysis only to those commits, then 73% of refactorings (also 73% when normalized for the frequency of Labeled commits) cannot be linked with a tool usage. Second, refactorings that occurred at an earlier time might not be committed until much later; this would inflate the count of refactorings found in CVS that I could not correlate to the use of a tool, and thus cause me to underestimate tool usage. I tried to limit this possibility by looking back several days before a commit to find uses of refactoring tools, but I may not have been completely successful.

3.4.8 Different Refactorings are Performed with and without Tools

Some refactorings are more prone than others to being performed by hand. In Section 3.4.7, I inferred that the EXTRACT METHOD tool is underused: the refactoring is instead being performed manually. However, it is unclear what other refactoring tools are underused. Moreover, there may be some refactorings that must be performed manually because no tool yet exists. I suspect that the reason that some kinds of refactoring — especially RENAME — are more often performed with tools is because these tools have simpler user interfaces.

To validate this hypothesis, I examined how the kind of refactorings differed between refactorings performed by hand and refactorings performed using a tool. Again, I correlated the refactorings that Parnin and I found by manually inspecting *Eclipse CVS* commits with the refactoring tool usage in the *Toolsmiths* data. In addition, when inspecting the *Eclipse CVS* commits, Parnin and I identified several refactorings that currently have no tool support.

The results are shown in Figure 3.2 on page 32. Tool indicates how many refactorings were performed with a tool; Manual indicates how many were performed without. The figure shows that manual refactorings were performed much more of-

ten for certain kinds of refactorings. For example, EXTRACT METHOD is performed 9 times manually but just once with a tool; REMOVE PARAMETER is performed 8 times manually and once with a tool. However, a few kinds of refactoring show the opposite tendency; RENAME METHOD, for example, is most often performed with a tool. You can also see from the figure that many kinds of refactorings were performed exclusively by hand, despite having tool support.

Parnin and I also observed 30 refactorings that did not have tool support; the most popular of these was MODIFY ENTITY PROPERTY, performed 8 times, which would allow developers to modify properties, such as `static` or `final`, without changing behavior. The same limitations apply as in Section 3.4.7.

3.5 Discussion

How do the results presented in Section 3.4 affect future refactoring research and tools?

3.5.1 Tool-Usage Behavior

Several of my findings have reflected on the behavior of programmers using refactoring tools. For example, my finding about how toolsmiths differ from regular programmers in terms of refactoring tool usage (Section 3.4.1) suggests that most kinds of refactorings will not be used as frequently as the toolsmiths hoped, when compared to the frequently used RENAME refactoring. For the toolsmith, this means that improving underused tools (or their documentation), especially tools for EXTRACT LOCAL VARIABLE, may increase tool use.

Other findings provide insight into the typical work flow involved in refactoring. Consider that refactoring tools are often used repeatedly (Section 3.4.2), and that programmers often do not configure refactoring tools (Section 3.4.3). For the toolsmith, this means that configuration-less refactoring tools, which have recently seen

increasing support in Eclipse and other environments, will suit the majority of, but not all, refactoring situations. In addition, my findings about the batching of refactorings provides evidence that tools that force the programmer to repeatedly select, initiate, and configure can waste programmers' time.

Questions still remain for researchers to answer. *Why* is the RENAME refactoring tool so much more popular than other refactoring tools? Why do some refactorings tend to be batched while others do not? Moreover, my experiments should be repeated in other projects and for other refactorings to confirm or disconfirm my findings.

3.5.2 Detecting Refactoring

In my experiments I have investigated the assumptions underlying several commonly used refactoring-detection techniques. Unfortunately, some techniques may need refinement to address the concerns that I have uncovered. My finding that commit messages in version histories are unreliable indicators of refactoring activity (Section 3.4.4) is at variance with an earlier finding by Ratzinger [66]. It also casts doubt on previous research that relies on this technique [28, 67, 76]. Thus, further replication of this experiment in other contexts is needed to establish more conclusive results. My finding that many refactorings are medium or low-level suggests that refactoring-detection techniques used by Weißgerber and Diehl [84], Dig and colleagues [12], Counsell and colleagues [10], and to a lesser extent, Xing and Stroulia [85], will not detect a significant proportion of refactorings. The effect that this has on the conclusions drawn by these authors depends on the scope of those conclusions. For example, Xing and Stroulia's conclusion that refactorings are frequent can only be bolstered when low-level refactorings are taken into consideration. On the other hand, Dig and colleagues' tool was intended to help upgrade library clients automatically, and thus has no need to find low-level refactorings. In general, re-

searchers who wish to detect refactorings automatically should be aware of what level of refactorings their tool can detect.

Researchers can make refactoring detection techniques more comprehensive. For example, I observed that a common reason for Ratzinger's keyword-matching to misclassify changes as refactorings was that a bug-report title had been included in the commit message, and this title contained refactoring keywords. By excluding bug-report titles from the keyword search, accuracy could be increased. In general, future research can complement existing refactoring detection tools with refactoring logs from tools to increase recall of low-level refactorings.

3.5.3 Refactoring Practice

Several of my findings bolster existing evidence about refactoring practice across a large population of programmers. Unfortunately, the findings also suggest that refactoring tools need further improvements before programmers will use them frequently. First, my finding that programmers refactor frequently (Section 3.4.6) confirms the same finding by Weißgerber and Diehl [84] and Xing and Stroulia [85]. For toolsmiths, this highlights the potential of refactoring tools, telling them that increased tool support for refactoring may be beneficial to programmers.

Second, my finding that floss refactoring is a more frequently practiced refactoring tactic than root-canal refactoring (Section 3.4.5) confirms that floss refactoring, in addition to being recommended by experts [22], is also popular among programmers. This has implications for toolsmiths, researchers, and educators. For toolsmiths, this means that refactoring tools should support flossing by allowing the programmer to switch quickly between refactoring and other development activities, which is not always possible with existing refactoring tools, such as those that force the programmer's attention away from the task at hand with modal dialog boxes (Section 3.4.5). For researchers, studies should focus on floss refactoring for the greatest general-

ity. For educators, it means that when they teach refactoring to students, they should teach it throughout the course rather than as one unit during which students are taught to refactor their programs intensively.

Finally, my findings that many refactorings are performed without the help of tools (Section 3.4.7) and that the kinds of refactorings performed with tools differ from those performed manually (Section 3.4.8) confirm the results of my Agile Open Northwest 2007 survey on programmers' under-use of refactoring tools. Note that these findings are based on toolsmiths' refactoring tool usage, which I regard as the best case. Indeed, if even toolsmiths do not use their own refactoring tools very much, why would other programmers use them more? Toolsmiths need to explore alternative interfaces and identify common refactoring workflows, such as reminding users to EXTRACT LOCAL VARIABLE before an EXTRACT METHOD or finding a easy way to combine these refactorings: the goal should be to encourage and support programmers in taking full advantage of refactoring tools. For researchers, more study is needed about exactly *why* programmers do not use refactoring tools as much as they could.

3.5.4 Limitations of this Study

First, all the data report on refactoring behavior for the Java language in the Eclipse environment. While this is a widely-used language and environment, the results presented in this chapter may not hold for other languages and environments. Second, *Users* and *Toolsmiths* may not represent programmers in general. Third, the *Users* and *Everyone* data sets may overlap with the *Toolsmiths* data set: both the *Users* and *Everyone* data sets were gathered from volunteers, and some of those volunteers may have been *Toolsmiths*. However, the size of the subject pools limit the impact of any overlap: fewer than 10% of the members of *Users* and 0.1% of the members of *Everyone* could be members of *Toolsmiths*.

3.5.5 Study Details

Details of my methodology, including my publicly available data, the SQL queries used for correlating and summarizing that data, the tools I used for batching refactorings and grouping CVS revisions, my experimenters' notebook, and my normalization procedure, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

3.6 Conclusions

In this chapter, I have presented a study that re-examined several previously-held assumptions and conclusions about how programmers refactor. I confirmed some assumptions and conclusions, and disconfirmed others. In the short term, the results will lay a foundation for the guidelines and tools that I discuss in the remainder of this dissertation:

- In the next chapter, I argue that because refactoring tools are underused and because refactoring is frequent, productivity can be improved by encouraging programmers to use refactoring tools more frequently.
- In Section 4.7, I use the finding that floss refactoring is common to motivate the need for most refactoring tools to align with floss refactoring.
- In Section 6.6.3, I use the finding that programmers repeat refactorings to motivate the need for tools that allow programmers to repeatedly apply a refactoring to several pieces of code.
- In Section 8.4, I use the finding that programmers do not often configure refactoring tools to motivate the need for tools that allow programmers to skip configuration.

In the long term, I hope that these results change the way that researchers think about refactoring and the way that they conduct research on refactoring behavior.

Chapter 4

A Problem with Refactoring Tools: Usability [1]

From the data presented in the last chapter, it appears that refactoring is a common practice, yet refactoring tools are underused. This is a missed opportunity: every time that programmers refactor without a tool, they may be refactoring more slowly and in a more error-prone manner than if they had used a refactoring tool. This prompts the question: why do programmers not use refactoring tools when they could? In this chapter, I present data that suggest that usability is at least one underlying cause of this underuse.

4.1 Contributions

The major contributions of this chapter are as follows:

- An exploratory study of programmers during refactoring that uncovers two significant usability barriers to refactoring tools (Section 4.5).
- A survey of 112 people that suggests that slow user-interfaces in refactoring tools are at least one cause of refactoring tool underuse (Section 4.6).

[1] Parts of this chapter appeared in the *Proceedings of the 2008 International Conference on Software Engineering* [49], as part of a journal paper in *IEEE Software* [50], and as part of my thesis proposal.

4.2 Usability, Guidelines, and the Value of Guidelines Specific to Refactoring

Usability is a property of a tool's user interface. Nielsen describes five components of usability: learnability, efficiency, memorability, errors, and satisfaction [54, p. 26]. Learnability is the ease with which users can learn to use a tool. Efficiency is the level of productivity users can achieve after learning the tool. Memorability is how well users remember how to use the tool after some period of time. Errors are the mistakes that users make when using a tool. Satisfaction is how well the tool pleases the users.

In this dissertation, I focus on efficiency, errors, and satisfaction. In later chapters, I measure efficiency by examining how quickly programmers can use refactoring tools, measure errors by determining the mistakes that programmers make while using the tools, and measure satisfaction by asking programmers their opinions of the tools. With the exception of Section 7.5.2, I do not address memorability or learnability, because the short-term experiments in which I demonstrate improvements in efficiency, errors, and satisfaction are generally unsuitable for demonstrating long-term improvements in learnability or memorability.

The **first** usability contribution of this dissertation is a set of guidelines for refactoring tools. One way to demonstrate improved usability is by comparing one tool against another; if one or more of the usability components are better when using one of the tools (without sacrificing the other components), then you can say that usability is better in that tool. If one tool has good usability, a common way to capture what makes that tool's user interface good is by distilling the "goodness" in the form of guidelines [55, 65, 71]. In this way, I build guidelines specifically for refactoring tools.

Several researchers and practitioners have created general guidelines based on user-interface experiments and experience, including Nielsen [55], Raskin [65] and

Shneiderman [71]footnote A more complete set of general usability guidelines can be found in Mealy and colleagues' collection [45, Appendix A].. The guidelines that I will present in the remainder of this dissertation are a refined version of broad usability guidelines, where the refinements are derived from observations about refactoring tool usage. For instance, my guideline "Refactoring tool configuration should not force the programmer to view or enter unnecessary configuration information" (Section 8.2) is a more specialized version of Shneiderman's "Minimal input actions by user" [71, p. 72]. Throughout this dissertation, I will link my guidelines to such previously proposed guidelines where applicable.

General guidelines, such as those proposed by Shneiderman, Nielsen, and Raskin, are just that — general — so interface designers and toolsmiths may find it difficult to apply them to specific user interfaces. Thus, Shneiderman states, general guidelines "must be interpreted, refined, and extended for each environment" [71, p. 62]: in this dissertation I interpret, refine, and extend them for refactoring tools.

The **second** usability contribution of this dissertation is a collection of refactoring tools that exemplify my guidelines. This is important because guidelines alone can be too vague to be useful to a user-interface designer or developer. Thus, as Tetzlaff and Schwartz have stated, guidelines should be "developed primarily to *complement* toolkits and interactive examples" [77]. The primary usability contributions of this dissertation, then, are (1) guidelines and (2) tools that exemplify those guidelines; my hope is that the two together will help inspire improved usability in future refactoring tools.

4.3 Why Usability is Important to Refactoring Tools

The success of a refactoring tool depends on the quality of its interface more than does the success of other software tools. For example, even though a compiler may have terrible error messages, the programmer is largely dependent on the compiler

and will develop strategies to cope with the poor interface. This is not so with refactoring tools: if the tool does not help programmers to be more efficient, they can simply refactor by hand. And as mentioned in Section 2.4, if a programmer refactors by hand, she may be refactoring more slowly and introducing more errors than had she used a refactoring tool.

4.4 Related Work

Several pieces of previous work have suggested that poor usability is an important problem with refactoring tools.

In his dissertation on refactoring, Opdyke pointed out that the user interface is a significant part of refactoring tools: "Speed and clarity of the user interface are important in a refactoring tool that supports design exploration. A refactoring tool that is too slow will 'get in the way' and discourage trying out alternative designs. Ideally, a refactoring should execute instantaneously." [58]

After creating one of the first refactoring tools, Roberts [69] noted that the tool "was rarely used, even by ourselves." In response, the tool was improved by following three usability guidelines: speed in program transformation, support for undoing refactorings, and tight integration with the development environment. It appears that most refactoring tools since then have heeded these guidelines, yet new usability issues have sprung up, as I have discovered (Section 4.5). In my work I build on Roberts' guidelines to improve usability of refactoring tools.

More recently, Mealy and colleagues [45] distilled specific requirements for refactoring tools from general usability guidelines. Mealy and colleagues' work is top-down (that is, concerned with producing a complete set of refactoring tool guidelines based on general guidelines), while my research is bottom-up (that is, concerned with producing guidelines derived from specific bottlenecks in the refactoring process). As Mealy and colleagues develop and evaluate a proof-of-concept refactoring

tool, I expect that it will become more clear how to implement their requirements.

Even more recently, based on their experience building and testing refactoring tools at Microsoft and Developer Express, Campbell and Miller have found usability to be an issue. They write: "Unfortunately, many refactoring tools suffer from deep discoverability and usability problems that make them less useful for general development" [7].

Thus, it appears that usability was, and remains, a problem with refactoring tools.

4.5 An Exploratory Study of Refactoring

Beginning in late 2005, I undertook a formative study of programmers during refactoring to better understand the usability problems that exist in modern refactoring tools. In my personal experience, error messages emitted by existing tools are non-specific and unhelpful in diagnosing problems. The purpose of this study was to ascertain if other programmers also find these messages unhelpful.

4.5.1 The Extract Method Refactoring

One refactoring that has enjoyed widespread tool support is called EXTRACT METHOD. A tool that performs EXTRACT METHOD takes a sequence of statements, copies them into a new method, and then replaces the original statements with an invocation of the new method. This refactoring is useful when duplicated code should be factored out and when a long method contains several code segments that are conceptually separate.

I studied the EXTRACT METHOD tool in the Eclipse programming environment [18]. I reasoned that the EXTRACT METHOD tool in Eclipse is worthy of study because it is a mature, non-trivial refactoring tool and because most refactoring tool user-interfaces are very similar. This claim of similarity is based on my review of 16 refactoring tools in 2005 [48, p 3]

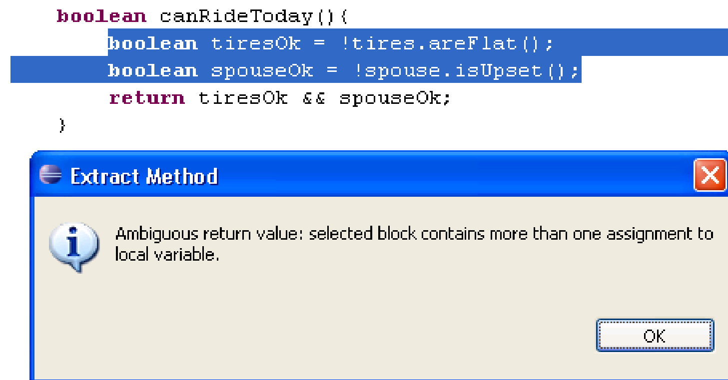


Figure 4.1: A code selection (above, highlighted in blue) that a tool cannot extract into a new method.

0	The selected code must be a list of statements.
1	Within the selection, there must be no assignments to variables that might be used later in the flow of execution. For Java, this can be relaxed to allow assignment to one variable, the value of which can be returned from the new method.
2	Within the selection, there must be no conditional returns. In other words, the code in the selection must either always return, or always flow beginning to end.
3	Within the selection, there must be no branches to code outside of the selection. For Java, this means no break or continue statements, unless the selection also contains their corresponding targets.

Table 4.1: Preconditions to the EXTRACT METHOD refactoring, based on Opdyke's preconditions [58]. I have omitted preconditions that were not encountered during the formative study.

To use the Eclipse EXTRACT METHOD tool, the programmer first selects code to be refactored, then chooses a refactoring to perform, then configures the refactoring via a "refactoring wizard," and then presses "OK" to execute the refactoring. If there is a precondition violation, the browser then presents the user with a generic textual error message. Figure 4.1 on the previous page displays an example of such an error message in Eclipse. Table 4.1 lists several preconditions for the EXTRACT METHOD refactoring.

4.5.2 Methodology

I observed eleven programmers perform a number of EXTRACT METHOD refactorings. Six of the programmers were Ph.D. students and two were professors from Portland State University; three were commercial software developers.

I asked the participants to use the Eclipse EXTRACT METHOD tool to refactor parts of several large, open-source projects:

- Azureus, a peer-to-peer file-sharing client (http://azureus.sourceforge.net);
- GanttProject, a project scheduling application (http://ganttproject.biz);
- JasperReports, a report generation library (http://jasperforge.org);
- Jython, a Java implementation of the Python programming language (http://www.jython.org); and
- the Java 1.4.2 libraries (http://java.sun.com/j2se/1.4.2/download.html).

I picked these projects because of their size and maturity.

Programmers were free to refactor whatever code they thought necessary. To give some direction, the programmers were allowed to use a tool to help find long methods, which can be good candidates for refactoring. However, the programmers

chose on which projects to run the long-method tool, and which candidates to refactor. I trained the subjects by showing them how to use the long-method finding tool and the standard Eclipse EXTRACT METHOD tool, by demonstrating how the tools can find a large method and extract a new method from it. Each refactoring session was limited to 30 minutes; programmers successfully extracted between 2 and 16 methods during that time.

4.5.3 Results

The study led to some interesting observations about how often programmers can perform EXTRACT METHOD successfully:

- In all, 9 out of 11 programmers experienced at least one error message while trying to extract code. The two exceptions performed some of the fewest extractions in the group, so were among the least likely to encounter errors. Furthermore, these two exceptions were among the most experienced programmers in the group, and seemed to avoid code that might possibly generate error messages.
- Some programmers experienced many more error messages than others. One programmer attempted to extract 34 methods and encountered errors during 23 of these attempts, while 2 programmers experienced no errors at all.
- Error messages regarding syntactic selection occurred about as frequently as any other type of error message (violating Precondition 0 in Table 4.1 on page 51). In other words, programmers frequently had problems selecting a desired piece of code. This was usually due to unusual formatting in the source code or to the programmer trying to select statements that required the editor to scroll.

- The remaining error messages concerned multiple assignments and control flow (violations of Preconditions 1 through 3 in Table 4.1 on page 51).

- The tool reported only one precondition violation, even if multiple violations existed. These observations suggest that, while trying to perform EXTRACT METHOD, programmers fairly frequently encounter a variety of errors arising from violated refactoring preconditions.

Based on my observations of programmers struggling with refactoring error messages, I make the following conjectures:

- Error messages were insufficiently descriptive. Programmers, especially refactoring tool novices, may not understand an error message that they have not seen before. When I asked what an error message was saying, several programmers were unable to explain the problem correctly.

- Programmers conflated error messages. All the errors were presented as graphically-identical text boxes with identically formatted text. At times, programmers interpreted one error message as an unrelated error message because the errors appeared identical at a quick glance. The clarity of the message text is irrelevant when the programmer does not take the time to read it.

- Error messages discouraged programmers from refactoring at all. For instance, if the tool said that a method could not be extracted because there were multiple assignments to local variables (Figure 4.1 on page 51), the next time a particular programmer came across any assignments to local variables, the programmer did not try to refactor, even if no precondition was violated.

This study reveals room for two types of improvements to EXTRACT METHOD tools. First, to prevent a large number of mis-selection errors, programmers need support in making a valid selection. Second, to help programmers recover successfully

from violated preconditions, programmers need expressive, distinguishable, and understandable feedback that conveys the meaning of precondition violations. More generally, this study suggests that usability is a problem with refactoring tools.

4.6 A Survey about Refactoring Behavior

I conducted a survey at Agile Open Northwest 2007 [48], a regional conference for enthusiasts of Agile programming. I asked 112 people why they chose not to use refactoring tools when they were available using a multiple-choice question. Among 71 people who used programing environments that have refactoring tools, a popular response was that "I can refactor faster by hand than with a tool" ($n = 24$). At the same time, only 2 people marked that "My code base is so large that the refactoring tool takes too long," suggesting that the back-end (code transformation) component is sufficiently fast in most cases. Together, these two responses suggest that, if the user interface to refactoring tools were faster, programmers would be more willing to use them.

4.7 Usable Floss Refactoring Tools

In this section, I propose principles for better refactoring tools based on previously presented data on how programmers refactor (an argument whose structure I will use repeatedly in the remaining chapters). Using the definition of floss refactoring (Section 2.3) and the observation that floss refactoring is more common (Section 3.4.5), I then argue that current refactoring tools do not support floss refactoring as well as they could, and that more usable refactoring tools can be built by supporting this common tactic.

4.7.1 Principles for Tools that Support Floss Refactoring

If a tool is suitable for floss refactoring, then the tool must support frequent bursts of refactoring interleaved with other programming activities. I propose five principles to characterize such support:

Principle 1 Allow the programmer to choose the desired refactoring quickly.

Principle 2 Allow the programmer to switch seamlessly between program editing and refactoring.

Principle 3 Allow the programmer to view and navigate the program code while using the tool.

Principle 4 Allow the programmer to avoid providing explicit configuration information.

Principle 5 Allow the programmer to access all the other tools normally available in the development environment while using the refactoring tool.

Unfortunately, refactoring tools do not always align with these principles; as a result, floss refactoring with tools can be cumbersome.

Recall performing the EXTRACT METHOD refactoring using Eclipse, as described in Section 2.4. After selecting the code to be refactored, you needed to choose which refactoring to perform, which you did using a menu (Figure 2.3 on page 10). Menus containing refactorings can be quite long and difficult to navigate; this problem gets worse as more refactorings are added to development environments. As one respondent complained in the free-form part of my Agile 2007 survey, the "[refactoring] menu is too big sometimes, so searching [for] the refactoring takes too long." Choosing the *name* that most closely matches the transformation that you have in your head is also a distraction: the mapping from the transformation to the name is

not always obvious. Thus, using a menu as the mechanism to initiate a refactoring tool violates **Principle 1**.

Next, most refactoring tools require configuration (Figure 2.4 on page 11). This makes the transition between editing and refactoring particularly awkward, as you must change your focus from the code to the refactoring tool. Moreover, it is difficult to choose contextually-appropriate configuration information without viewing the context, and a modal configuration dialog like that shown in Figure 2.4 on page 11 obscures your view of the context. Furthermore, you cannot proceed unless you provide the name of the new method, even if you do not care what the name is. Thus, such configuration dialogs violates **Principle 2**, **Principle 3**, and **Principle 4**.

Before deciding whether to apply the refactoring, you are given the opportunity to preview the changes in a *difference viewer* (Figure 2.5 on page 12). While it is useful to compare your code before and after refactoring, presenting the code in this way forces you to stay inside the refactoring tool, where no other tools are available. For instance, in the difference viewer you cannot hover over a method reference to see its documentation — something that can be done in the normal editing view. Thus, a separate, modal refactoring preview violates **Principle 5**.

Although this discussion used the Eclipse EXTRACT METHOD tool as an example, I have found similar problems with other tools. These problems make the tools less useful for floss refactoring than might otherwise be the case.

4.7.2 Tools for Floss Refactoring

Fortunately, some tools support floss refactoring well, and align with my principles. Here are several examples.

In Eclipse, while you initiate most refactorings with a cumbersome hierarchy of menus, you can perform a MOVE CLASS refactoring simply by dragging a class icon in the Package Explorer from one package icon to another. All references to

```
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == ((Long)obj).longValue();
    }
    return false;
}
```

```
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == m(obj);
    }
    return false;
}

private long m(Object obj){
    return ((Long)obj).longValue();
}
```

Figure 4.2: At the top, a method in `java.lang.Long` in an X-develop editor. At the bottom, the code immediately after the completion of the EXTRACT METHOD refactoring. The name of the new method is `m`, but the cursor is positioned to facilitate an immediate RENAME refactoring.

the moved class will be updated to reflect its new location. This simple mechanism allows the refactoring tool to stay out of your way; because the class and target package are implicitly chosen by the drag gesture, you have already provided all the configuration information required to execute the refactoring. Because of the simplicity and speed of this refactoring initiation mechanism, it adheres to **Principle 1**, **Principle 2**, and **Principle 4**.

The X-develop environment [75] makes a significant effort to avoid modal dialog boxes for configuring its refactoring tools. For instance, the EXTRACT METHOD refactoring is performed without any configuration at all, as shown in Figure 4.2. Instead, the new method is given an automatically generated name. After the refactoring is complete, you can change the name by placing the cursor over the generated name, and typing a new name: this is actually a RENAME refactoring, and the tool

makes sure that all references are updated appropriately. X-develop refactoring tools adhere to **Principle 2** and **Principle 4**.

Rather than displaying a refactoring preview in a separate difference view, Refactor! Pro [15] marks the code that a refactoring will modify with *preview hints*. Preview hints are editor annotations that let you investigate the effect of a refactoring before you commit to it. Because you do not have to leave the editor to see the effect of a refactoring, preview hints adhere to **Principle 3** and **Principle 5**.

Thus, characteristics make tools are less suited to how programmers usually refactor, while others are more suited. What is the difference between the two groups of tools? In the remaining chapters, I discuss this difference by proposing new user interfaces for refactoring tools that align with how programmers usually refactor. I also present guidelines that have informed the design of these user interfaces, guidelines that I hope will promote the development of more usable refactoring tools in the future.

Chapter 5

The Identification Step: Finding Opportunities for Refactoring [1]

Before a programmer can refactor code, she must recognize the need for refactoring (Figure 2.6 on page 14). *Smells* are patterns in code that can help programmers recognize that code should be refactored [22]. For example, consider the following code snippet:

```
class TrainStation{
  int lengthOf(Train t) {
    return  t.locomotiveCount() +
            t.boxcarCount() +
            1; //the caboose
  }
...
```

The method `lengthOf` exhibits the FEATURE ENVY smell, because the method sends several messages to a `Train` object, but it sends no messages to the `TrainStation` object. FEATURE ENVY is a problem that can make software more difficult to change because a class's responsibilities are not contained in the class itself, but are spread throughout "envious" classes that access the class's members. Table 5.1 on the next page describes several other code smells. In general, code smells indicate that refactoring may be appropriate, but are highly subjective and context-dependent.

[1] A preliminary version of this chapter appeared in the *Proceedings of the International Workshop on Recommendation Systems for Software Engineering* [51].

DATA CLUMPS	A group of data objects that is duplicated across code [22]
FEATURE ENVY	Code that uses many features from classes other than its own [22]
REFUSED BEQUEST	A method that overrides a superclass method, but does not use the super method's functionality [22]
SWITCH STATEMENT	A switch statement, typically duplicated across code [22]
MESSAGE CHAIN	A series of method calls to "drill down" to a desired object [22]
TYPECAST	Changing an object from one type to another type [13]
INSTANCEOF	An operator that introspects on the type of an object [13]
MAGIC NUMBER	A hard-coded value that is poorly documented [22]
LONG METHOD	A method with too much code [22]
LARGE CLASS	A class with too much code [22]
COMMENTS	Comments indicate that code is not self-explanatory [22]

Table 5.1: Some smell names and descriptions

FEATURE ENVY can be alleviated by delegating the functionality to the `Train` class by sequencing three smaller refactorings: EXTRACT METHOD, then MOVE METHOD, and then RENAME METHOD, to produce the following code:

```
class TrainStation{
  int lengthOf(Train t) {
    return  t.length();
  }
...
class Train{
  int length() {
    return  locomotiveCount() +
            boxcarCount() +
            1; //the caboose
  }
...
```

The mechanics of these refactorings are not the topic of this chapter; instead I focus on how the programmer *recognizes* that code needs to be refactored.

Until recently, programmers have been forced to identify smells manually, which can be difficult for two reasons. First, novice programmers sometimes cannot locate smells as proficiently as more experienced programmers, as Mäntylä has demonstrated in an experiment [43]. Second, it is burdensome, even for expert programmers, to inspect every piece of code for every possible smell (22 are cataloged in Fowler's book alone [22]).

Fortunately, many smells can be detected automatically by tools called *smell detectors*. As future work, Opdyke's dissertation stated the need for user-interface research into smell detectors: "User interface approaches could be studied for assisting a user in making refactoring related design decisions" [58, p. 183]. Once a smell detector has found a smell, how can it communicate its findings to the programmer as efficiently as possible?

5.1 Contributions

The major contributions of this chapter are as follows:

- User-interface guidelines for making more usable tools that help programmers

to find candidates for refactoring, guidelines derived from the strengths of existing tools (Section 5.2).

- A new tool, called Stench Blossom, to find candidates for refactoring that is designed to fit into the floss refactoring workflow (Section 5.3).
- The first experiment to put such a tool in the hands of programmers. The experiment provides evidence that Stench Blossom helps programmers find more candidates for refactoring; that programmers generally believe that my postulated guidelines state desirable properties of smell detectors; and that my tool generally obeys those guidelines (Section 5.4).

5.2 Guidelines and Related Work

To better understand what makes an effective smell detection tool, and to postulate user-interface guidelines for such a tool, I examine two existing approaches for displaying code smells. The first, visualizations, typically provides a view separate from the source code, in which smells are represented graphically (for example, Figure 5.1 on the following page). The second, editor annotations, layers graphics or text on top of the programmer's editor to convey smell information (for example, Figure 5.2 on the next page). I will discuss each in turn; it is important to note that the *strengths* of visualizations are often the *weaknesses* of editor annotations, and vice versa. However, when I speak about the weaknesses of visualizations and editor annotations, I am speaking about how they *typically* are designed. Indeed, limitations of both kinds of user interfaces could be overcome with some creative retooling.

5.2.1 Visualizations

Several smell detector visualizations have been proposed in prior work. The *Crocodile* tool displays code smells using a 3-dimensional visualization where the

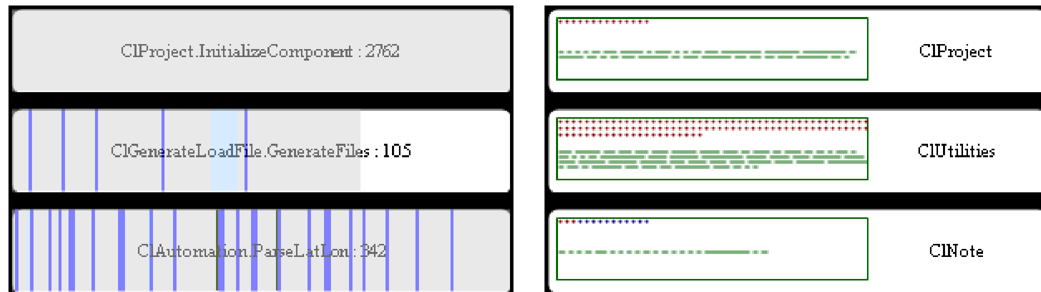


Figure 5.1: Examples of a smell visualization in Noseprints [62]. On the left, information about Long Method for 3 classes, and on the right, information about Large Class for 3 other classes. This visualization appears inside of a window when the programmer asks the Visual Studio programming environment to find smells in a code base. Screenshots provided courtesy of Chris Parnin.

```
(Class<Long>) Class.getPrimitiveClass("long");
```

Figure 5.2: A compilation warning in Eclipse, shown as a squiggly line underneath program code. This line, for example, calls attention to the fact that this expression is being Type-cast.

distance between objects represents some smell [73]. The *jCosmo* tool analyzes the entire program and displays a graph; the size and color of the graph nodes show which parts of the system are affected by which code smells [13]. More recently, the *Noseprints* tool (Figure 5.1) displays smells using a 2-dimensional, full-screen visualization [62]. These three visualizations have at least six desirable properties when displaying smells.

First, visualizations are **Scalable**. Code smells can emanate from many pieces of code, and one piece of code can emit several smells. Consider again the `TrainStation` example. While it is a relatively small method, it contains at least three code smells: Feature Envy, Magic Number, and Comments. Depending on the rest of the program, other smells such as Refused Bequest may be present in the snippet as well. Indeed, nearly *all* the code in this method smells! Underlining everything that contains even a whiff of a code smell could quickly

overwhelm the programmer, making the detector useless. Thus:

- ◇ **Scalability.** A smell detector should not display smell information in such a way that a proliferation of code smells overloads the programmer.

Scalability is an important quality of many user interfaces, usually expressed as the avoidance of "information overload" [41].

Visualizations are scalable because zooming in or out of a visualization allows the programmer to grasp smell data more easily. Editor annotations are not scalable, because almost every piece of code in a programmer's editor could be annotated and because several annotations may overlap on the same source code.

Second, visualizations can be **relational**. Sometimes a smell does not emanate from a single point in the code, but instead speaks of relationships *between* several program elements. Like compilation errors, code smells can relate to several program elements that may be distributed across the program text. For instance, REFUSED BEQUEST is not simply a problem with an overriding method: it is a problem with that method, with its overridden superclass method, and potentially with sibling methods that override that superclass method but do not call `super`. Thus:

- ◇ **Relationality.** A smell detector should display smell information relationally when related code fragments give rise to smells.

Visualizations can easily show related program elements by linking them together with connectors or colors. Editor annotations are generally not relational; they point at one particular contiguous piece of code as problematic. Moreover, editor annotations cannot easily show relations between non-local program elements.

Third, visualizations are **Biased**. Not every kind of smell that a tool can detect has equal value to the programmer. This is because some smells are obvious to the naked eye (e.g., LONG METHOD), while others are difficult for a programmer to find (e.g., FEATURE ENVY) [43]. Thus:

◇ **Bias.** A smell detector should place emphasis on smells that are more difficult to recognize without a tool.

Visualizations can be biased because one smell can easily be emphasized over another, such as by varying glyph size or hue. Editor annotations are typically not biased because code is either annotated (contains a smell) or is not annotated (does not contain a smell), with no room in between.

Fourth, visualizations are **task-centric**. Because floss refactoring does not encourage refactoring for its own sake, it is important that a smell detector does not encourage a programmer to refactor excessively. Thus:

◇ **Task-centricity.** A smell detector should not distract from the programmer's primary task, if the need for refactoring is weak.

Task-centricity is an important property of information display in general; as Raskin puts it, "Systems should be designed to allow users to concentrate on their jobs" [65].

Visualization tools are usually implemented in a task-centric manner because they are only shown when the programmer asks for them. Editor annotations can distract from a programmer's task because they are always overlaid on program text, and may encourage the programmer to refactor at the slightest whiff of a code smell.

Fifth, visualizations are **estimable**. Smells such as DUPLICATION may be spread throughout a whole class whereas others may be localized in only one place. The extent of such spread can help the programmer determine whether or not a smell should be refactored away, and how much effort and reward such a refactoring would entail. Thus:

◇ **Estimability.** A smell detector should help the programmer estimate the extent of the smell spread throughout the code.

Estimability is similar to Shneiderman's recommendation for constructive guidance, although that recommendation was in the domain of recovering from errors [70, p. 58].

Visualizations can help programmers estimate the extent of smells in their code by, for example, increasing the size or clustering of visual objects with their increasing spread. Editor annotations sometimes do not help the programmer estimate the extent of a smell because annotations are typically binary, either being shown or not shown depending on whether the tool judges that the smell exists or not.

5.2.2 Editor Annotations

Editor annotations have been proposed by several researchers for use in smell detectors. Built on top of the Eclipse programming environment, CodeNose underlines locations in the program text where smells have been detected [74], much like Eclipse's standard compilation warnings (Figure 5.2 on page 64). A similar line-based indicator for smell detectors has been independently proposed by Hayashi and colleagues [26], Bisanz [3], and Tsantalis and colleagues [82]. These editor annotations have at least four desirable properties when displaying smells.

First, editor annotations have high **availability**. If a programmer were to use a smell detector during floss refactoring, she would need to run it frequently, interleaved with program modifications. During floss refactoring, the programmer should not have to frequently go through a series of steps to see if a tool finds any code smells. Thus, I postulate that availability is an important guideline of smell detectors:

- ◇ **Availability.** A smell detector should make smell information as available as soon as possible, with little effort on the part of the programmer.

Highly-available code smell detection has been expressed as a tool requirement by

Mealy and colleagues: "Provide incremental exception (code smell) checking with inline, non-disruptive feedback" [45].

Editor annotations can be always available because program analysis can run in the background and show the results on top of the source code. Visualizations are typically not available unless the programmer specifically requests them by running a special tool and viewing the results in a separate window. However, either of these user-interfaces *could* be built with high or low availability.

Second, editor annotations are **unobtrusive**. Due to interleaving of coding and refactoring during floss refactoring (Section 3.4.5), a smell detector should be unobtrusive:

◇ **Unobtrusiveness.** A smell detector should should not stop the programmer from programming while gathering, analyzing, and displaying information about smells.

Unobtrusiveness is similar to Shneiderman's recommendation regarding internal locus of control: "experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions" [71, p. 62].

Editor annotations are unobtrusive because they can be seen while working on a primary coding task, and can be used even without directly looking at them; having editor annotations on the programmer's visual periphery may be sufficient to make her aware that her code is smelly. Visualizations are typically obtrusive, again, because they often require the programmer to wait to see the visualization, while the tool analyzes the code.

Third, editor annotations are **context-sensitive**. Because the programmer performs floss refactorings only if they help to accomplish an immediate programming goal, the programmer is most interested in smells related to the code on which she is currently working. Fixing smells in a context-*insensitive* manner may be an ineffi-

cient way of using resources, or may even be counter-productive. Thus:

◇ **Context-Sensitivity.** A smell detector should first and foremost point out smells relevant to the current programming context.

Mankoff and colleagues have stated that **context-sensitivity** is important in information displays: "the information should be useful and relevant to the users in the intended setting" [42].

Editor annotations are context-sensitive because they decorate the code that the programmer is working on. Visualizations are typically not context sensitive, because they either visualize the whole program [13] or programmer-specified parts of the program [73, 62].

Finally, both editor annotations and visualizations can be **expressive**. Smells can be complex and difficult to understand. Why is this? Smells are potentially more difficult to understand than compilation errors because a piece of code either generates a compilation error or it does not, but an instance of a code smell may be subtle or flagrant, widespread or centralized, or anywhere in between. A smell detector that communicates these properties may be helpful to the programmer when she judges whether or not to refactor. Thus:

◇ **Expressiveness.** A smell detector should go further than simply telling the programmer that a smell exists; it should help the programmer find the source(s) of the problem by explaining *why* the smell exists.

Like **Estimability**, **Expressiveness** is a way to achieve Shneiderman's recommendation for constructive guidance so that the programmer can make intelligent choices about the next step [70, p. 58].

Editor annotations can be expressive when they provide more information at a programmer's request; the typical user interface to accomplish this is a tooltip, where

a programmer hovers over a code annotation to reveal a textual description of the smell. Visualizations can be expressive as well, although the mechanism will vary from tool to tool.

In this section I have demonstrated that visualizations and editor annotations have complementary strengths. This raises the question of whether it is possible to combine all of these strengths in one user interface?

5.3 Tool Description

To demonstrate how all of the guidelines in Section 5.2 can be implemented, I have built a prototype smell detector called Stench Blossom that provides three views of code smells. By default, *Ambient View* shows a smell visualization while the developer is working with code. Then, if the programmer notices something unusual, she mouses over the visualization to identify specific smells in *Active View*. If she desires details, she requests more information by mouse clicking to reveal details in *Explanation View*. While I describe the tool textually in this section, it is more useful to see it in action; a series of short screencasts can be found at http://multiview.cs.pdx.edu/refactoring/smells. In the description that follows, I emphasize how Stench Blossom satisfies a guideline by setting the name of the guideline in **bold**.

5.3.1 Ambient View

The initial view of the smell detector is an ambient information display [61], where a visual representation of contextually relevant smells is displayed in the editor, but behind the program text (Figure 5.3 on the following page). This visualization is intended to be visible and **available** at all times during code editing and navigating. It is also intended to be light enough to avoid being distracting, and thus it is intended to be **task-centric**. Analysis also happens in the background without the programmer

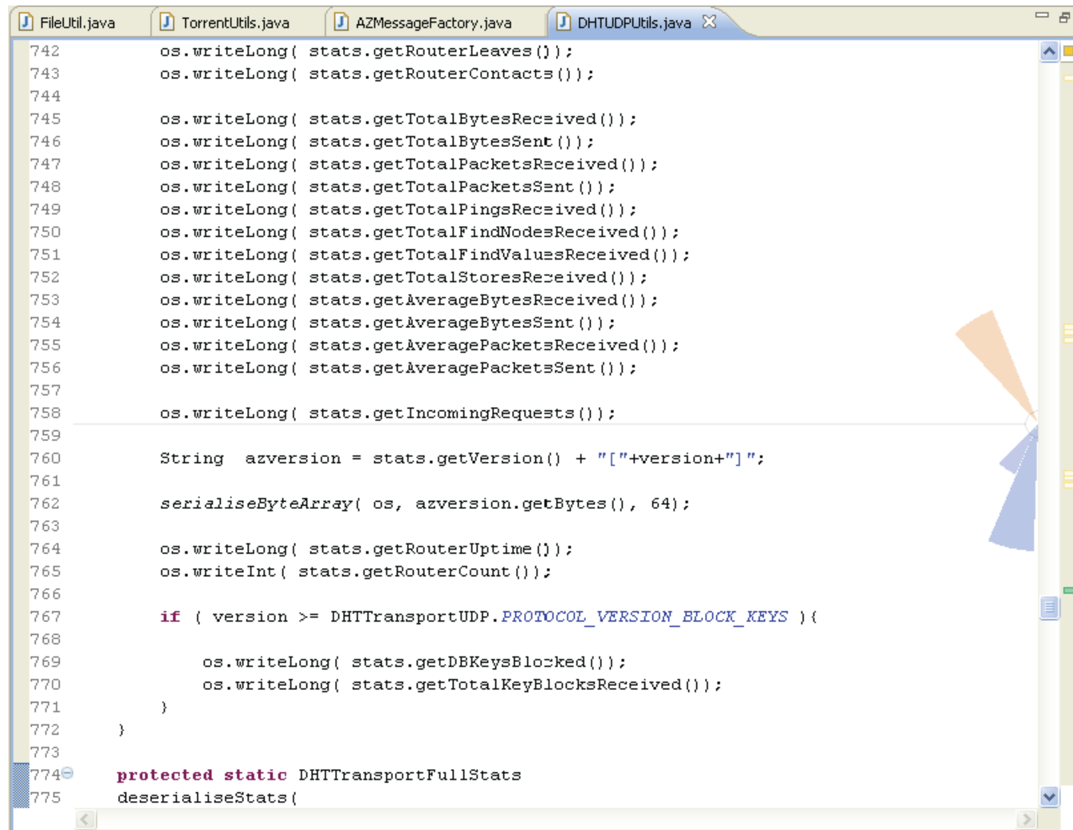


Figure 5.3: *Ambient View*, displaying the severity of several smells at the right of the editor.

having to request it, somewhat like Eclipse's incremental compilation [18], and thus is designed to be **unobtrusive**.

The visualization is composed of sectors of a circle, which I call petals, radiating from a central point in a half-circle. Each petal represents a code smell, and the radius of each petal represents the severity of the smell in the current programming context. For example, in Figure 5.3, the southernmost petal indicates the strongest smell while the northernmost petal indicates the weakest smell. As the programmer navigates through the code, the program text flows in front of the visualization, and the radius of each petal changes as the programmer's context changes. The radius of each petal is controlled by a smell analyzer that evaluates a smell in the programmer's

context. However, the maximum screen area available for each petal is bounded, and thus the visualization is designed to **scale** as the number of smells increases.

Petals are colored from red-to-green, north-to-south. Smells are assigned to petals such that the southernmost (and greenest) petal is the most obvious smell and the northernmost (and reddest) petal is the least obvious smell. My subjective obviousness ordering is reflected in Table 5.1 on page 61, where the least obvious smell appears at the top, as it does in Ambient View[2]. For example, in Figure 5.3 on the preceding page, the view indicates that there is a strong unobvious smell (in this case, FEATURE ENVY, although the smell names are intentionally omitted from this view) as well as a strong obvious smell (LARGE CLASS). This feature is intended to allow the programmer to judge **bias** at a glance because more obvious smells are visually distinguishable from less obvious smells, both spatially (top-to-bottom) and chromatically (red-to-green). If the programmer notices that the visualization is generally top-heavy (and mostly red), then there is a strong smell that the programmer may not otherwise notice; in contrast, if the visualization is generally bottom-heavy (and mostly green), the programmer can infer that while there is a strong smell, she is more likely to be already aware of it.

The purpose of this view is to allow the programmer to occasionally attend to the visualization to see if there are any strong, relevant code smells and to get a rough estimate of their degree. Little work or commitment is required on the part of a programmer to find out whether a smell exists; she needs only to glance at the visualization. This is unlike most existing smell detection tools, which require the programmer to activate the tool and to inspect the results. Indeed, such existing smell detectors are designed to be usable only while inspecting code, but not while fixing bugs or adding features, making them usable only in one out of three of Fowler's refactoring tasks [22, pp.58-59].

[2]I have implemented analyzers for all the smells in Table 5.1 on page 61, except for REFUSED BEQUEST, MAGIC NUMBER, and COMMENTS.

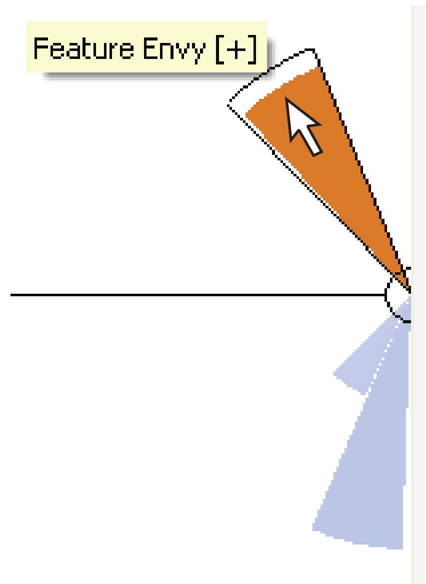


Figure 5.4: *Active View*, where the programmer has placed the mouse cursor over a petal representing FEATURE ENVY to reveal the name of the smell and a clickable [+] to allow the programmer to transition to *Explanation View*.

5.3.2 Active View

If the programmer observes something interesting or unusual in the *Ambient View*, she can then mouse-over a petal to reveal the name of that petal's associated smell. Furthermore, the petal's color is darkened to bring it into the programmer's focus. In Figure 5.4, the programmer has moused-over the second smell from the top. If the programmer is interested in further details of the detected smell, she can click on the smell label to activate *Explanation View*.

The purpose of *Active View* is to provide a little more information than *Ambient View*, and to help the programmer transition to *Explanation View*. Again, the transition from view to view in order to reveal more information was designed to be as fast and as painless as possible.

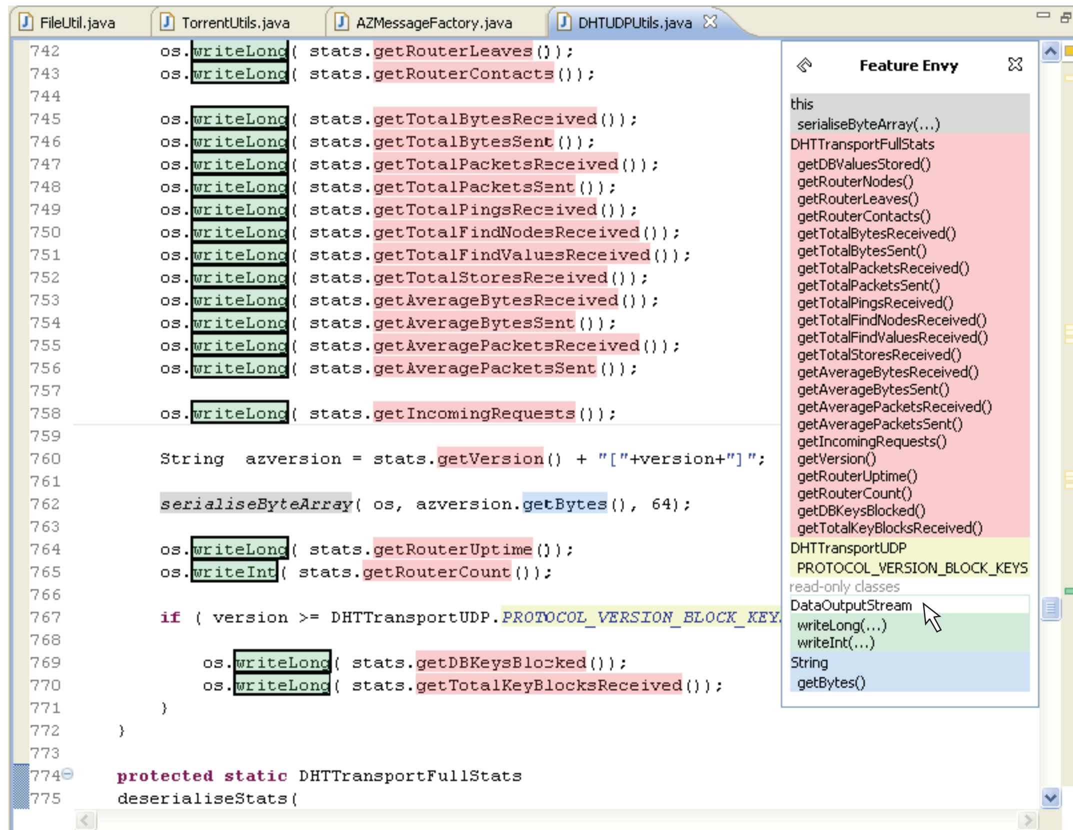


Figure 5.5: *Explanation View*, showing details about the smell named in Figure 5.4 on the previous page.

5.3.3 Explanation View

Explanation View provides detailed information about a particular smell. In essence, this view was designed to explain *why* the smell's petal has the displayed radius in the most **expressive** way possible. Each smell is displayed using different visual elements, but the display of *Explanation View* typically has two common components.

Summary Pane. In Figure 5.5, a summary pane is displayed at the upper right of the editor. This pane is fixed relative to the program code (that is, it does not move when the programmer navigates away), but may be moved by the programmer using the mouse. Generally, this pane displays a summary of the data collected from the smell analyzer. In Figure 5.5, the summary pane for the FEATURE ENVY smell

shows that 22 members from the `DHTTransportFullStats` class are accessed, while only one member of the current class is accessed; this is the reason that Stench Blossom rates FEATURE ENVY as so severe. The `DataOutputStream` class label is moused over in the figure; this caused all references to members in that class to be outlined in black.

Editor Annotations. The code in the editor is typically annotated to point out the origin of smells. For example, in the Figure you can see where the members of the `DataOutputStream` class are referenced in the code. All references to a particular external class are **related** visually by using the same color highlight. The extent of the problem is also **estimable**; for instance, in Figure 5.5 on the preceding page, you can see that many messages are sent to `DHTTransportFullStats` throughout this code. Looking at the code in this figure from a programmer's perspective, I might judge that because the code "envies" the `DHTTransportFullStats` class, this code would be more cohesive if I refactored it and placed it in a new method in the `DHTTransportFullStats` class.

In summary, *Explanation View* was built to help the programmer not only understand *if* code smells, but *why* the code smells.

5.3.4 Details of Stench Blossom

While I have outlined how Stench Blossom works in general, a number of details have significant bearing on the tool's practicality.

First, how does Stench Blossom determine the radius of each petal in *Ambient View*? The maximum size of each petal is fixed so that it does not monopolize the editor. Each individual smell analyzer is responsible for calculating a scalar metric for one smell in the programmer's working context, a calculation that produces a value between zero and the maximum radius of the petal. The formula for this metric is different for different analyzers; some formulae are more complex than others. For

instance, LARGE CLASS uses a relatively simple metric because the radius increases as the size of the class increases, while the metric for FEATURE ENVY incorporates the number of external classes referenced, the number of external members referenced, and whether internal members are referenced.

Second, how does the Stench Blossom search for smells efficiently? Several smell analyzers require complex program analysis, so as the number and complexity of analyzers increase, the development environment may begin to respond more slowly. Having detection run in a background thread and caching smell results for unchanged program elements are important techniques for maintaining acceptable performance. Moreover, I hope that a more intelligent search strategy, starting in the programmer's current context and radiating outward to less contextually relevant code, will improve performance even further. Smell analysis may also be made more efficient by using heuristics to analyze smells for *Ambient View*, but using full static analysis in *Explanation View*. In this way, the development environment can remain responsive during normal development and be fully accurate during smell inspection.

Third, what constitutes the programmer's "current context?" In my implementation, I define current context as the union of all methods, whole or partial, that are visible in the active editor. In the future, I may use more sophisticated definitions of context, such as the task contexts used by the Mylyn tool [36] or Parnin and Görg's usage contexts [60].

5.4 Evaluation

I evaluated Stench Blossom, and thus, indirectly, my guidelines, by conducting an experiment where programmers used the tool to find and analyze smells. Experiment materials, including the experimenter's notebook and the results database, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

	Professionals	**Students**	**Both**
Count	6	6	12
Years of programming experience (median)	12.5	5.5	9.5
Hours spent per week programing (median)	30	17.5	30
At least some refactoring knowledge (subject count)	6	4	10
At least some smell knowledge (subject count)	4	0	4
Eclipse users (subject count)	4	5	9
Refactoring tool users (subject count)	4	1	5

Table 5.2: Programming experience of subjects.

5.4.1 Subjects

I recruited a total of 12 subjects: 6 commercial Java developers and 6 Portland State University students from a graduate class on relational database management systems. Subjects were recruited using an email that stated that participants needed to be at least moderately familiar with Java, and unfamiliar with the smell detector described here. During the experiment, all subjects confirmed that they were familiar with Java, and additionally that they were familiar with refactoring. The majority reported that they had at least heard of the concept of code smells.

Subjects from the class were asked to volunteer to participate in exchange for extra credit on one programming assignment. Professional subjects were drawn from a pool of local professional programmers who had volunteered previously at Java and Eclipse user group meetings. Professional subjects were not compensated, although every subject was offered a beverage during the experiment.

Table 5.2 lists the experience of the professional and student programmers, as

self-reported in a pre-experiment questionnaire. Additionally, all subjects use Integrated Development Environments, and are at least moderately familiar with Java. These data suggest that subjects arrived with the requisite amount of programming experience, and a varying level of experience with refactoring and smells.

5.4.2 Methodology

I conducted the experiment using a laptop (1.7 GHz, 2GB of RAM, 15.4 inch wide screen display running 1280×800 resolution) with an external mouse. I conducted each experiment one-on-one, with one subject and myself as experiment administrator.

Subjects were divided into four groups to mitigate learning effects. Half of the subjects did smell detection tasks without the aid of Stench Blossom first, then with the aid of Stench Blossom, while the other half did the smell detection with Stench Blossom first, then without it. Within these two groups, half of the subjects worked over codeset A first, then B second, and half over codeset B first, then A second. I chose codesets A and B to contain an approximately equal variety of smells.

The experiment had four parts. First, the subject filled out a 1-page questionnaire regarding their programming, refactoring, and smell-detection experience. Subjects were then given eight 3×5 cards, each containing a smell name and description on the front, and an example on the back. Subjects were asked to read these cards within a few minutes, and were told that they would later be asked to find smells as well as explore some smell details.

Second, subjects were asked to skim four java files, top to bottom, and mention any smells that they noticed. For two of the files, subjects looked for the smells manually, and for the other two they used the smell detector. Before using the detector, I gave each subject a demonstration as I read aloud the following description:

> The tool is represented by a visualization behind your Java code. It looks

> a bit like a bunch of petals on a flower. Each petal represents a smell, and you can hover over to see the name of the smell. The size of the petal represents how bad that smell is in the code that you are looking at. As this tripwire passes over methods, or when the cursor is in a method, the smells for that method are visualized. This part of the tool is intended to give you an idea of which smells are present. There's more detail to the tool, but I'll get to that later.

The programmer then began the task, and I recorded which of the 8 smells that the programmer noticed, with and without Stench Blossom.

Third, subjects analyzed FEATURE ENVY in four different methods: two methods with Stench Blossom, and two methods without. I gave the subjects a demonstration as I read aloud the following description:

> Suppose that I glance at the smell indicator and see that Feature Envy is high. I can then click on its label, and get a detailed view of what's going on. The movable sheet shows me which classes members are referenced, and assigns each class a color. So, for instance, I can see that many members of DHTTransportFullStats are referenced, but only one member in this class is referenced. The associated members are highlighted in source code, and I can mouse-over the classes and members to emphasize their occurrences in code. Looking at this detail, I might conclude that the method, or some parts of it, should be moved to DHTTransportFullStats.

I then read subjects the following task description:

> So the task that I want you to do is to use the tool to help you make some judgments about the code; how widespread the Feature Envy is, how likely you are to remove it, and how you might do it.

During the experiment, I recorded these judgements.

In the fourth and final part, a post-experiment questionnaire, I asked programmers about their experiences using Stench Blossom, as well as their opinion about smell detectors in general. Specifically, I asked programmers to rate whether the 9 usability guidelines were important, and whether the smell detector adhered to those guidelines.

Additionally, the questionnaire asked programmers to rate two other guidelines that a smell detector might exhibit but that I did *not* postulate in Section 5.2:

- *Decidability*, the property that the tool should help the programmer decide whether to remove a smell. This is similar to Shneiderman's recommendation for constructive guidance [70, p. 58].
- *Consistency* the property that the tool should have a user interface consistent with the rest of the environment. This derives directly from Nielsen's "consistency and standards" heuristic [55].

I included these two guidelines because I hypothesize that they are *not* important to smell detectors, and thus they can provide a baseline against which to test the guidelines that I do postulate.

5.4.3 Results

5.4.3.1 Quantitative Results

Regarding the first task — recognizing smells in code — the median number of smells found without the assistance of Stench Blossom was 11, while the median number of smells found with the assistance of Stench Blossom was 21. The difference between smells found with Stench Blossom and those found without is statistically significant ($p = .003$, $df = 11$, $z = 2.98$, using a Wilcoxon matched-pairs signed-ranks test). This aligned with subjects' opinions: all indicated that it was difficult to look for all 8

smells at once. All subjects indicated that the smell detector found information that they would not have found as quickly. Eight of the twelve indicated that the detector found information that they would not have found at all. All subjects indicated that Stench Blossom was useful for the given tasks, and all but one indicated that they would use the smell detector when they code, if it were available. The sole dissenter did not say why she would not use the tool.

Table 5.3 on the following page lists how subjects rated each guideline that I postulated in Section 5.2. In the left column, after the guideline name, the guideline description is listed as it appeared in the post-experiment questionnaire (the name of each guideline did not appear). The middle major column lists how the subjects rated each guideline in general; the questionnaire labeled this column as “How important is the characteristic to any smell detection tool?” The right major column lists how well Stench Blossom obeyed that guideline; the questionnaire labeled this column as “Do you agree that the characteristic applies to the tool you just used?” In the questionnaire subjects marked one entry in each major column. In Table 5.3, the aggregates of all responses are displayed; the darker the table cell, the more participants marked that response. In Table 5.3, I order guidelines primarily with the highest mean guideline scores appearing first, and secondarily by the highest mean obedience scores.

Guidelines that were not included in the originally postulated list of 9 guidelines (Section 5.2) are *italicized* in Table 5.3. Note that subjects tended to rank the postulated guidelines, as a whole, significantly higher than the guidelines that I did not postulate ($p < .001$, $df = 130$, $z = 3.69$, using a Wilcoxon rank-sum test), suggesting that programmers do indeed believe that my guidelines are generally important to usable smell detectors.

Guideline	General Importance					Tool Obedience				
	Not Important	Somewhat Important	Important	Very Important	Essential	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
Unobtrusiveness: The tool should not block me from my other work while it analyzes or finds smells.	0	0	0	1	11	0	0	1	2	9
Context-Sensitivity: The tool should tell me first and foremost about smells related to the code I'm working on.	0	1	1	3	7	0	2	1	4	5
Scalability: The tool should not overwhelm me with the smells that it detects.	0	1	1	3	7	0	2	4	2	4
Bias: The tool should emphasize smells that are difficult to see with the naked eye.	0	1	0	6	5	0	0	1	4	7
Estimability: The tool should help me estimate the extent of a smell in the code.	0	0	3	3	6	0	0	1	6	5
Task-centricity: The tool should not distract me.	0	1	1	5	5	0	2	0	5	5
Relationality: When showing me details about code smells, the tool should show me the relationships between effected program elements.	1	1	3	4	3	1	1	1	7	2
Availability: The tool should make smell information available to me at all times.	1	2	2	4	3	0	0	1	4	7
Consistency: The tool should have a user interface consistent with the rest of the environment.	1	2	2	5	2	1	2	2	5	2
Expressiveness: In addition to finding smells for me, the tool should tell me why smells exist.	3	0	3	3	3	1	1	3	6	1
Decidability: The tool should help me decide whether to remove a smell from the code.	3	2	4	2	1	1	4	2	4	1

Table 5.3: Post-experiment results regarding guidelines.

5.4.3.2 How Smells were Identified *without* Stench Blossom

When I asked subjects to look for the 8 smells in the code, subjects reported that they found it difficult to keep them all in mind at once. Overall, 4 subjects "somewhat agreed" and 8 "strongly agreed" that "it was difficult to look for all 8 smells at the same time." While looking for smells, a subject remarked "I realize [that] I forgot about the LONG METHOD one" and "TYPECAST: I'd totally forgotten," even though this subject had reviewed the smells less than 10 minutes prior and was among the 3 programmers who rated themselves most knowledgeable about code smells. Likewise, even when readily apparent by inspecting code, some smells were sometimes overlooked by programmers. For example, after overlooking a `switch` statement several times, one programmer commented "I can't believe I didn't see it."

Both when looking for code smells and when judging whether or not to refactor, subjects sometimes used simple heuristics for analyzing code rather than full analysis. Consider DATA CLUMPS, where, for example, the parameters `int a`, `int b`, `int c`, `int d`, `int e`, `int f`, all appear in several different method declarations. During the experiment, when asked about why she said that DATA CLUMPS were present in the source code, one subject said that a method declaration contained six parameters, yet she did confirm that any of those six parameters were repeated in other method declarations. The programmer may not have done this check because it would be too time consuming to do so; indeed, another programmer remarked "it's hard to find data clumping...I have to go back and forth" between method declarations.

Likewise, programmers used heuristics for making judgements about FEATURE ENVY, where, for example, the members of some class are referenced many times outside of that class in "envying" code. One explicitly mentioned heuristic was that if the method being inspected "is static... [then] we're not referencing... this class." This heuristic is useful in that a static method cannot reference instance members, but

fails when the method references static members. Another heuristic, used by several programmers, attempts to find referenced classes by assuming that each variable is of a different class. This works when each variable is of a different class, but is not effective when several variables are of the same class, or for member references to the super class, which appear identical to references to the current class. For example, this code snippet encountered during the experiment from `javax.print` contains a variable `flavor` that is tested to see whether it meets one of several different properties:

```
public Object getSupportedAttributeValues(...,DocFlavor flavor,...){
  ...
  if (flavor == null ||
      flavor.equals(DocFlavor.SERVICE_FORMATTED.PAGEABLE) ||
      flavor.equals(DocFlavor.SERVICE_FORMATTED.PRINTABLE) ||
      flavor.equals(DocFlavor.BYTE_ARRAY.GIF) ||
      flavor.equals(DocFlavor.INPUT_STREAM.GIF) ||
      flavor.equals(DocFlavor.URL.GIF) ||
  ...
```

Using the heuristic, programmers were often unable to recognize the full extent of the FEATURE ENVY in this code. If you use the heuristic, you would look at all the messages sent to the variable `flavor`, and conclude that the code is referencing `flavor`'s class's members exactly 5 times (specifically, 5 references to `equals`). What you might fail to notice, however, is that `flavor`'s class is `DocFlavor`, exactly the same `DocFlavor` whose static members are referenced 5 other times (specifically, two references to `SERVICE_FORMATTED`, one to `BYTE_ARRAY`, one to `INPUT_STREAM`, and one to `URL`). In total, the code is actually referencing the `DocFlavor` class 10 times, using both an instance method and several static fields. Indeed, it appeared that several subjects did not recognize this, and proposed refactorings that produced code similar to this:

```
public Object getSupportedAttributeValues(...,DocFlavor flavor,...){
  ...
  if (flavor == null ||
     flavor.isSupported(new DocFlavor[]{
                  DocFlavor.SERVICE_FORMATTED.PAGEABLE,
                  DocFlavor.SERVICE_FORMATTED.PRINTABLE,
                  DocFlavor.BYTE_ARRAY.GIF,
                  DocFlavor.INPUT_STREAM.GIF,
                  DocFlavor.URL.GIF,
  ...
public class DocFlavor{
  ...
  public boolean isSupported(DocFlavor[] flavors){
   for(DocFlavor flavor : flavors)
      if(this.equals(flavor))
          return true;
   return false;
  }
  ...
```

In the above code, programmers did not realize that they did not need to pass in an array of `DocFlavor`s to the `isSupported` method[3]. Instead, `isSupported` could be responsible for those `DocFlavor`s itself, suggesting the more concise and more cohesive:

[3]Like all refactoring decisions, this isn't necessarily the "right" decision. Subjects had the choice of passing in `DocFlavors` or not; the point here is that subjects appeared *unaware of the choice* when making the refactoring decision.

```
public Object getSupportedAttributeValues(...,DocFlavor flavor,...){
  ...
  if (flavor == null ||
      flavor.isSupported()){
  ...
public class DocFlavor{
  public boolean isSupported(){
     return  equals(SERVICE_FORMATTED.PAGEABLE) ||
             equals(SERVICE_FORMATTED.PRINTABLE) ||
             equals(BYTE_ARRAY.GIF) ||
             equals(INPUT_STREAM.GIF) ||
             equals(URL.GIF) ||
    ...
  }
}
```

At times, then, such code analysis heuristics used by programmers led them to results that may not be optimal.

5.4.3.3 How Smells were Identified *with* Stench Blossom

Subjects confirmed that code smells were highly subjective. For example, several programmers had different definitions of what "too big" means for LONG METHOD and LARGE CLASS. Several subjects agreed with Stench Blossom — that counting the number of characters is a useful for gauging how long something is — although some commented that the tool should not have included comments when gauging size. Some subjects stated that counting statements or expressions in the abstract syntax tree is the only useful metric for length. One subject noted that "if it fits on the page, it's reasonable." Likewise, programmers made comments indicating that smells were not binary, but encompassed a range of severities; for instance, smells were "borderline," "obvious," or "relative" to the surrounding code.

During the experiment, I observed that, when subjects are looking for smells using Stench Blossom, they sometimes ignored the smell indicators. Several subjects

ignored the visualization when the petals were small; others sometimes ignored the visualization regardless of the petal size.

I observed subjects use Stench Blossom to find smells in two different ways. As I expected, some subjects looked at the visualization, then at the code to see if they agreed with what the visualization was telling them. However, some subjects looked at the code first, then looked at the visualization to see if the tool confirmed their findings.

Judging as an external observer, it appeared to me that subjects made refactoring judgements about FEATURE ENVY with about equal confidence with and without the tool. However, upon questioning, 10 out of 12 subjects said that the tool improved their confidence in their refactoring judgement, and 11 out of 12 said that the tool helped them to make more informed judgements.

5.4.3.4 Suggestions for Tool Improvements

Subjects made many suggestions on what could be improved in Stench Blossom. A frequent request was the need for configurability, which the tool does not currently provide (although this was mentioned by Mealy and colleagues [45]). This was especially true of the size of the petals; some programmers felt that the petals were sometimes overstating the smells, sometimes the opposite. A machine learning approach (such as neural networks) combined with a direct manipulation of the visualization (such as dragging the radius of the petal when the programmer disagrees with it) may help provide configurability in a way that is unobtrusive.

Some programmers wanted to be able to zoom out on the code when using the *Explanation View*, often because the code under consideration spanned more than one screen. While it is difficult to have both the ability to zoom out and the ability to edit code at the same time, it may be sufficient to reduce the font size of the code on demand to achieve this result.

The inclusion of LARGE CLASS with the rest of the smells proved problematic for some programmers, largely because it is class-wide whereas the other smells are only method-wide. This difference in scope is a problem because LARGE CLASS and the other smells are presented in the same way. Behaviorally, this problem manifested itself in that some programmers repeatedly looked at the LARGE CLASS petal, even though it did not change as the programmer navigated from method to method within the same class.

While I designed the smell detector to provide quick access to detailed smell information, several programmers desired even faster access. One programmer commented that the detector should forgo the *Active View* altogether; when a petal is moused-over, it should show *Explanation View* immediately. The cost of this modification would be that the programmer may inadvertently activate the *Explanation View* when moving the mouse near the right scroll bar. Another programmer commented that all smell names, not just from one petal, should be displayed on mouse over. This modification may make it difficult for the programmer to visually associate a smell name with a specific petal. While both of these suggestions would entail such design tradeoffs, they both also increase *information efficiency* [65], and thus are worthy of consideration for future smell detector design.

5.4.4 Threats to Validity

There are several threats to validity in this experiment. First, the experiment was not comparative; I did not compare my smell detector against an existing smell detector. As a consequence, I cannot claim that Stench Blossom is better than an existing tool. I did not perform a comparative experiment for two reasons:

- There is no representative existing tool to compare against. Although there are Eclipse smell detector plugins that utilize the underlining interface that I

could compare Stench Blossom against, I am not aware of any Eclipse smell visualizations.

- Any comparison against an existing tool would require the experiment to work within the boundaries of the intersection of all the tools, a space that would be extremely small. For instance, van Emden and Moonen's tool [13] implements only two smells (INSTANCEOF and TYPECAST); having an experiment with only these two smells would produce only very limited results.

A second limitation is that the experiment simulated two inspection tasks, yet I have claimed Stench Blossom is more useful for coding tasks. I did not perform an experiment with coding tasks for several reasons:

- Performing a coding task requires that the code and coding task is sufficiently simple that it can be rapidly learned, but this requirement conflicts with two other requirements of the experiment: programmers must traverse a sufficiently large and smelly code base so that they can observe a large number of smells.

- During a realistic coding task, there is no guarantee that the programmer will interact with the tool sufficiently frequently to allow them to make informed judgments about it. Indeed, I would not expect programmers to be interacting with this tool more than 10% of the time. By having the programmers use the tool frequently, I have higher confidence that programmers made informed judgements.

The biggest difference between coding and inspecting is that when coding, programmers would likely be more focused on their coding task than on looking for defects or smells in code. There is a danger, then, that programmers would either be too distracted by the tool while coding, or wouldn't notice the tool if it was trying to convey information. Thus, I asked programmers to estimate if the tool would be too

distracting or whether it would get their attention at the right time while coding. Ten out of twelve programmers estimated that the tool would not be too distracting, while ten out of twelve estimated that it would get their attention at the right time.

Another threat to validity in this experiment is that the 6 students received extra credit for participating, and thus I may have attracted a non-representative sample of students who were performing poorly in the class and needed the extra credit. Again, this is a general threat to all experiments that offer extra credit for participation. However, I attempted to counterbalance this somewhat by taking a stratified sample where half of the participants were uncompensated volunteer programmers.

A final threat to validity is that subjects may have biased the results by wanting to please the experimenter. While this is a threat to many experiments, it is especially salient in this experiment because much of my results are derived from programmers' estimations and opinions, rather than direct measurements how they used this tool. This is a side-effect of smell detectors in general, because there is no an objective standard of what is the "right" response from a programmer when it comes to smells. For example, if a programmer comes across a method that Stench Blossom says is very long and the programmer says nothing about it exhibiting a LONG METHOD smell, I cannot conclude that the tool failed, because she may have judged that the method is small enough in her opinion.

An attempt that I made to mitigate this threat was not to disclose that I had built Stench Blossom myself[4] until after the experiment, so that programmers would be less likely to withhold opinions to avoid hurting my feelings. Likewise, most opinions that I asked of the programmers were written down on paper by the subjects themselves, and I did not read what the subject wrote until after they had left the experiment room. Thus, I believe that I mitigated this risk as much as was reasonably possible.

[4] One exception to this occurred when a subject asked in the middle of the experiment whether I made the tool myself.

5.4.5 Discussion

Looking at the results, it appears that Stench Blossom is useful for locating smells and for helping programmers to make more informed, confident, refactoring judgements. The results suggest that the tool has value in general to ease the mental burden of remembering several smells at once, even when those smells are clearly present in the code. Moreover, it appears that the tool also has value in accurately bringing together relevant information for making refactoring judgements, and avoiding the need for error-prone heuristics.

On the matter of whether my postulated guidelines are useful guidelines for building more usable smell detectors, the results appear positive. In general, most of my guidelines appear to be rated as important by the programmers, although a minority of programmers appear to believe that some guidelines are not at all important. For example, the postulated guideline that was judged the least important, expressiveness, was judged as "not important" by 3 subjects. Interestingly, these 3 programmers were all volunteers from the classroom, and were the second, third, and fourth least experienced programmers among the 12 subjects. This suggests that, perhaps, less experienced programmers do not value a tool that can explain its reasoning, and believe that needing such an explanation is a sign of poor programming ability.

Programmers also appear to believe that Stench Blossom obeys these guidelines, again with some disagreement between individuals. The two guidelines over which there was the most disagreement were whether tool was scalable and expressive; the reason for this disagreement is unclear.

After observing programmers use Stench Blossom, I am more convinced that code highlighting for communicating smells is not an effective mechanism for communicating smells to programmers. This is because programmers have a wide variety of opinions on what smells bad and what does not. Any binary indicator such as underlining, would either miss or overemphasize smells that the programmer disagrees

with. Thus, such false-negatives and false-positives may erode programmers' trust in the tool, making them less likely to use it in the future.

5.5 Future Work

The need for configurability was a frequent request among experiment subjects, so research into how to make this efficient is future work. Further work on the balance between getting a programmer's attention and not being too distracting is also necessary. Small bugs and improvements were revealed during the experiment as well, and are deserving of further attention. Finally, a longer-term evaluation of the smell detector may reveal more interesting usage patterns, especially whether the tool can get a programmer's attention at the right times.

5.6 Conclusions

In this chapter, I have addressed the usability of tools that help programmers during the *identify* step if the refactoring process (Section 2.5). I introduced guidelines for building tools to help programmers identify opportunities for refactoring that align with how programmers refactor, guidelines derived from the strengths of the user-interfaces of existing smell detectors. I also described how these guidelines could be built into a single tool as a behind-code, always-up-to-date visualization. The evaluation that I presented in this chapter suggests that my smell detection tool, Stench Blossom, can help programmers to find and understand more smells with greater confidence. The evaluation also suggests that programmers value the guidelines, and that my smell detector aligns well with those guidelines.

Chapter 6

The Selection Step: Communicating What Code to Refactor [1]

Every use of a refactoring tool requires the programmer to choose program elements to be refactored. This is commonly done by selecting characters in an editor. As the formative study described in Section 4.5 demonstrated, selecting code as input to a refactoring tool can be surprisingly difficult for programmers. In this chapter I explore how to build tools that help programmers during the *selection* step of the refactoring process (Section 2.5).

6.1 Contributions

The major contributions of this chapter are as follows:

- Two new tools to help programmers select program statements for input to refactoring tools (Section 6.2).
- An experiment that demonstrates that these two tools can help programmers reduce code selection errors by 84 percent and 95 percent, and improve selection speed by 46 percent and 24 percent (Section 6.3).
- Guidelines for making tools that help programmers select code for refactoring (Sections 6.4 and 6.6.2).

[1] Parts of this chapter appeared in the *Proceedings of the 2008 International Conference on Software Engineering* [49] and as part of my thesis proposal.

```
boolean isWellDressed(){
    if(jersey.isSpandex()){
        return shorts.isSpandex();
    }
    return true;
}
```

Figure 6.1: The Selection Assist tool in the Eclipse environment, shown covering the entire `if` statement, in green. The user's selection is partially overlaid, darker.

- A user interface, called refactoring cues, that eliminates syntax selection errors, keeps the programmer's focus in the editor and enables selection of multiple program elements for refactoring (Section 6.6.3).

6.2 Tool Description

In this section, I describe two tools that I have built for the Eclipse environment [18] that address the problem of accurately selecting code as input to the EXTRACT METHOD refactoring demonstrated in the formative study (Chapter 4). These tools are built to *prevent* the selection errors encountered during that study, as recommended by Nielsen: "even better than good error messages is a careful design which prevents a problem from occurring in the first place" [55]. You can download the tools and view a short screencast here: http://www.multiview.cs.pdx.edu/refactoring.

6.2.1 Selection Assist

The Selection Assist tool helps programmers in selecting whole statements by providing a visual cue of the textual extent of a program statement. The programmer begins by placing the cursor in the white space in front of a statement. A green highlight is then displayed on top of the text, from the beginning to the end of a statement, as shown in Figure 6.1. Using the green highlight as a guide, a programmer can then select the statement normally with the mouse or keyboard.

This tool is similar to tools found in other development environments. DrScheme,

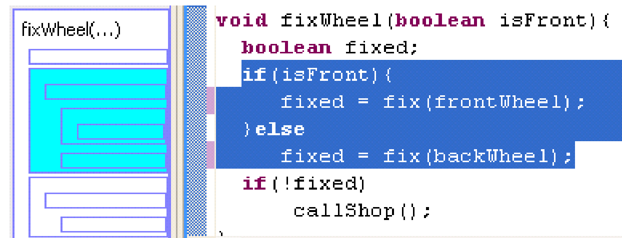


Figure 6.2: Box View tool in the Eclipse environment, to the left of the program code.

for example, highlights the area between two parentheses in a similar manner [16], although that highlighting disappears whenever cursor selection begins, making it ineffective as a selection cue. Vi and other text editors have mechanisms for bracket matching [34], but brackets do not delimit most statements in Java, so these tools are not always useful for selecting statements. Some environments, such as Eclipse, have special keyboard commands to select statements, but some programmers prefer the mouse. Selection Assist allows the programmer to use either the mouse or the keyboard for selection tasks.

6.2.2 Box View

I designed a second tool to assist with selection, called Box View; it displays nested statements as a series of nested boxes. Box View occupies a panel adjacent to program text in which it displays a uniform representation of the code, as shown in Figure 6.2. Box View represents a class as a box with labeled method boxes inside of it. Inside of each method are a number of nested boxes, each representing a nested statement. When the programmer selects a part of a statement in the editor, the corresponding box is colored orange. When the programmer selects a whole statement in the editor, the corresponding box is colored light blue. When the programmer selects a box, Box View selects the corresponding program statement in the program code.

Like Selection Assist, programmers can operate Box View using the mouse or keyboard. Using the mouse, the programmer can click on boxes to select code, or

select code and glance at the boxes to check that the selection includes only full statements (contiguous light blue). Using the keyboard, the programmer can select sibling, parent and child statements. Box View was inspired by a similar tool in Adobe GoLive (http://www.adobe.com/products/golive) that displays an outline of an HTML table.

Box View scales fairly well as the level of statement nesting increases. In methods with less than ten levels of nesting, Box View requires no more screen real estate than the standard Eclipse Outline View. In more extreme cases, Box View can be expanded horizontally to enable the selection of more deeply nested code.

6.3 Evaluation

Having demonstrated that programmers have difficulty in selecting code as input to EXTRACT METHOD tools (Chapter 4) and having proposed two new selection tools as solutions, I conducted a study to ascertain whether or not the new tools overcome this usability problem.

6.3.1 Subjects

I drew subjects from Professor Andrew Black's object-oriented programming class. Professor Black gave every student the option of either participating in the experiment or reading and summarizing two papers about refactoring. In all, 16 out of 18 students elected to participate. Most students had around 5 years of programming experience and three had about 20 years.

About half the students typically used integrated development environments such as Eclipse, while the other half typically used editors such as vi [34]. All students were at least somewhat familiar with the practice of refactoring.

6.3.2 Methodology

The experiments were performed over the period of a week, and lasted between $\frac{1}{2}$ and $1\frac{1}{2}$ hours per subject. The subjects first used three selection tools: mouse and keyboard, Selection Assist, and Box View. Subjects were randomly assigned to one of five blocks; code and tools was presented in a different, randomized order between blocks. I chose code from the open source projects described in Chapter 4. Each subject used every tool.

When subjects began the experiment, the test administrator showed them how to use the first of the three selection tools, depending on which block she was assigned to. The administrator demonstrated the tool for about a minute, told subjects that their task was to select all `if` statements in a method, and then allowed them to practice the task using the selection tool until they were satisfied that they could complete the task (usually less than 3 minutes). Subject were then told to perform the task in 3 different methods from 3 classes; the methods contained about two dozen `if` statements in total. I classified a selection as correct when it spanned from just before the “i” in `if` to just after the `if` statement’s closing bracket “}”, being permissive of any additional selected whitespace. I classified all other selections as mis-selections, such as when the selection does not include the closing bracket. The training session and the selection task were then repeated for the two other tools on two different code sets.

6.3.3 Results

Table 6.1 on the next page shows the combined number of `if` statements that subjects selected correctly and incorrectly with each tool. Table 6.2 on the following page shows the mean time across all participants to select an `if` statement, and the time normalized as a percentage of the selection time for the mouse and keyboard.

From Table 6.1, you can see that there were far more mis-selections using the

	Total Mis-Selected `if` Statements	**Total Correctly Selected `if` Statements**
Mouse/Keyboard	37	303
Selection Assist	6	355
Box View	2	357

Table 6.1: Total number of correctly selected and mis-selected `if` statements over all subjects for each tool.

	Mean Selection Time	**Selection time as Percentage of Mouse/Keyboard Selection Time**
Mouse/Keyboard	10.2 seconds	100%
Selection Assist	5.5 seconds	54%
Box View	7.8 seconds	76%

Table 6.2: Mean correct selection time over all subjects for each tool.

mouse and keyboard than using Selection Assist, and that Box View had the fewest mis-selections. Table 6.2 indicates that Selection Assist decreased mean selection time from 10.2 seconds to 5.5 seconds (46% faster), and that Box View decreased selection time to 7.8 seconds (24% faster). Both speed increases are statistically significant ($p < .001$, using a t-test with a logarithmic transform to normalize long selection-time outliers).

The left graph in Figure 6.3 on the next page shows individual subjects' mean times for selecting `if` statements using the mouse and keyboard against Selection Assist. Here you can see that all subjects but one (labeled 'a') were faster using the Selection Assist than using the mouse and keyboard (subjects below the dashed line). You can also see that all subjects but one (labeled 'b') were more error prone using the mouse and keyboard than with Selection Assist. The difference in error-rate is statistically significant ($p = .003$, $df = 15$, $z = 3.01$, using a Wilcoxon matched-pairs signed-ranks test).

The right graph in Figure 6.3 on the following page compares the mouse and keyboard against Box View. Here you see that 11 of the 16 subjects are faster using

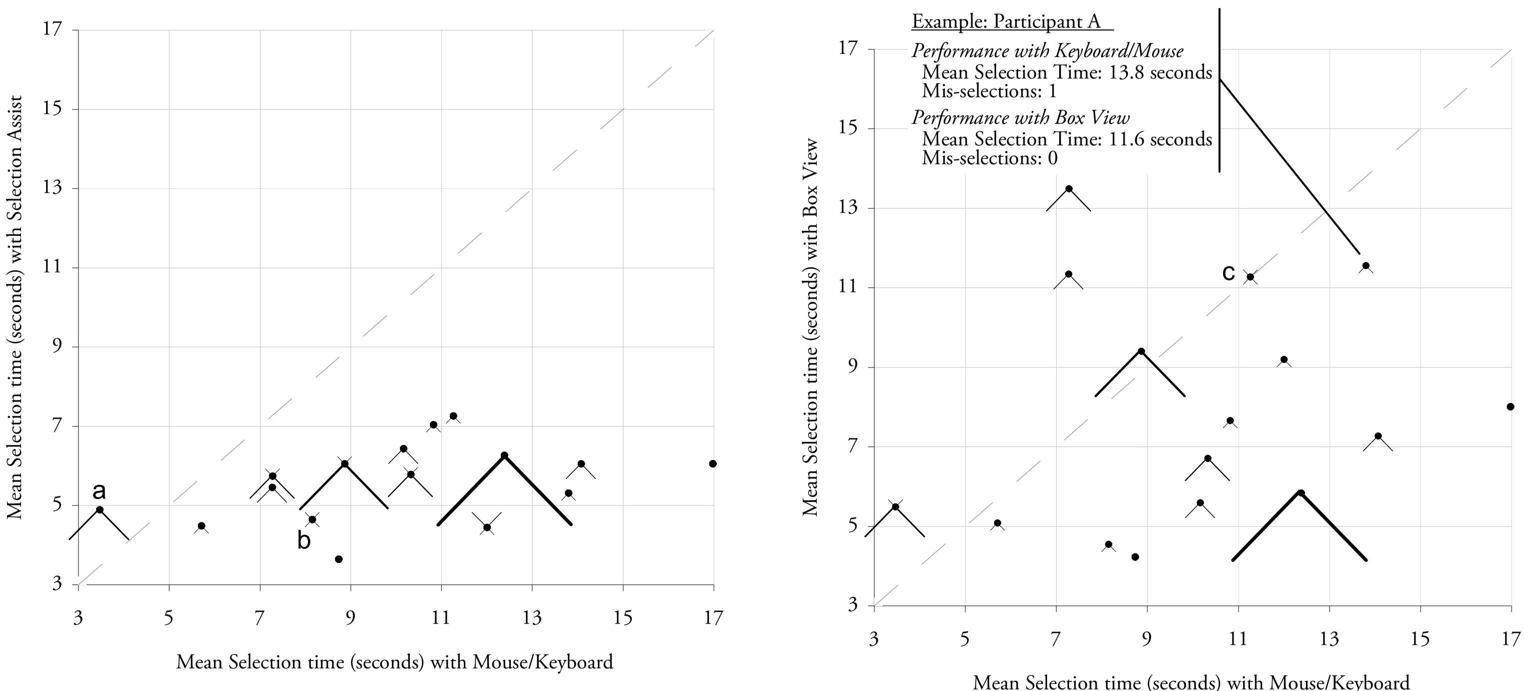


Figure 6.3: Mean time in seconds to select `if` statements using the mouse and keyboard versus Selection Assist (left) and Box View (right). Each subject is represented as a whole or partial X. The distance between the bottom legs represents the number of mis-selections using the mouse and keyboard. The distance between the top arms represents the number of mis-selections using Selection Assist (left) or Box View (right). Points without arms or legs represent subjects who did not make mistakes with either tool.

Box View than using the mouse and keyboard. You can also see that all subjects except one (labeled 'c') are less error prone with Box View. The difference in error-rate is statistically significant ($p = .001$, $df = 15$, $z = 3.35$, using a Wilcoxon matched-pairs signed-ranks test).

I administered a post-test questionnaire that allowed the subjects to express their preferences among the three tools. The survey itself and a summary of the responses can be found in my technical report [48]. Significance levels are reported using a two-tailed Wilcoxon matched-pairs signed-ranks test.

Most users did not find the keyboard or mouse alone helpful in selecting `if` statements, and rated the mouse and keyboard significantly lower than either Box View ($p = .001$, $df = 15$, $z = 3.25$) or Selection Assist ($p = .002$, $df = 15$, $z = 3.13$). All users were either neutral or positive about the helpfulness of Box View, but were divided about whether they were likely to use it again. Selection Assist scored the highest of the selection tools, with 15 of 16 users reporting that it was helpful and that they were likely to use it again.

Overall, the subjects' responses showed that they found Selection Assist and Box View superior to their traditional counterparts for the tasks given to them. More importantly, the responses also showed that the subjects felt that the new tools would be helpful outside the context of the study.

6.3.4 Threats to Validity

Although the quantitative results discussed in this section are encouraging, several factors must be considered when interpreting these results.

Every subject used every tool, but a flaw in the study design caused the distribution of tools to code sets to be uneven as shown in Table 6.3 on the next page. In the most extreme instance of unevenness, one code set was traversed only twice with the mouse and keyboard while another code set was traversed eight times using Selection

Tool	Code Set A	Code Set B	Code Set C
Mouse/Keyboard	6	6	4
Selection Assist	2	6	8
Box View	8	4	4

Table 6.3: The number of times subjects used each tool to select `if` statements in each code set.

Assist. However, because each code set was chosen to be of roughly equal content and difficulty, I do not believe this biased the results in favor of any particular tool.

The experiment tested how well programmers can use tools to select code, but tool usability is also affected by factors that I did not test. For example, while Box View is more accurate than Selection Assist, Box View takes up more screen real estate and requires switching between views, which may be disorienting. In short, each tool has usability tradeoffs that are not visible in these results.

Finally, the code samples selected in these experiments may not be representative. I tried to mitigate this by choosing code from large, mature software projects. Likewise, the programmers in this experiment may not be representative, although the subjects reported a wide variety of programming experience.

6.3.5 Discussion

Both Box View and Selection Assist help programmers to select code quickly and accurately. Box View appears to be preferable when the probability of mis-selection is high, such as when statements span several lines or are formatted irregularly. Selection Assist appears to be preferable when a more lightweight mechanism is desired and statements are less than a few lines long.

6.4 Guidelines

The tools described in this chapter are demonstrably faster, more accurate, and more satisfying to use. However, they represent only a small contribution: they are improvements to only one out of dozens of refactoring tools. Nevertheless, I reason that the interaction techniques embodied in these tools are applicable to almost all refactoring tools because most refactoring tools require the programmer to select a piece of code to be refactored.

By studying how programmers use existing refactoring tools and the new tools that I have described in this chapter, I have induced a number of usability guidelines for refactoring tools. In this section, I link my experiment and the design of my tools to each guideline.

Users can normally select code quickly and efficiently, and any tool to assist selection should not add overhead to slow down the common case. Box View adds context switching overhead from the editor to the view, which I believe contributed to its relative slowness and lower likeliness-to-use-again rating, as compared to Selection Assist. Thus:

- ◇ **Task-centricity.** A tool that assists in selection should not distract from the programmer's primary task.

Both Box View and Selection Assist work regardless of the format of code; in particular, Box View abstracts away formatting completely by displaying statements uniformly. Thus:

- ◇ **Uniformity.** A tool that assists in selection should help the programmer to overcome unfamiliar or unusual code formatting.

The accuracy improvement of Box View over Selection Assist appears to be because the only possible selections in Box View were sequences of whole program

statements. Thus:

◇ **Atomicity.** A tool that assists in selection should help eliminate as many selection errors as possible by separating character-based selection from program-element selection: the only possible selections should be those that are valid inputs to the refactoring tool.

Because standard editor selection is task-agnostic, programmers made selection errors during the experiment. Conversely, because Box View and Selection Assist are optimized for EXTRACT METHOD, they reduced selection errors. Thus:

◇ **Task-specificity.** A tool that assists in selection should be task specific.

While these guidelines may seem self-evident, they are rarely implemented in contemporary refactoring tools. These guidelines, however, are specific versions of the general usability principle of avoiding errors [71, p. 63].

6.5 Related Work: Alternative Selection Techniques

O'Connor and colleagues implemented an EXTRACT METHOD tool using a graph notation to help the programmer recognize and eliminate code duplication [57]. This approach avoids selection mistakes by presenting program structure as an abstract syntax tree where nodes are the only valid selections.

Several development environments, including Eclipse, include editor commands that enable the programmer to expand or contract their current selection to surrounding or contained program elements. When the programmer has not selected a whole statement, EXTRACT METHOD in Eclipse sometimes suggests that the programmer execute these commands to correct the selection. Selection Assist could be paired with this suggestion to provide visual feedback as to what currently is and what would be selected if the programmer were to accept the suggestion.

An outline, such as the Eclipse Outline View, displays classes, methods and fields as a hierarchy of icons, usually in a pane adjacent to the program text. Any program element can be selected merely by clicking on its icon, an operation that is more error resistant than text selection in an editor. Icon selection also allows multiple elements to be selected for refactoring. However, multiple icon selection has two chief disadvantages. First, it does not apply to program elements at a finer granularity than presented by the outline, such as statements, as needed for EXTRACT METHOD. Second, it requires the programmer to change focus from the text view to the outline view, which may itself slow the programmer down.

In iXj, a programmer can select several program elements at once using a program transformation language based on examples [4]. Code is selected by first mouse-selecting an example in the editor and then generalizing that example using the mouse in a separate view. This interaction may not be fast enough for floss refactoring. Furthermore, while iXj can assist in selecting some kinds of program elements, such as expressions, it does not help select every kind of program element that a programmer might want to restructure.

In summary, while previous approaches have improved the accuracy of selection and added the ability to select several program elements at once, they cannot be generalized to the selection of every kind of program element.

6.6 Generalization to Other Refactorings

So far, I have introduced two tools to help programmers select code appropriate for refactoring: Selection Assist and Box View. However, neither tool follows every guideline in Section 6.4. Furthermore, while programmers using these tools demonstrated significant improvements in both speed and accuracy of selection, the tools were limited to just one refactoring. In this section, I describe a running example that I use in this and subsequent chapters (Section 6.6.1), two new selection guide-

lines (Section 6.6.2), and a tool that implements all of my selection guidelines (Section 6.6.3).

6.6.1 A Running Example

In the remainder of this chapter, and in Chapters 6 through 8, I use a small program restructuring as a running example. It is an EXTRACT LOCAL VARIABLE refactoring.

Suppose that you start with this code in a video game application:

```
characterCount = 4 + 1;
```

Now suppose you realize that you want to say explicitly that `4` is the number of ghosts in your game. So you would like to extract `4` into a local variable with a meaningful name, to produce this code:

```
int ghostCount = 4;
...
characterCount = ghostCount + 1;
```

I will use this EXTRACT LOCAL VARIABLE refactoring to motivate some additional guidelines and in several tool examples that follow.

6.6.2 Two More Selection Guidelines

In addition to the guidelines presented in Section 6.4, here I present two additional guidelines. I did not introduce these guidelines earlier in this chapter because their need was not apparent until after I performed the formative study[2] (Section 4.5).

The programmer may not know what selections are valid inputs to the tool, especially if she has not previously used that tool. For instance, if a programmer wants to

[2] The need for these guidelines was not apparent in the study by the nature of the task that I assigned to participants.

move a method up into a superclass (the PULL UP refactoring), it may not be clear whether she should select the whole method declaration, the method heading, or the method name. To make clear what to select to perform a refactoring, I propose that

- ◇ **Explicitness.** A refactoring tool should allow the programmer to explicitly see what the tool expects as input before selection begins.

This guideline is somewhat similar to Nielsen's "recognition rather than recall," which states that a tool should minimize a user's memory load [55].

Selecting multiple program elements for refactoring is usually impossible with present-day refactoring tools. For instance, I know of no existing tool that can extract both the constants `4` and `1` of the running example into variables in a single step. However, it seems that programmers would benefit from this ability, as I observed two data sets where refactoring tool uses appear in such groups 30% and 47% of the time (Section 3.4.2). Based on these observations, I propose that a refactoring tool

- ◇ **Multiplicity.** A tool that assists in selection should allow the programmer to select several program elements at once.

This guideline is a special case of Shneiderman's "minimal input actions by the user," which states that "fewer input actions means greater operator productivity" and "redundant data entry should be avoided" [71].

6.6.3 Tool Description: Refactoring Cues

In this subsection, I describe a tool called refactoring cues that is used for selecting code and configuring refactoring tools. I encourage the reader to watch screencasts, which provide a short but illuminating look at how this tool works in practice (http://multiview.cs.pdx.edu/refactoring/activation).

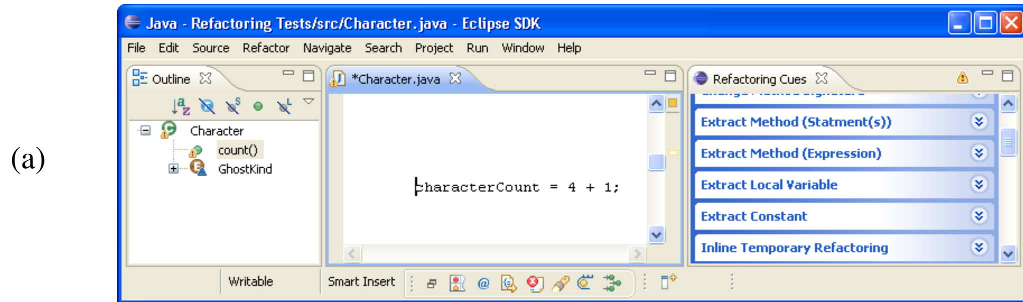
Refactoring cues were designed to improve the selection and configuration steps of the refactoring process (Section 2.5) by taking an "action-first" approach. This means that choosing the refactoring (the action) precedes choosing what to refactor.

The programmer begins by clicking on a refactoring from the adjacent, non-modal *palette* (Figure 6.4a, right pane), which displays the available refactorings. After a refactoring is initiated, two things happen simultaneously. (1) The palette item expands to reveal configuration options (Figure 6.4b, right pane) and (2) *cues* (highlighted program elements with thin borders) are drawn over the program text to show all the program elements to which the selected refactoring can be applied (Figure 6.4b, left).

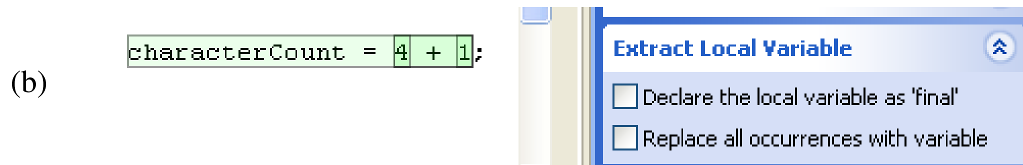
The programmer now chooses specific program elements to be refactored. To choose a program element, she mouse clicks anywhere inside the overlaid cue, an interaction that I call *targeting*. Cues that are not targeted are colored green while those that are targeted (that is, are chosen for refactoring) are colored pink. A cue with nested children can be targeted by clicking on the cue directly, or by clicking repeatedly on one of the cue's children. For example, Figure 6.4c illustrates what happens when the programmer clicks on `4` several times. A targeted cue (pink, center column) indicates the expression to be refactored (right column). If a cue is not targeted (green), then a mouse click targets it (first row). If a cue or one of its ancestors is targeted, then the targeted cue becomes un-targeted and its parent becomes targeted (second and third rows). Clicking within an already-targeted outermost cue un-targets all children (last row).

In the example, the programmer wants to extract the number `4`, so she clicks on `4` once. She then presses a hotkey or button to perform the transformation. At this point, the refactoring engine modifies the code and gives default names to any newly created program elements. At the same time, the configuration options are hidden in the palette and the cues are removed from the editor. Finally, the cursor is positioned

(a)



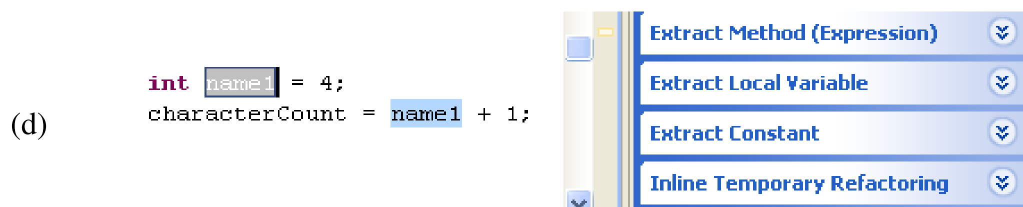
(b)



(c)

Mouse Click	Cue Coloration	Program element to be refactored
First	`characterCount = 4 + 1;`	`4`
Second	`characterCount = 4 + 1;`	`4 + 1`
Third	`characterCount = 4 + 1;`	`characterCount = 4 + 1`
Fourth	`characterCount = 4 + 1;`	`(none)`

(d)



(e)

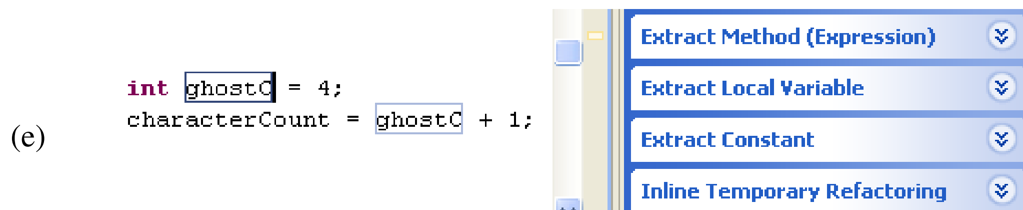


Figure 6.4: The several-step process of using refactoring cues.

```
if(name.equals("blinky")){
    return GhostKind.BLINKY;
}else if(name.equals("pinky")){
    return GhostKind.PINKY;
}else if(name.equals("inky")){
    return GhostKind.INKY;
}else if(name.equals("clyde")){
    return GhostKind.CLYDE;
}
```

Figure 6.5: Targeting several cues (the pink rectangles) at once using a single selection; the programmer's selection is shown by the grey overlay.

to allow the programmer to change the newly created, default-named program elements with a standard Eclipse linked in-line RENAME, as shown in Figure 6.4d and 6.4e.

Although I have described just one interaction with refactoring cues, the user interface allows some flexibility in the way that the programmer interacts with the tool.

- In addition to using the palette, programmers have the option of using any other initiation mechanism, such as hotkeys, linear menus, or pie menus.

- Because initiation happens before selection, multiple program elements are targeted in the same way as a single program element. In the example, this means that, during the selection step, the programmer may click the `4` and then the `1`, indicating that *both* should be refactored in one transformation.

- As an alternative to clicking to target a refactoring cue, a programmer can use an ordinary text selection to target a cue or several cues (Figure 6.5). This technique not only reduces the effort required to target several program elements at once, but it also provides backward compatibility with existing refactoring tools: any selection valid as input to a standard refactoring tool is also valid for refactoring cues. Selection using the keyboard is also supported.

Refactoring cues meet all of the guidelines proposed in Sections 6.4 and 6.6.2. They were designed to reduce selection errors by making all cues acceptable input to the refactoring tool. The technique of targeting program elements for refactoring using visual cues can be applied to all refactorings. Refactoring cues were also designed to be faster than standard refactoring tools because multiple elements can be refactored at once. They make what is refactorable explicit, which may reduce programmer guesswork by making explicit what is refactorable, and also may allow the programmer to remain focused on the text of the program rather than on a configuration panel.

An evaluation of refactoring cues can be found in Section 8.5. The evaluation suggests that programmers can select code at least as quickly and accurately with refactoring cues as with the mouse or keyboard, and that programmers believe that they would use refactoring cues in conjunction with existing tools.

6.7 Conclusions

In this chapter, I have addressed the usability of tools that help programmers during the *selection* step if the refactoring process (Section 2.5). Selecting code as input to a refactoring tool can be a surprisingly difficult task for programmers. Better tools can help programmers in this task by allowing them to select code more quickly and accurately. Such tools can be as conservative as Selection Assist, which is passive and does not change the programmer workflow at all, or can be as progressive as refactoring cues, which change that workflow considerably. In this chapter, I have stated several guidelines that characterize what makes these tools usable, with the hope that toolsmiths will take these guidelines, tools, and supporting evidence into consideration when building the selection mechanisms for the next generation of refactoring tools.

Chapter 7

The Initiation Step: Choosing which Refactoring Tool to Execute

To use a refactoring tool, a programmer must at some point communicate to the programming environment which refactoring that it should perform. I call this the *initiation* step of the refactoring process (Section 2.5).

7.1 Contributions

The major contributions of this chapter are as follows:

- Guidelines for making refactoring tools that are easier to initiate (Section 7.2).
- The use of pie menus in refactoring tools, which provide a fast and memorable user interface for tool initiation (Section 7.4).
- An experiment that explores the relationship between pie menu item placement and memory recall (Section 7.5.2), a contribution that has implications outside the domain of refactoring tools.

7.2 Guidelines

Once a program element has been selected, the programmer must indicate which refactoring tool to initiate, usually with a linear menu or a hotkey (Figure 7.1 on the next page). A linear menu is a system or context menu where items appear in a

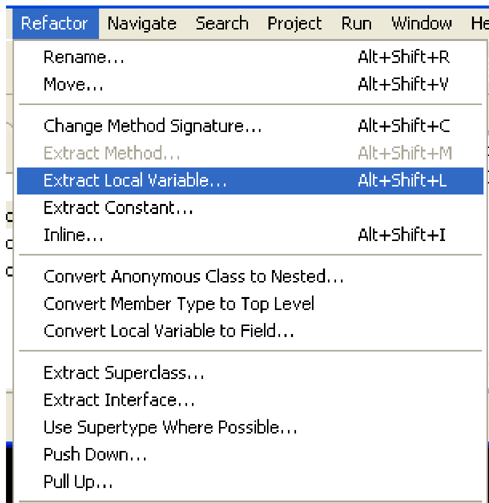


Figure 7.1: Initializing a refactoring from a system menu in Eclipse, with hotkeys displayed for some refactorings.

linear list, possibly with submenus. A hotkey is a combination of key presses on a keyboard.

System menus can be slow, because the user must take significant time to navigate *to* them, then *through* them, and finally *point at* the desired menu item [6]. For refactoring, the problem is worsened when development environments offer a very long list of possible refactorings from which the programmer must choose. As one programmer complained in my Agile survey, the "[refactoring] menu is too big sometimes, so searching [for] the refactoring takes too long." Context menus — which appear at the cursor and whose contents depends on the cursor location — avoid the problem of navigating *to* the menu, but worsen the problem of navigating *through* the menu, because the context menu must be shared with a large number of non-refactoring items.

The slowness of linear menus is a problem during refactoring because using a refactoring tool *must* be fast during floss refactoring (Section 4.7). The speed at which a refactoring can be initiated is critical, suggesting that the tool should:

◇ **Task-centricity.** Initiation of a refactoring tool should not distract from the programmer's primary task.

This guideline is similar to Shneiderman's "enable frequent users to use shortcuts" [71, p. 61] to "increase the pace of interaction."

Using a hotkey might seem to be an ideal way of speeding up the initiation of refactoring tools. However, hotkeys are difficult to remember [24], especially for refactorings. One reason is that the mapping from a code transformation to a hotkey is often indirect. A programmer must take the structural transformation that she wants to perform (such as "take this expression and assign it to a temporary variable"), recall the *name* of that refactoring (EXTRACT LOCAL VARIABLE), and finally map that name to a contrived hotkey (remembering that "Alt+Shift" means refactor, and that "L" is for "Local"). The task is especially difficult because the names of the refactorings are themselves somewhat capricious[1], and because the refactorings must compete with the hundreds of other hotkey commands in the development environment. In Murphy and colleagues' data describing 41 Eclipse developers [47], the median number of hotkeys that programmers used for refactoring was just 2; the maximum number of hotkeys used by any programmer was 5. This suggests that programmers do not often use hotkeys for initiating refactoring tools. Thus:

◇ **Identifiability.** Initiation of a refactoring tool should not rely exclusively on the names of refactorings, but rather use a mechanism that more closely matches how programmers think about refactoring, such as structurally or spatially.

This guideline is similar to Nielsen's heuristic, "Match between system and the real world" [55] by using "concepts familiar to the user, rather than system-oriented terms."

[1]For example, Eclipse's EXTRACT LOCAL VARIABLE is called INTRODUCE EXPLAINING VARIABLE in Fowler's book [22].

7.3 Related Work: Alternative Tool Initiation Techniques

To my knowledge, all existing refactoring tools are invoked with hotkeys, linear menus, or some combination of the two. However, alternative techniques are used to initiate commands outside of the domain of refactoring tools.

My guideline about using an initiation mechanism that matches the structural and spatial nature of refactorings aligns with the directionality of pie and marking menus. Pie menus are special types of context menus where items appear around a circle, rather than in a linear list [6]. Marking menus are pie menus where you can gesture in the direction of item that you want, even before that item is displayed [37]. In the Fabrik programming environment [40], programmers can initiate common programming commands using a mechanism similar to pie menus, commands such as those that connect and disconnect visual program components. However, such menus do not appear to have made inroads into modern Integrated Development Environments.

7.4 Tool Description

In this section, I describe a tool called pie menus for refactoring that are used for initiating a refactoring tool. I encourage the reader to watch my screencasts, which provide a short but illuminating look at how this user interface works in practice (http://multiview.cs.pdx.edu/refactoring/activation).

Pie menus for refactoring[2] are designed to speed the initiation step of the refactoring process. My implementation of pie menus is based on the SATIN implementation [29]. Pie menus are not new; my contribution is the application of pie menus to refactoring and the rules for placing refactorings in particular directions.

To further speed the process, once the refactoring has been initiated using the pie menu, the transformation is executed immediately, rather than displaying a config-

[2]Technically, what I here call pie menus are actually *marking menus*, because they allow the user to gesture to initiate a refactoring.

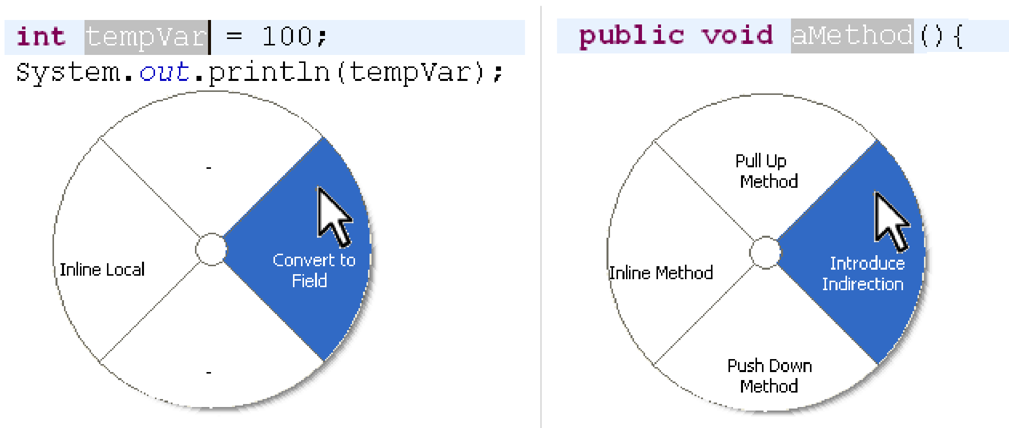


Figure 7.2: Two pie menus for refactoring, showing applicable refactorings for a method and temporary variable, respectively.

uration dialog. Pie menus for refactoring are invoked like a standard context menu, but with a dedicated mouse button or hotkey; the default is the middle mouse button. Pie menus are context sensitive, so different menus are associated with different kinds of program elements, such as statements, method names, and variable names. Figure 7.2 shows two examples.

The advantage of pie menus for refactoring is that the structural nature of many refactorings can be mapped onto the placement of the labels around the pie menu. I use three rules to determine the placement of a refactoring. First, directional refactorings are placed in their respective direction, so, for example, PULL UP is on top. Second, refactorings that are inverses are placed on opposite sides of the menu, so, for example, INLINE METHOD is opposite EXTRACT METHOD. Third, refactorings that are conceptually similar are placed in the same location in different contexts, so, for example, INLINE METHOD and INLINE LOCAL VARIABLE occupy the same quadrant in different menus. Refactoring appears to be one of the applications of pie menus that Callahan and colleagues say "seem to fit well into the mold of the pie menu design," and for which they demonstrated that pie menus have a distinct speed advantage [6].

Moreover, I hypothesize that my placement rules help programmers *recall* or *infer* the location of refactoring menu items, even before the programmer builds muscle memory. This is important for refactorings, because most refactoring tools are currently used infrequently (Section 3.4.7), and so muscle memory would not be an effective recall mechanism for those refactorings. I validate this hypothesis with an experiment described in Section 7.5.2.

My pie menus are restricted to four items with no submenus. The restriction to four items increases the speed at which pie menus can be used [37] and reduces the complexity of user interaction. Furthermore, because some programmers prefer to use the keyboard, the restriction allows programmers to use the menus with a simple hotkey scheme (one button to bring-up the menu and then a press of the up, down, left, or right arrow to invoke an item). In fact, my placement rules could be used without pie menus, in a pure hotkey scheme that might reduce the cognitive overhead of name-based hotkeys. Of 22 refactorings that can be initiated via a linear menu in Eclipse, 11 can be initiated with my pie menus; my menus also support three additional refactorings not currently in Eclipse. In comparison, six refactorings can be initiated with hotkeys by default in Eclipse. Table 7.1 on page 118 shows how each refactoring tool in Eclipse can be initiated.

The restriction also means that certain refactorings do not appear on my pie menus, especially refactorings that seem to have no associated direction, such as RENAME. However, I do not view incompleteness as a problem for several reasons. First, programmers currently use several mechanisms for initiating different refactorings [47], and thus programmers may be willing to use some other initiation mechanism in addition to pie menus. Second, my pie menu design *could* provide for the initiation of all refactorings by allowing submenus, or more than 4 items per menu, although this would compromise speed or accuracy of selecting menu items. Third, I argue that completeness at the expense of simplicity is not always desirable when

it comes to user interfaces; indeed, the length of the menu containing all refactorings in Eclipse may have been the cause of my survey respondent's usability complaint mentioned in Section 7.2.

Pie menus for refactoring were designed to meet each of the guidelines outlined in Section 7.2. In short, using pie menus to initiate refactorings gives programmers the speed of hotkeys with the low entry barrier of linear menus. Pie menus that use my placement rules may facilitate the transition from beginner (a programmer who uses refactoring tools infrequently) to expert (a programmer who always uses the tool when refactoring) in a way that is not possible with name-based hotkeys, linear menus, or pie menus without my placement rules.

7.5 Evaluation

7.5.1 Previous Studies: Pie Menus *vs.* Linear Menus

Here I compare pie menus for refactoring to the state-of-the-practice refactoring tool initiation mechanism, which is the linear menus. I do not compare pie menus against hotkeys, both because refactoring hotkeys are user-mappable and thus can be as fast as pressing a single key, and because users of hotkeys can continue to use hotkeys with pie menus.

Two previous studies suggest that four-item marking menus are faster and less error-prone than linear context menus. Callahan and colleagues [6] described a study in which the speed improvement of pie menus over linear context menus was statistically significant. Furthermore, when Callahan and colleagues' measurements are compared with Kurtenbach and Buxton's measurements [37], marking menus are more than 3 times faster than pie menus. Callahan and colleagues also observed that pie menus are less error-prone than linear menus, but the statistical significance was marginal. These results strongly suggest that programmers should be able to use my

Refactoring	Linear Menus	Hotkeys	Pie Menus
Rename	Yes	Alt+Shift+R	
Move	Yes	Alt+Shift+V	
Change Method Signature	Yes	Alt+Shift+C	
Extract Method	Yes	Alt+Shift+M	Right
Extract Local Variable	Yes	Alt+Shift+L	Right
Extract Constant	Yes		
Inline	Yes	Alt+Shift+I	Left
Convert Anonymous Class to Nested	Yes		Right
Convert Member Type to Top Level	Yes		Right
Convert Local Variable to Field	Yes		Right
Extract Superclass	Yes		
Extract Interface	Yes		
Use Supertype Where Possible	Yes		
Push Down	Yes		Bottom
Pull Up	Yes		Top
Introduce Indirection	Yes		Right
Introduce Factory	Yes		Right
Introduce Parameter Object	Yes		
Introduce Parameter	Yes		
Encapsulate Field	Yes		Right
Generalize Declared Type	Yes		
Infer Generic Type Arguments	Yes		
Convert Nested to Anonymous	No		Left
Increase Visibility	No		Right
Decrease Visibility	No		Left

Table 7.1: How refactorings can be initiated using Eclipse 3.3 and my current implementation of pie menus, in the order in which each refactoring appears on the system menu (Figure 7.1 on page 112); for pie menus, the direction in which the menu item appears is shown in the third column. I implemented the last three refactorings specifically for pie menus.

pie menus[3] faster and with fewer errors than a linear menu of the same size.

7.5.2 Memorability Study: Pie Menus with and without Placement Rules

In Section 7.4, I noted that users of pie menus for refactoring may not become experts (that is, know *a priori* the gesture of the desired refactoring) because becoming an expert requires repeated use of a refactoring tool. I hypothesized that my placement rules help programmers to recall the direction of items on a pie menu. In this section I describe an experiment that tests this hypothesis by measuring memory recall. This is the first experiment to explore the effect of pie-menu item placement on recall.

7.5.2.1 Methodology

In this experiment, I asked programmers to memorize the direction (left, right, top, or bottom) of a refactoring on a pie menu. In the training phase, I gave programmers a paper packet containing 9 pages. Each page contained an initial piece of code, a refactored piece of code, and a pie menu with the associated refactoring highlighted on one of the four menu quadrants (Figure 7.3, top). I told programmers to try to associate the before-and-after code with the direction on the pie menu.

In the testing phase immediately following the training phase, I gave the programmers the same 9 pages, but in a different order and with the labels on the pie menus removed. I told programmers to try to recall where the refactoring appeared on the pie menu. During the training phase, I allowed programmers 30 seconds to read and understand the refactoring from the before-and-after code, and also to remember the direction of each refactoring. I gave subjects such a short period because I wanted them to spend most of the time reading and understanding the code transformation, and only a small amount of time remembering the direction of a refactoring

[3] Recall that my pie menus are actually marking menus, so for my implementation, Kurtenbach and Buxton's results are better predictors than those of Callahan and colleagues.

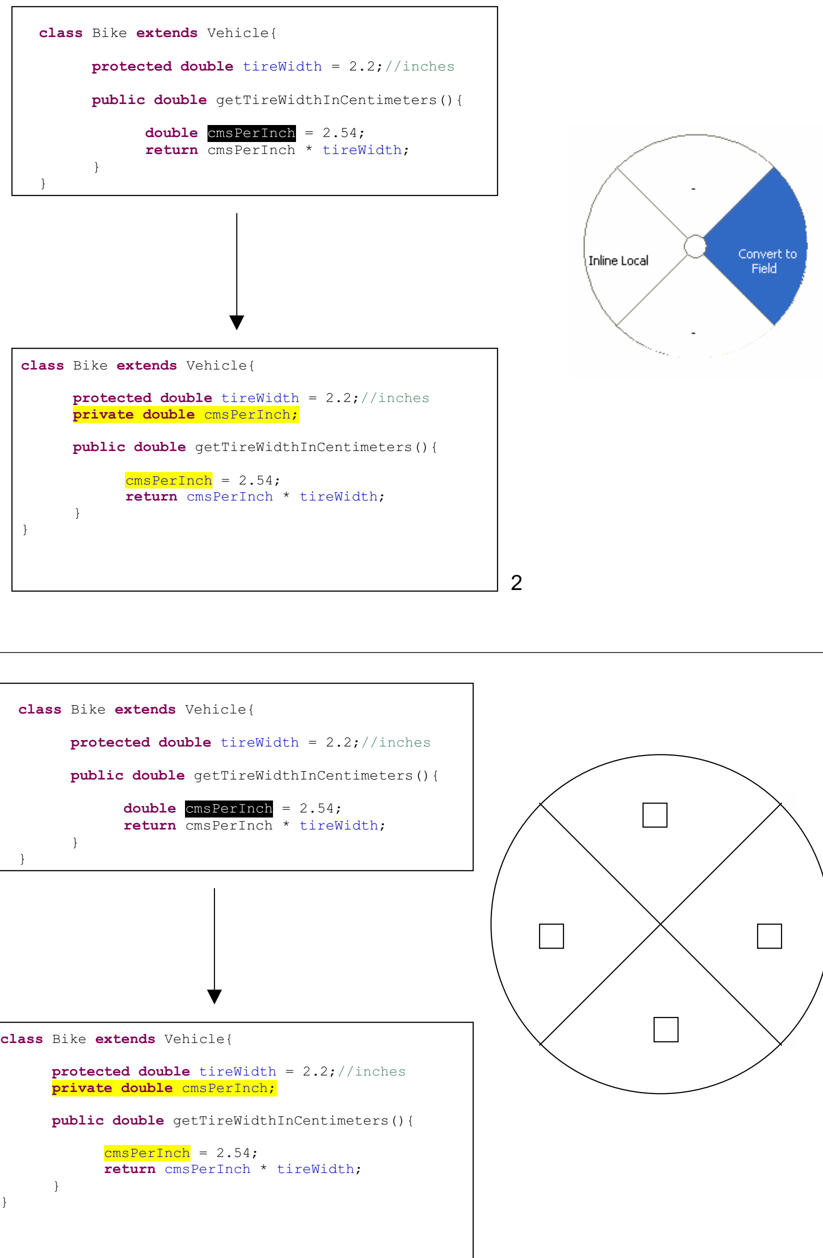


Figure 7.3: A sample training page (top) and a sample recall page (bottom). The refactorings (left, as program code before-and-after refactoring) are the same on both pages. Subjects were instructed to put a check mark in the appropriate direction on the recall page.

item. This was because few programmers in the wild are going to spend significant time memorizing the direction of items on a pie menu. Interviewing some dry-run experiment participants informally confirmed that little time was spent memorizing.

Using before-and-after code to identify the refactoring, rather than using only its name, had several advantages:

- Programmers who had no knowledge of refactoring terminology could still participate in the experiment.
- My choice of refactoring names would not confuse programmers who had experience with refactoring, but who used different terminology (see Section 8.2).
- If programmers think of a refactoring as a transformation of code and not as a name in a book or in a development environment, then the experiment more closely matches how programmers refactor in the wild.

During the testing phase, programmers were given a total of 5 minutes to recall the direction of all 9 refactorings. The bottom of Figure 7.3 shows an example of a recall page. Additionally, during that time programmers were asked to guess the direction of a refactoring that they had *not* seen during the training period. The refactoring to be guessed was EXTRACT LOCAL VARIABLE, which, according to my placement rules, should appear in the same direction as the CONVERT LOCAL TO FIELD, ENCAPSULATE FIELD, and INTRODUCE INDIRECTION (see Table 7.1 on page 118), three refactorings that the subjects had seen during training.

More information about this experiment, including the test materials and the resulting data set can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

7.5.2.2 Subjects

I recruited 18 programmers to participate in this experiment. In an attempt to sample a diverse population, these programmers were recruited from three sources. Seven were students from a graduate programming class on Java design patterns, six were computer-science research assistants, and five were programmers from industry. To expose the programming class to the concept of refactoring, I gave students a 20-minute presentation on refactoring two weeks prior to the experiment.

The only prerequisite for participating in the experiment was the ability to read Java code. Two subjects were removed from the programming class set because one did not meet this prerequisite and one did not follow directions during the experiment, leaving a total of 16 participants. With the exception of offering refreshments during the experiment, I did not compensate the participants for their time.

Within each set, each programmer was randomly assigned to one of two groups. In the experimental group, programmers were trained on pie menus that contained refactoring items placed according to my rules. In the control group, programmers were trained on pie menus that contained refactoring items placed in opposition to each of my rules[4].

7.5.2.3 Results

Overall, subjects in the control group could recall a median of 3 refactoring directions, while subjects in the experimental group could recall a median of 7.5 refactoring directions. The difference is statistically significant ($p = .011$, $df = 14$, $z = 2.553$ using a Wilcoxon rank-sum test). The results of the experiment are shown in Figure 7.4 on the next page.

Six out of the eight subjects in the experimental group correctly guessed the di-

[4] I placed them in opposition rather than randomly because most random placements happen to obey one or more of my rules

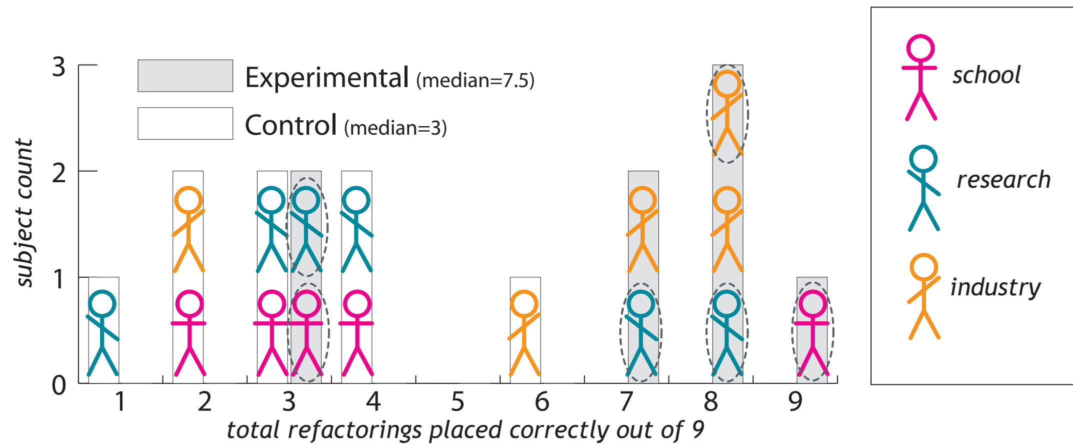


Figure 7.4: A histogram of the results of the pie menu experiment. Each subject is overlaid as one stick figure. Subjects from the experimental group who correctly guessed the refactoring that they did not see during training are denoted with a dashed oval.

rection of the refactoring that they had not seen during training. This suggests that this group of programmers were not simply recalling what they had learned, but had synthesized a mental model regarding where refactorings "should" appear.

7.5.2.4 Threats to Validity

There are three limitations of this experiment. First, subjects were asked to memorize the direction of 9 refactorings, which represent only about one-tenth of the refactorings cataloged by Fowler [22]. The effect of trying to recall the full catalog of refactorings is unclear. Second, I cannot easily explain the outliers in the data set (one overperforming control group subject and two underperforming experimental group subjects, Figure 7.4). It may be the case that some programmers can easily recall the direction of a refactoring on a pie menu, regardless of their placement, and that some programmers have difficulty recalling the direction, even when placed using my rules. Third, the experiment was conducted in such a way that it is difficult to discern which of the three placement rules (outlined in Section 7.4) was most helpful.

Tool	Generality	Keyboard or Mouse	Speed	Memorability
Hotkeys	Some	Keyboard	Fast	Low
Linear Menus	All	Both	Slow	*
Pie Menus	Some	Both	Fast	High

Table 7.2: A comparison of initiation mechanisms for refactorings tools.

7.5.2.5 Discussion

The results of the experiment confirm the hypothesis that my placement rules help programmers to recall the direction of refactorings. More importantly, the results support my theory that refactorings map intuitively to the directionality provided by pie menus. I believe that this will help programmers to initiate refactorings quickly, while building muscle memory, and without having to resort to rote memorization. Combined with the speed and error-resistance demonstrated in previous studies (Section 7.5.1), this improvement in recall suggests that pie menus for refactoring, and the guidelines that inspired their implementation, provide improved usability over conventional mechanisms for initiating refactoring tools. In addition to the results presented in this section, I describe a study regarding programmers' opinions of pie menus for refactoring in Section 8.5.2.

7.5.3 Summary: A Comparison

Table 7.2 compares the features and performance of pie menus for refactoring with existing refactoring tool user interfaces. By **Generality** I mean the number of refactorings the tool supports. For example, all refactorings can be initiated with linear menus, but typically (by default) only a subset can be initiated with hotkeys. The **Keyboard or Mouse** column refers to whether a tool can be used with the keyboard, the mouse, or both. The **Speed** and **Memorability** columns are summaries of the results presented in this chapter. The asterisk indicates that a column does not apply to that tool.

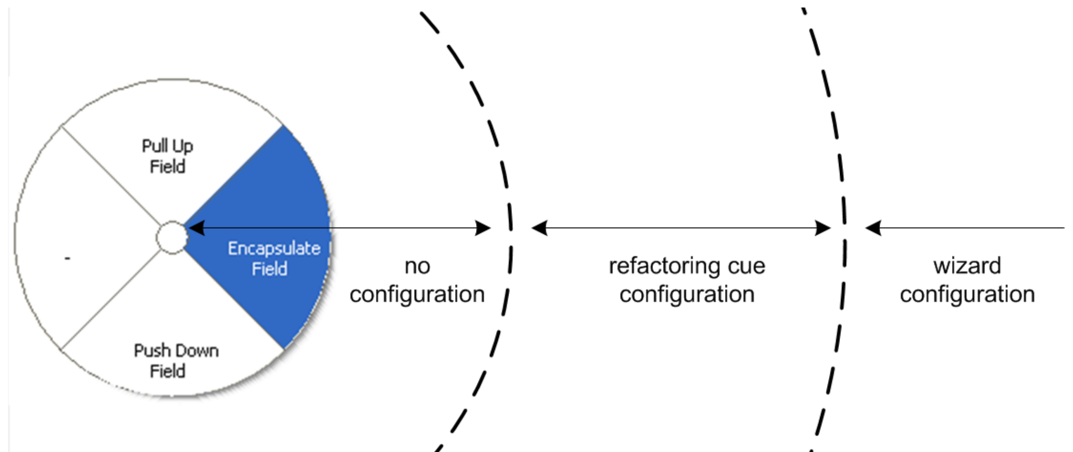


Figure 7.5: A pie menu for refactoring with distance-from-center indicating what kind of configuration to perform.

7.6 Future Work

I would like to investigate how to integrate pie menus for refactoring and refactoring cues. This could be accomplished by allowing pie menus to initiate a refactoring with no configuration (as in the current implementation), configuration through refactoring cues (allowing the programmer to select more code to refactor), or configuration through standard Eclipse wizards, depending on the mouse distance from the menu's center (Figure 7.5). Heuristically, the greater the distance from the marking menu's center, the more heavyweight the configuration. This approach makes pie menus for refactoring much more like control menus [64].

I would also like to address deficiencies exposed during my programmer interviews (Section 8.5.2). The most common dislike of programmers regarding pie menus was that they were "too ugly" and, ironically, obtrusive. In response, I plan on implementing a labels-only design, allowing a large proportion of the circular menu to be transparent or translucent.

7.7 Conclusions

In this chapter, I have presented guidelines for the *initiation* of refactoring tools (Section 2.5), and pie menus for refactoring, an initiation mechanism designed to meet those guidelines. Previous results have suggest that pie menus for refactoring will be faster than traditional linear menus of equivalent size. The study presented in this chapter suggests further that my design rules for placing refactorings on pie menus improves memorability, which I hope will aid programmers as they use refactoring tools with increasing frequency and build muscle memory.

Chapter 8

The Configuration Step: Tweaking how a Refactoring Tool Executes

In addition to telling the refactoring tool which refactoring to perform, a programmer may also supply configuration information regarding additional information that the tool should use to perform the refactoring. This is called the *error interpretation* step of the refactoring process (Section 2.5).

8.1 Contributions

The major contributions of this chapter are as follows:

- Guidelines for making refactoring tools that are easier to configure (Section 8.2).
- Refactoring cues, a tool previously described in (Section 6.6.3), which includes a user interface that allows the programmer to bypass tool configuration when it is unnecessary (Section 8.4).
- An analysis that shows that refactoring cues are at least as fast and error-resistant as traditional refactoring tools (Section 8.5.1).
- A survey that suggests that programmers are willing to try refactoring cues.

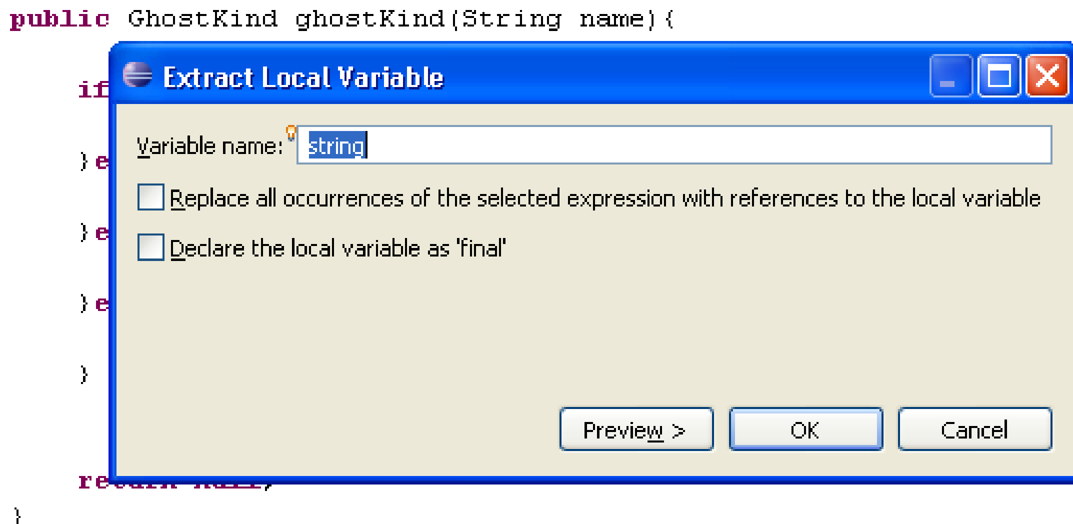


Figure 8.1: Configuration gets in the way: an Eclipse configuration wizard obscures program code.

8.2 Guidelines

Once a tool has been initiated, programmers sometimes configure it. In the example from Section 6.6.1, you want to choose a name for the new local variable. Most refactoring tools use a modal dialog box or multi-page "wizard" for configuration. The Eclipse EXTRACT LOCAL VARIABLE configuration interface is illustrated in Figure 8.1.

A modal dialog box compels programmers to configure the refactoring by not letting them do anything else until the configuration is finished. However, it appears that programmers rarely perform any configuration at all. In Section 3.4.2, I presented data that showed that one group of 4 programmers changed default configuration parameters less than 10% of the time, while another group of 6 programmers estimated that they changed default parameters 25% of the time. Thus, users should be able to bypass configuration entirely the remainder of the time, revealing this guideline:

◇ **Bypassability.** Refactoring tool configuration should not force the program-

mer to view or enter unnecessary configuration information.

This guideline is a more specific form of Shneiderman's proposed guideline, "Minimal input actions by user" [71, p. 72].

A modal configuration dialog also forces the refactoring tool to become the programmer's primary focus. This is undesirable because refactoring is a subsidiary task in floss refactoring, as I explained in Section 7.2. Thus:

◇ **Task-centricity.** Refactoring tool configuration should not interrupt the programmer's primary task.

Disrupting a programmer's focus may cause visual disorientation, a problematic issue in modern development environments [11].

Furthermore, the configuration dialog may obstruct the view of the source code, which can hinder the programmer's ability to configure the refactoring. For example, choosing a meaningful variable name may require using programming tools to examine the rest of the program. So I recommend that:

◇ **Accessibility.** Refactoring tool configuration should not obstruct a programmer's access to other tools, including the source code editor itself.

This guideline is similar to Raskin's comment that "it is always safe to avoid modes" [65, p. 50]. The reason, according to Raskin, is that you sometimes make errors when you think that a system is in one state, but it is actually in another because it is modal. In the case of refactoring, the programmer would expect to have access to her normal programming tools, but this is not the case when configuration is modal.

```
characterCount = 4 + 1;
```

```
int v = 4;
characterCount = v + 1;
```

```
int ghostC = 4;
characterCount = ghostC + 1;
```

Figure 8.2: The user begins refactoring by selecting the 4 in X-develop, as usual (top). After initiating EXTRACT LOCAL VARIABLE (middle), the user types "ghostC" (bottom), using a linked in-line RENAME refactoring tool.

8.3 Related Work: Alternative Configuration Techniques

To avoid the distraction of configuring a refactoring, some development environments try to avoid configuration altogether. In the X-develop environment [75], the programmer supplies the name of a newly-created program element *after* the refactoring has been performed, using a linked in-line RENAME. For example, when an EXTRACT LOCAL VARIABLE refactoring is performed, the tool extracts an expression into a new variable, which is given a default name (Figure 8.2, middle). The programmer may change the name by typing (Figure 8.2, middle and bottom), which causes all references to that variable to be updated in real time by the refactoring tool. I feel that this is a natural way to bypass configuration, but it is not applicable to all refactorings. Eclipse and IntelliJ IDEA [32] also use this in-line rename technique for some refactorings.

8.4 Tool Description

The refactoring cues tool, described in Section 6.6.3 and pictured in Figure 6.4 on page 108, handles configuration by providing configuration options in a non-modal palette to the right of the program editor. The palette is designed to stay out of the programmers' way because it is non-modal, and appears to the side of the screen

rather than in the center. Moreover, because the palette takes up valuable screen real estate, it does not have to be displayed by default. Instead, programmers can designate it as an Eclipse "Fast View," bringing it up only when they wish to perform configuration. The individual user interface widgets in the palette are the same as those widgets that would appear on a standard modal refactoring wizard, with one important exception: rather than typing names of any newly created program element into the palette, new names are typed directly into the editor using the standard Eclipse in-line rename.

8.5 Evaluations

I present two studies that suggest that refactoring cues are an improvement over existing user interfaces to refactoring tools. In the first study, I compare refactoring cues to conventional refactoring tools analytically with respect to speed and error rate. In the second study, I describe an interview that I conducted to determine professional programmers' opinions towards refactoring cues, as well as the previously described pie menus (Section 7.4).

8.5.1 Analytical Study: Refactoring Cues vs. Traditional Tools

In this section I will argue that refactoring cues are faster and less error prone than conventional refactoring tools. By a conventional refactoring tool I mean one that requires that the user select code, initiate the refactoring, and then configure it with a wizard or dialog, as described in Section 8.2. I will use a step-by-step analysis to compare the speed and likelihood of error using refactoring cues and conventional refactoring tools.

Method for goal: Refactor with Traditional Tool

Step	
Step 1. Select Code **Operator**: Make selection with mouse	S_{CT}
Step 2. Initiate Tool	I_{CT}
Step 3. Type element name	N_{CT}
Step 4. Enter other information in GUI	G_{CT}
Step 5. Execute	E_{CT}
Step 6. **If** More code to refactor **Then** go to **Step 1**	M_{CT}

Return with goal accomplished

Method for goal: Refactor with Refactoring Cues

Step	
Step 1. Initiate Tool	I_{RC}
Step 2. Select Code **Operator**: Make selection with mouse	S_{RC}
Step 3. Enter other information in GUI	G_{RC}
Step 4. **If** More code to refactor **Then** go to **Step 2**	M_{RC}
Step 5. Execute	E_{RC}
Step 6. Type element name	N_{RC}

Return with goal accomplished

Figure 8.3: NGOMSL methods for conventional refactoring tools (top) and refactoring cues (bottom).

8.5.1.1 Analysis by Stepwise Comparison

Here I argue in a stepwise manner that refactoring cues are at least as fast and error-resistant as conventional refactoring tools by comparing their NGOMSL methods [33]. An NGOMSL method is a sequential list of user interface actions for achieving a goal, used for measuring a user interface's efficiency. While traditional NGOMSL analysis does not allow for errors made by the user, this comparison of NGOMSL methods does consider such errors made by programmers when using refactoring tools.

In Figure 8.3 on the previous page, I outline the major steps necessary to use each refactoring tool. To keep the methods general, I omit details for several steps, such as how one enters configuration information in each tool's GUI. To the right of each step, a capital letter and a subscript labels which tool the step belongs to. For example, E_{CT} refers to the Execution step using a Conventional Tool. The NGOMSL method for conventional tools refers to the selection, initiation, and configuration steps of the model shown in Figure 2.6 on page 14.

I chose NGOMSL as a way to describe how programmers use refactoring tools because of its standardized structure, convenient and flexible notation, and ability to express high-level goals rather than individual keystrokes. Usually, NGOMSL is used to estimate *how long* it takes for a human to learn and use a user interface by assigning times to user interface operations. Here, instead, I use NGOMSL to *compare* two different user interface operations in terms of both speed and error-resistance.

I will use colors for clarity and introduce some notation for brevity. I have color-coded the steps in each method so that the correspondence between the steps in the two methods is more obvious. I will use $=$ to mean "is equally fast and error-resistant as," $\succ$ to mean "is faster or more error-resistant than," and $\succeq$ to mean "is at least as fast and error-resistant as." Thus, you can think of $=$ to mean "as good as," $\succ$ to mean "is better than," and $\succeq$ to mean "is at least as good as." Step-by-step, I demonstrate that refactoring using refactoring cues is at least as fast and error-resistant as with conventional tools.

$\boldsymbol{S_{RC} \succeq S_{CT}}$**.** Any valid editor selection in conventional refactoring tools is a valid cue selection with refactoring cues, so in these cases, $S_{RC} = S_{CT}$. Furthermore, refactoring cues are more error-resistant in certain cases, such as when a programmer over-selects by a few parentheses. In other cases, refactoring cues allow program elements that are leaves (that is, do not contain other program

elements of the same kind, such as constants) to be selected with a single drag gesture. So in a some cases $S_{RC} \succ S_{CT}$, otherwise $S_{RC} = S_{CT}$.

$\boldsymbol{I_{RC} \succeq I_{CT}}$**.** Whatever initiation mechanism is used for conventional refactoring tools can also be used for refactoring cues, whether it be hotkeys, linear menus, or pie menus. In these cases, $I_{RC} = I_{CT}$. I have also made available a third initiation mechanism, the palette, which has various advantages and disadvantages when compared to hotkeys and menus (which will not be compared here). If the palette is advantageous and is used then $I_{RC} \succ I_{CT}$, otherwise $I_{RC} = I_{CT}$ when it is not used.

$\boldsymbol{N_{RC} = N_{CT}}$**.** Names are entered into a text box using conventional tools and directly into the editor with a linked, in-line rename using refactoring cues. However, from the user's perspective, using either tool for entering a new name is a matter of typing, so assuming that the same editing operations are available in both the text box and editor, $N_{RC} = N_{CT}$.

$\boldsymbol{G_{RC} \succeq G_{CT}}$**.** Configuration with the GUI in refactoring cues is no different from configuration with conventional tools, except that the user need not restart the process when she wishes to navigate the underlying source code. In cases when the programmer does not wish to perform any configuration, she can retain focus on the editor with refactoring cues but must shift focus to a dialog with conventional tools. In these cases, $G_{RC} \succ G_{CT}$, otherwise $G_{RC} = G_{CT}$.

$\boldsymbol{E_{RC} = E_{CT}}$**.** The underlying refactoring engine is the same for both refactoring cues and conventional tools, and both tools are initiated with a keystroke or button press, so the execution step is identical. Therefore, $E_{RC} = E_{CT}$.

$\boldsymbol{M_{RC} \succeq M_{CT}}$**.** In Section 6.4, I showed that refactoring multiple elements was a significant use case. A programmer may choose to refactor multiple program

element using either tool. With a conventional tool, she must repeat steps 1 through 5, but with refactoring cues, she need repeat only steps 2 and 3. So when multiple program elements need to be refactored, $M_{RC} \succ M_{CT}$, otherwise (when the programmer wishes to refactor only one element) $M_{RC} = M_{CT}$.

8.5.1.2 Threats to Validity

NGOMSL, and the GOMS family of analysis techniques in general, is limited in that it considers only procedural aspects of usability, not perceptual issues or users' conceptual knowledge of the system [33]. This analysis also does not consider mental operations, such as when the programmer is deciding which refactoring she wants to perform.

Additionally, a limitation of my analysis is that it does not take into account every use case. The most significant omitted cases are as follows:

- The programmer does not have to perform every step; she can bypass entering information into the GUI or typing an element name. These steps can be completely skipped using refactoring cues, but the programmer must at least dismiss a dialog when using conventional tools. Refactoring cues are slightly faster in this situation.

- Some steps can be reordered, such as N_{CT} and G_{CT}. However, this should not significantly change the speed and error-resistance of the refactoring cues.

- While the programmer can use the keyboard or mouse for selection, my analysis assumes that the selection step is performed with the mouse. However, using the keyboard for selection when using refactoring cues requires one key press more than when using a conventional tool. I am confident that the cost of this key press is overwhelmed by the speed improvements achieved elsewhere with refactoring cues, but I cannot show this with the step-by-step comparison

given here. Therefore, for this analysis, I excluded the use of the keyboard during the selection step.

8.5.1.3 Discussion

The analysis suggests that refactoring cues are at least as fast and error-resistant as conventional refactoring tools because using both interfaces involves similar steps, just in a different order. When the order of the steps is factored out, both tools can be used in similar ways, and under certain significant use cases, refactoring cues are more error-resistant or faster, according to my analysis. These two improvements suggest that refactoring cues, and the guidelines that inspired their implementation, provide improved usability over the traditional refactoring tool user interface.

8.5.2 Opinion Study: Pie Menus, Refactoring Cues, Hotkeys, and Linear Menus

So far in this section I have talked about how quickly and accurately refactoring cues and pie menus for refactoring can be used, but I have neglected programmer satisfaction. If a programmer does not *want* to use a tool, then it does not matter how fast or accurate it is.

8.5.2.1 Methodology

In an attempt to assess how programmers feel about using my pie menus and refactoring cues, I conducted interviews at the 2007 O'Reilly Open Source Convention in Portland, Oregon. Each interview lasted about 20 minutes. Subjects answered a brief oral questionnaire regarding their programming and refactoring experience. Subjects were then shown four videos, each lasting about 90 seconds, demonstrating a programmer's development environment while performing a refactoring:

- In the first video, the programmer is shown initiating a number of refactorings in series via a linear popup menu.
- In contrast, the second video shows the programmer performing the same refactorings with pie menus (albeit slowly, to allow the viewer to read the menus).
- In the third video, a programmer is shown using a conventional refactoring tool for several refactorings.
- In contrast, the fourth video shows the programmer performing the same refactorings with refactoring cues.

Each of these videos can be viewed at http://multiview.cs.pdx.edu/refactoring/activation. After the second and fourth video, the interviewer compared pie menus for refactoring and refactoring cues with the standard Eclipse tools by verbally enumerating the advantages and disadvantages of each tool. This reiterated the qualities of the tools shown in the videos and highlight some disadvantages that were not obvious from the videos. For instance, programmers were told that I placed items on pie menus in a way that I believed was memorable, but also that I omitted some popular refactorings for which my placement rules provide no guidance, such as RENAME. A full list of the advantages and disadvantages presented to each subject is shown in Table 8.1 on the next page. I believe that telling the programmers about the advantages and disadvantages allowed them to make a more informed estimate of how the tools might effect their refactoring behavior.

I chose to show the subjects videos instead of allowing them to use the tools for two reasons. First, the videos ensured that each subject saw the tool working over the same code in the same manner. Second, because my tool was a prototype when I conducted the evaluation, the videos ensured that the subjects did not encounter bugs in my implementation that might have interfered with the experiment.

Pie Menus for Refactoring
+ Can be activated using hotkeys or the mouse
+ Faster to activate than context menus
+ Easier to remember than hotkeys
+ Placement should be especially memorable
– Some refactorings do not make sense to put on this menu
– Adding new refactorings will disrupt memory

Refactoring Cues
+ Can be activated using hotkeys or mouse
+ Many items can be refactored at once
+ Selection errors are cut down
+ Makes "what is refactorable" explicit
+ Configuration becomes optional
– All refactoring activators must be shown

Table 8.1: Advantages and disadvantages of pie menus and refactoring cues enumerated by the interviewer, labeled with + for advantage and – for disadvantage.

More information about this experiment, including the survey and results, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

8.5.2.2 Subjects

I approached conference attendees to determine whether they were appropriate interviewees. I attempted to approach attendees indiscriminately, and interviewees were chosen if they had recently programmed Java, knew what refactoring was, and did not know the interviewer or his research. Of all the attendees that the interviewer approached, about one quarter met these criteria. While the interviewees cannot be considered a random sample of programmers, I believe that the interviewees were unbiased and fairly representative because I sampled from a national conference whose topic was orthogonal to refactoring. In total, I fully interviewed 15 attendees, while one declined to complete the interview due to time constraints. Subjects were not compensated.

The 15 interviewees reported a mean of 16 years of programming experience and spent an average of about 27 hours per week programming; 13 of the 15 interviewees were Eclipse users. I characterize the group as experienced programmers because 13 of 15 subjects had been programming for at least 10 years and because 13 of 15 subjects spent at least 20 hours per week programming at the time of the interview.

8.5.2.3 Results

When asked whether having the option of using pie menus, in addition to other initiation mechanisms, would increase their usage of refactoring tools, most interviewees said that they would use the tools to refactor the same amount as they use them currently. However, 6 of 15 interviewees estimated that the presence of pie menus would encourage them to refactor more. When I asked the same question about refactoring cues, 8 of 15 reported that they would use refactoring tools more often if refactoring cues were available. Several subjects expressed difficulty in answering this question without using the tools, but the positive responses indicate that both tools may encourage programmers to refactor more often using tools and less often by hand.

I asked one additional question at the end of the last 11 interviews. In that question, interviewees were asked to estimate how often they might use refactoring tools with hotkeys, linear menus, pie menus, or refactoring cues, if all were available in the same development environment. On average, pie menus were rated higher than linear menus, a difference that was statistically significant ($p = .028$, $df = 10$, $z = 2.2$, using a Wilcoxon matched-pairs signed-rank test). This result is notable, because subjects initially reported that linear menus were the frequent method of initiating refactoring tools. No other differences were statistically significant; refactoring cues, pie menus, and hotkeys were all estimated to be used about the same amount.

8.5.2.4 Threats to Validity

There are two main limitations of this survey. First, the programmers saw videos of how the tools worked, but were not able to try the tools themselves; programmers' actual usage of the tool will vary from their estimated usage. Second, the refactorings in the videos were created to exercise the features of each new refactoring tool. Therefore, the refactorings shown to the programmers may not be representative of typical refactorings in the wild.

8.5.2.5 Discussion

The results of the experiment suggest that programmers are willing to try both pie menus and refactoring cues as part of their existing refactoring toolbox.

8.5.3 Summary: A Comparison

Table 8.2 on the following page compares the features and performance of refactoring cues with existing refactoring tool user interfaces. The **Steps Addressed** column names the steps in the refactoring model that the tool helps the programmer to accomplish (Section 2.5). By **Generality**, I mean the number of refactorings that the tool supports. The **Selection Error-Resistance** column refers to how well the tool helps programmers to avoid and recover from mis-selected program elements, again, in a simplified manner. The **Stay in Editor** column refers to whether a tool allows programmers to stay focused in the editor while refactoring. An asterisk indicates that a column does not apply to a tool.

8.6 Future Work

Conducting a case-study evaluation of my refactoring tools would be valuable in the future. While it has been helpful to conduct interviews and laboratory experiments,

Tool	Steps Addressed	Generality	Selection Error-Resistance	Stay in Editor
Editor Selection	Selection	All	Low	Yes
Selection Assist	Selection	One	Medium	Yes
Box View	Selection	One	High	No
Wizard/Dialog	Configuration	All	*	No
Linked In-line	Configuration	Some	*	Yes
Refactoring Cues	Selection & Configuration	All	High	Yes

Table 8.2: A comparison of selection and configuration mechanisms for refactoring tools.

I believe that observing long-term tool usage is a better predictor of how programmers will behave in the wild. Such a study should measure what kinds of errors are made; how long it takes to perform individual refactorings; the number and variety of refactorings that are performed over the long-term; and whether users are satisfied after using my refactoring tools for several weeks.

Cues may not be readable after many levels of nesting because many nested cues are so dark that they completely obscure program text, and thus alternative cue coloring strategies may be desirable. One such strategy is cushion tree maps [39], which give the illusion of cue depth without the need for varied saturation. Cue colors should be changed as well, as color-blind programmers report difficulty in distinguishing pink and green cues.

I designed refactoring cues to make it faster to refactor several program elements at once, and two future improvements may make this use case even faster. First, if multiple refactorings are executed at once, it may be possible to amortize the cost of precondition checking and code transformation in the refactoring engine itself. Second, refactoring cues could be modified to refactor-on-select, eliminating a key press in the refactoring process.

8.7 Conclusions

In this chapter, I have addressed the usability of tools that help programmers during the *configuration* step if the refactoring process (Section 2.5). Refactoring tools that provide a user interface for configuration typically block the programmer from accessing other tools and shifts focus from the code to the refactoring tool itself. Because refactoring tools are rarely configured, it behooves toolsmiths to build refactoring tools that do not force programmers to configure them. While compelling programmers into configuration has long been the required workflow for using refactoring tools, refactoring cues and the guidelines that I have presented in this chapter illustrate how non-obtrusive configuration can be achieved.

Chapter 9

The Error Interpretation Step: Recovering When Refactorings Go Wrong [1]

When a refactoring tool is unable to perform a programmer-requested refactoring — a violation of a refactoring precondition — it generally presents a message in a dialog box in an attempt to explain what precondition was violated and why. As my exploratory study demonstrated in Section 4.5, programmers can have difficulty understanding such error messages. In this chapter I demonstrate that changing the user-interface to refactoring tools can improve programmers' ability to understand precondition violations. This is called the *error interpretation* step of the refactoring process (Section 2.5).

9.1 Contributions

The major contributions of this chapter are as follows:

- A new in-code visualization to help programmers understand violations of EXTRACT METHOD preconditions (Section 9.2).
- An experiment that shows that this visualization can improve the diagnosis of precondition violations for EXTRACT METHOD by between 79 percent and 91

[1] Parts of this chapter appeared in the *Proceedings of the 2008 International Conference on Software Engineering* [49].

```
boolean areWheelsTrue(){

  Wheel front = bike.getFrontWheel();
  Wheel rear = bike.getRearWheel();

  boolean trued =isWheelTrue(front);
  trued = trued && isWheelTrue(rear);

  return trued;
}
```

Figure 9.1: Refactoring Annotations overlaid on program code. The programmer has selected two lines of code (between the dotted lines) to extract. Here, Refactoring Annotations show how the variable will be used: `front` and `rear` will be parameters, as indicated by the arrows into the code to be extracted, and `trued` will be returned, as indicated by the arrow out of the code to be extracted.

percent, as well as speeding up diagnoses by 72 percent (Section 9.3).

- Guidelines for making better representations of precondition violations for refactoring tools (Section 9.4).
- A taxonomy of refactoring preconditions, which I derived from refactoring tools for 4 different languages, and an application of the guidelines to the taxonomy (Section 9.6.1).
- An evaluation that suggests that refactoring tool usability is improved when violations are presented in accordance with the guidelines (Section 9.6.2).

9.2 Tool Description

In this section, I describe a tool that I built for the Eclipse environment [18] that addresses the problems demonstrated in the formative study described in Chapter 4. You can download the tool, called Refactoring Annotations, and view a short screencast here: http://www.multiview.cs.pdx.edu/refactoring/refactoring_annotations.

In Section 4.5, I described the EXTRACT METHOD refactoring, in which a set of contiguous statements are moved from an existing block of code into a new method.

In that section I also presented several preconditions that must be true before a refactoring tool can execute EXTRACT METHOD (Table 4.1 on page 51). As the results of the formative study described in Section 4.5 suggest, programmers have difficulty understanding the textual error messages presented by refactoring tools, messages that attempt to explain why these preconditions are violated. Figure 4.1 on page 51 shows an example. As an alternative to error messages, I designed Refactoring Annotations to display violated preconditions for the EXTRACT METHOD refactoring. More broadly, Refactoring Annotations can be thought of as a kind of graphical error message.

Refactoring Annotations overlay program text to express control- and data-flow information about a specific extraction. Each variable is assigned a distinct color, and each occurrence of the varible is highlighted, as shown in Figure 9.1 on the previous page. Across the top of the selection, an arrow points to the first use of a variable that will have to be passed as an argument into the extracted method. Across the bottom, an arrow points from the last assignment of a variable that will have to be returned. L-values have black boxes around them, while r-values do not. An arrow to the left of the selection simply indicates that control flows from beginning to end.

These annotations are intended to be most useful when preconditions are violated, as shown in Figure 9.2 on the following page. When the selection contains assignments to more than one variable, multiple arrows are drawn across the bottom showing multiple return values (Figure 9.2 on the next page, top). When a selection contains a conditional return, an arrow is drawn from the return statement to the left, crossing the beginning-to-end arrow (Figure 9.2 on the following page, middle). When the selection contains a branch (`break` or `continue`) statement, a line is drawn from the branch statement to its corresponding target (Figure 9.2 on the next page, bottom). In each case, Xs are displayed over the arrows, indicating the location of the violated precondition.

```
void goOnVacation(){

  Bike roadBike = getRoadBike();
  Bike mountainBike = getMtnBike();


  loadOnCar(roadBike,mountainBike);
}

boolean curbHop(int curbHeight){
  int hopHeight = liftFrontWheel();

  if(hopHeight<curbHeight){
    endo();
    return FAILURE;
  }

  liftRearWheel();
  return SUCCESS;
}

boolean goForRide(){

  while(!tired()){
    rotatePedals(10);
    if(this.hasCrashed())
      break;
  }

  return SUCCESS;
}
```

Figure 9.2: Refactoring Annotations display an instance of a violation of refactoring precondition 1 (`goOnVacation`), precondition 2 (`curbHop`), and precondition 3 (`goForRide`), described in Table 4.1 on page 51.

When code violates a precondition, Refactoring Annotations are intended to give the programmer an idea of how to correct the violation. Often the programmer can enlarge or reduce the selection to allow the extraction of a method. Other solutions include changing program logic to eliminate `break` and `continue` statements; this is another kind of refactoring.

Refactoring Annotations are intended to scale well as the amount of code to be extracted increases. For code blocks of tens or hundreds of lines, only a few values are typically passed in or returned, and only the variables holding those values are colored. In the case when a piece of code uses or assigns to many variables, the

annotations become visually complex. However, I reason that this is desirable: the more values that are passed in or returned, the less cohesive the extracted method. Thus, I feel that code with visually complex Refactoring Annotations should probably not have EXTRACT METHOD performed on it. As one developer has commented, Refactoring Annotations visualize a useful complexity metric.

Refactoring Annotations are intended to assist the programmer in resolving precondition violations in two ways. First, because Refactoring Annotations can indicate multiple precondition violations simultaneously, the annotations give the programmer an idea of the severity of the problem. Correcting a conditional return alone will be easier than correcting a conditional return, and a branch, and multiple assignments. Likewise, correcting two assignments is likely easier than correcting six assignments. Second, Refactoring Annotations give specific, spatial cues that can help programmers to diagnose the violated preconditions.

Refactoring Annotations are similar to a variety of prior visualizations. My control flow annotations are visually similar to Control Structure Diagrams [27]. However, unlike Control Structure Diagrams, Refactoring Annotations depend on the programmer's selection, and visualize only the information that is relevant to the refactoring task. Variable highlighting is like the highlighting tool in Eclipse, where the programmer can select an occurrence of a variable, and every other occurrence is highlighted. Unlike Eclipse's variable highlighter, Refactoring Annotations distinguish between variables using different colors; moreover, the variables relevant to the refactoring are highlighted automatically. In Refactoring Annotations, the arrows drawn on parameters and return values are similar to the arrows in the DrScheme environment [16], which draws arrows between a variable declaration and each variable reference. Unlike the arrows in DrScheme, Refactoring Annotations automatically draw a single arrow for each parameter and for each return value. Finally, Refactoring Annotations' data-flow arrows are like the code annotations drawn in a program

slicing tool built by Ernst [14], where arrows and colors display the input data dependencies for a code fragment. While Ernst's tool uses more sophisticated program analysis than the current version of Refactoring Annotations, it does not include a representation of either variable output or control flow.

9.3 Evaluation

In the experiment described in this section, programmers used both the standard error message dialogs in Eclipse and Refactoring Annotations to identify problems in a selection that violated preconditions of the EXTRACT METHOD refactoring. I evaluated subjects' responses for speed and correctness.

9.3.1 Subjects

I conducted this experiment just after the selection experiment described in Section 6.3 and with the same 16 students from the object-oriented programming class.

9.3.2 Methodology

I randomly assigned subjects to one of two blocks; a different random code presentation order was used for each block. When subjects began this experiment, I showed them how the EXTRACT METHOD refactoring works using the standard Eclipse refactoring tool. I then demonstrated and explained each precondition violation error message shown in a dialog box by this tool; this took about 5 minutes. I then told subjects that their task was to identify each and every violated precondition in a given code selection, assisted by the tool's diagnostic message. I then allowed subjects to practice using the tool until they were satisfied that they could complete the task; this usually took less than 5 minutes. The subjects were then told to perform the task on 4 different EXTRACT METHOD candidates from different classes.

	Missed Violation	Irrelevant Code	Mean Identification Time
Error Message	11	28	165 seconds
Refactoring Annotations	1	6	46 seconds

Table 9.1: The number and type of mistakes when finding problems during the EXTRACT METHOD refactoring over all subjects, for each tool, and the mean time to correctly identify all violated preconditions. Subjects diagnosed errors in a total of 64 refactorings with each tool. Smaller numbers indicate better performance.

The experiment was then repeated using Refactoring Annotations on a different code base.

9.3.3 Results

Table 9.1 counts two kinds of mistakes made by subjects. "Missed Violation" means that a subject failed to recognize one or more preconditions that were being violated. "Irrelevant Code" means that a subject identified some piece of code that was irrelevant to the violated precondition, such as identifying a `break` statement when the problem was a conditional `return`.

Table 9.1 indicates that programmers made fewer mistakes with Refactoring Annotations than with the error messages. Using Refactoring Annotations, subjects were much less likely to miss a violation and much less likely to misidentify the precondition violations. The overall difference in the number of programmer mistakes per refactoring was statistically significant ($p = .003$, $df = 15$, $z = 2.95$, using a Wilcoxon matched-pairs signed-ranks test).

Table 9.1 also shows the mean time to find all precondition violations correctly, across all subjects. On average, subjects recognized precondition violations more than three times faster using Refactoring Annotations than using the error messages. The recognition time difference was statistically significant ($p < .001$ using a t-test with a logarithmic transform to remedy long recognition time outliers).

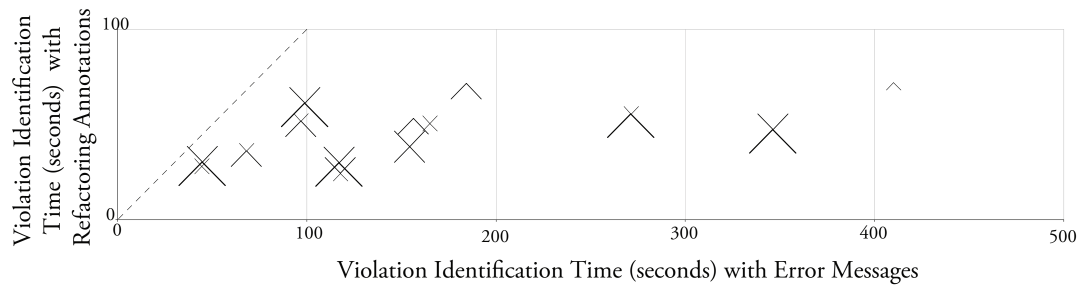


Figure 9.3: For each subject, mean time to identify precondition violations correctly using error messages versus Refactoring Annotations. Each subject is represented as an X, where the distance between the bottom legs represents the number of imperfect identifications using the error messages and the distance between the top arms represents the number of imperfect identifications using Refactoring Annotations.

Figure 9.3 shows the mean time to identify all precondition violations correctly for each tool and each user. Note that I omitted two subjects from the plot, because they did not correctly identify precondition violations for any code using the error messages. The dashed line represents equal mean speed using either tool. Since all subjects are below the dashed line, all subjects are faster with Refactoring Annotations. Most users were also more accurate using Refactoring Annotations.

Overall, Refactoring Annotations helped the subjects to identify every precondition violation in 45 out of 64 cases. In only 26 out of 64 cases, the error messages allowed the subjects to identify every precondition violation. Subjects were faster and more accurate using Refactoring Annotations than using error messages.

I administered a post-test questionnaire that allowed the subjects to express their preferences for the two tools they tried. Significance levels are reported using a Wilcoxon matched-pairs signed-ranks test.

Subjects were unanimously positive on the helpfulness of Refactoring Annotations and all subjects said they were likely to use them again. In comparison, the reviews of the error messages were mixed. Differences between the tools in helpfulness ($p = .003$, $df = 15$, $z = 3.02$) and likeliness to use ($p = .002$, $df = 15$, $z = 3.11$)

were both statistically significant. Concerning error messages, subjects reported that they "still have to find out what the problem is" and are "confused about the error message[s]." In reference to the error messages produced by the Eclipse tool, one subject quipped, "who reads alert boxes?"

Overall, the subjects' responses showed that they found the Refactoring Annotations superior to error messages for the tasks given to them. More importantly, the responses also showed that the subjects felt that Refactoring Annotations would be helpful outside of the context of the study. Materials from this experiment, including raw results, questionnaires, and my experimenter's notebook, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

9.3.4 Threats to Validity

Although the quantitative results discussed in this section are encouraging, one major limitation must be considered when interpreting these results in addition to the limitations discussed in Section 6.3.4. Every subject first used the Eclipse error messages and then used Refactoring Annotations. I originally reasoned that the fixed order was necessary to educate programmers about how EXTRACT METHOD is performed because my tool did not transform the code itself. Unfortunately, the fixed order may have biased the results to favor Refactoring Annotations due to a learning effect. In hindsight, I should have made an effort to vary the tool usage order. However, the magnitude of the differences in errors and speed, coupled with the strong subject preference, suggests that Refactoring Annotations are preferable to refactoring error message dialog boxes.

9.3.5 Discussion

The results of the experiment suggest that Refactoring Annotations are preferable to an error-message-based approach for showing precondition violations during the

EXTRACT METHOD refactoring. Furthermore, the results indicate that Refactoring Annotations communicate the precondition violations effectively. When a programmer has a better understanding of refactoring problems, I believe the programmer is likely to be able to correct the problems and successfully perform the refactoring.

9.4 Guidelines

By studying how programmers use both existing refactoring tools and the new tools that I have described in this chapter, I have induced a number of usability guidelines for refactoring tools. In this section, I link my experiment and the design of Refactoring Annotations to each guideline.

During the experiment, error messages required programmers to invest significant time to decipher the message. Refactoring Annotations reduced that time. Thus:

- ◇ **Expressiveness.** Representations of refactoring errors should help the programmer to comprehend the problem quickly by clearly expressing the details: the programmer should not have to spend significant time understanding the cause of an error.

By coloring the location of precondition violations in the editor, programmers could quickly and accurately locate problems using Refactoring Annotations during the experiment. With standard error messages, programmers were forced to find the violation locations manually. A tool should tell the programmer what it just discovered, rather than requiring the programmer "to basically compile the whole snippet in my head," as one Eclipse bug reporter complained regarding an EXTRACT METHOD error message [1]. Thus:

- ◇ **Locatability.** Representations of refactoring errors should indicate the location(s) of the problem.

Refactoring Annotations show all errors at once: during the experiment, programmers could quickly find all violated preconditions. In contrast, programmers that used error messages in dialog boxes had to fix one violation to find the next. Thus, to help programmers assess the severity of the problem:

◇ **Completeness.** Representations of refactoring errors should show all problems at once.

Counting the number of Xs using Refactoring Annotations gives programmers a visual estimate of the degree of the problem, whereas the error messages do not. For instance, the error messages did not indicate how many values would need to be returned from an extracted method, just that the number was greater than one. The programmer should be able to tell whether a violation means that the code can be refactored after a few minor changes, or whether the refactoring is nearly hopeless. Thus:

◇ **Estimability.** Representations of refactoring errors should help the programmer estimate the amount of work required to fix violated preconditions.

Violations are often not caused at a single character position, but instead arise from a number of related pieces of source code. Refactoring Annotations related variable declarations and references using the same color, allowing the programmer to analyze the problem one variable at a time. More generally, relations can be represented using arrows and colors, for example. Thus:

◇ **Relationality.** Representations of refactoring errors should display error information relationally, when appropriate.

Looking for Xs in the Refactoring Annotations allowed programmers to quickly distinguish errors from other types of information. In other words, programmers should not have to wonder whether there is a problem with the refactoring. Thus:

◇ **Perceptibility.** Representations of refactoring errors should allow programmers to easily distinguish precondition violations (showstoppers) from warnings and advisories.

In the experiment, programmers using error messages confused one kind of violation for another kind. This wasted programmers' time because they tried to track down violations that did not exist. Programmers using Refactoring Annotations, which use distinct representations for distinct errors, rarely conflated different kinds of violations. Thus:

◇ **Distinguishability.** Representations of refactoring errors should allow the programmer to easily distinguish between different types of violations.

Comprehensibility, **Locatability**, and **Relationality** are similar to Shneiderman's recommendation that error messages be specific rather than general, so that the user understands the cause of the error [70, p. 59]. Likewise, **Locatability**, **Completeness**, and **Estimability** are all designed to achieve Shneiderman's recommendation for constructive guidance, so that the user can successfully recover from an error [70, p. 58]. **Perceptibility** and **Distinguishability** are similar to Nielsen's "Help users recognize…errors" and "consistency and standards," the latter of which states that "users should not have to wonder whether different words, situations, or actions mean the same thing" [55].

9.5 Related Work: Existing Research on Refactoring Errors

Research on refactoring preconditions has largely focused on ensuring behavior preservation, rather than on how to present preconditions in an understandable manner to the programmer. Opdyke showed that program behavior is preserved when certain preconditions are satisfied in the C++ programming language [58]. At about the

same time, Griswold defined preconditions for meaning-preserving program transformations for Scheme [23].

Many tools provide support for the EXTRACT METHOD refactoring, but few deviate from the wizard-and-error-message interface described in Section 4.5.1. However, some tools silently resolve some precondition violations. For instance, when you try to extract an invalid selection in Code Guide, the environment expands the selection to a valid list of statements [25]. With a tool that makes such a strong assumption about what you wanted, you may then end up extracting more than you intended. With Xrefactory, if you try to use EXTRACT METHOD on code that would return more than one value, the tool generates a new tuple class [86]. Again, this tool makes strong assumptions about what you intended.

9.6 Generalization to Other Refactorings

Thus far in this chapter, I have demonstrated how error messages for the EXTRACT METHOD refactoring can be improved by using a graphical representation. What I have not yet discussed is whether the guidelines presented in Section 9.4 are generalizable to precondition violations for other refactorings. In the remainder of this chapter, I characterize precondition violations and explain how the existing guidelines can be extended to other precondition violations (Section 9.6.1). I also describe an evaluation of this extension (Section 9.6.2).

9.6.1 A Taxonomy of Refactoring Preconditions

It would be ideal to define a taxonomy of refactoring preconditions by studying a formal, language-agnostic foundation of refactorings, but such a foundation does not yet exist. Perhaps the closest such foundation that I know of is Opdyke's dissertation, in which Opdyke rigorously studied refactorings for the C++ language [58], but the refactorings that he studied are based on refactorings observed in just one case

study. Thus, Opdyke's refactorings (and preconditions) are not complete, nor necessarily representative of the variety of refactorings that most programmers perform. Indeed, much of what is known about refactoring preconditions is language specific and has little formal foundation. Refactoring preconditions vary from refactoring to refactoring, and from language to language. However, there are some preconditions common to many refactorings across several languages. For example, whether performing EXTRACT LOCAL VARIABLE or RENAME CLASS, whether refactoring in Java or in C++, the name of the new variable or class must be different from the name of any existing variable or class in scope.

Instead of a formal foundation, I built my taxonomy of preconditions on error messages displayed by existing refactoring tools. This is because, in practice, much information about refactoring preconditions is buried in the refactoring tools themselves, although how many preconditions a particular tool checks for depends on the tool's maturity. For example, while a refactoring toolsmith may take great pains to ensure that a tool checks all preconditions that she can think of, a few unanticipated preconditions inevitably slip through the cracks [83] and may emerge sometime down the road in the form of bug reports. In this section, I identify refactoring preconditions by examining the internals of refactoring tools.

9.6.1.1 Methodology for Deriving a Precondition Taxonomy

My goal is to draw general conclusions about how to represent preconditions. Thus I am not overly concerned about undiscovered refactoring preconditions, as long as I can say I have studied a fairly representative sample of refactoring preconditions. To find such a sample, I have classified refactoring preconditions detected by 4 different refactoring tools.

- **Eclipse JDT.** This tool is the standard Eclipse Java refactoring tool [20]. I gathered preconditions from a key-value text file used to store each precon-

dition error message. This is the most mature refactoring tool in the group, containing 537 error messages in total. I inspected a version from the Eclipse CVS repository checked out on August 4, 2008.

- **Eclipse CDT.** This tool is built for C++ as a plugin [17] to the C/C++ environment for Eclipse [19]. I gathered precondition error messages in the same way as with Eclipse JDT. This refactoring tool contained a total of 77 error messages. I inspected version "0.1.0.qualifier".

- **Eclipse RDT.** This tool is built for the Ruby environment for Eclipse [2]. I gathered precondition error messages in the same way as with Eclipse JDT, although the error messages were spread among several files. This refactoring tool contained a total of 73 error messages. I inspected Subversion revision 1297.

- **HaRe.** This tool is built for refactoring Haskell programs [79]. I gathered precondition error messages by searching for invocations of the `error` function, which was typically followed by an error message that indicated a violated refactoring precondition. This refactoring tool contained a total of 204 error messages. I inspected the March 27, 2008 update of HaRe 0.4.

9.6.1.2 Taxonomy Description

For each error message, I manually gave it a primary classification based on similarities to other error messages, and I iterated on this taxonomy until similar error messages appeared in the same category. Table 9.2 on page 159 shows my final taxonomy. Categories are indented when they are a subcategory; for instance, inaccurate analysis is a kind of analysis problem. You should note that the number of error messages in each taxonomy category is not indicative of the importance of a particular category. This is because some general error messages that apply to several refactor-

ings appear just once, and also because some tool categories are unpopulated because of the relative immaturity of the tool. Next, I explain each category, provide a few example error messages from actual tools, and describe how my guidelines apply to that category. I will also provide mockups of how Refactoring Annotations can be extended to preconditions in the taxonomy. A database containing the taxonomized error messages can be found at http://multiview.cs.pdx.edu/refactoring/error_taxonomy.

- Analysis problems occur when the refactoring tool encounters problems while analyzing code. This category is subdivided into inaccurate analysis, incompatibility, compilation errors, internal error, and inconsistent state.

  – Inaccurate analysis occurs when the refactoring engine cannot analyze some piece of source code, and thus is not confident about the accuracy of the results. Below are some examples of error messages in this category (··· denotes text that a refactoring tool supplies dynamically).

Example: Inaccurate Analysis Errors

Tool	Refactoring	Message
JDT	EXTRACT METHOD	Several type parameters cannot be mapped to the super type ···
CDT	ENCAPSULATE FIELD	No such method ···
RDT	INLINE METHOD	Sorry, could not guess the type of the selected object.
HaRe	Multiple	Can't find module name

Errors in inaccurate analysis are too vague to provide any useful representation to the programmer. For the most part, this category encapsulates exceptions that are encountered in situations that the toolsmith did not expect, and thus little can be done to improve error expressiveness to the

Category	JDT	CDT	RDT	HaRe
analysis problem	0	0	0	0
inaccurate analysis	35	4	2	2
incompatibility	5	1	0	0
compilation errors	27	1	3	0
internal error	24	5	0	36
inconsistent state	15	2	0	0
unsaved	4	1	0	0
deleted	4	0	0	0
misselection	0	0	0	0
selection not understood	30	26	19	33
improper quantity	0	5	0	2
misconfiguration	3	0	0	0
illegal name	6	7	1	15
unconventional name	11	0	0	0
identity configuration	4	0	7	3
unbound configuration	7	2	0	2
unchangeable	3	0	0	0
unchangeable reference	2	0	0	0
unchangeable source	16	0	0	0
unchangeable target	12	0	0	0
unbinding	12	0	0	1
control unbinding	35	2	4	10
data unbinding	18	5	3	4
name unbinding	20	0	0	2
inheritance unbinding	11	0	1	0
clash	6	5	0	24
control clash	17	3	5	9
data clash	16	0	3	3
name clash	38	3	0	2
inheritance clash	9	0	0	0
inherent	0	0	0	0
context	38	0	7	4
own parent	4	0	0	0
structure	17	0	13	9
property	45	3	3	0
vague	37	1	0	22
unknown	6	1	2	21

Table 9.2: A precondition taxonomy (left column), with counts of error messages in each taxonomy category for each refactoring tool (right columns).

programmer. The error and the programmers' context may be relayed back to the toolsmith, so that the tool can be improved in the future.

– Incompatibility refers to errors that occur when the refactoring tool tries to produce code that the current, outdated compiler does not understand.

Example: Incompatibility Errors

Tool	**Refactoring**	**Message**
JDT	PULL UP	Moving ··· which contains varargs to destination type ··· will result in compile errors, since the destination is not J2SE 5.0 compatible.
CDT	Multiple	Unsupported Dialect.

Errors in the incompatibility category can be improved in two ways in accordance with the guidelines. **Comprehensibility** may be improved by providing links to help documents that explain why one type of refactoring can have incompatibility problems. For instance, for the second example in the table, the help document could state what varargs are and that they do not exist in versions of Java prior to 5.0. **Locatability** may be improved by showing which program elements are incompatible, and also where compatibility information can be changed. For example, fixing Java 5.0 incompatibility issues can be achieved by changing project settings in Eclipse, so showing how the programmer can access those settings improves **locatability**.

– Compilation errors occur when the refactoring tool cannot analyze source code because it contains compilation errors.

Example: Compilation Error Errors

Tool	Refactoring	Message
JDT	Inline Method	The method declaration contains compile errors. To perform the operation you will need to fix the errors.
CDT	Multiple	The translationunit contains one or several problems. This can be caused by a syntax error in the code or a parser flaw. The refactoring will possibly fail.
RDT	Multiple	There is a syntax error somewhere in the project, the refactoring might not work on these files.

Improving compilation errors is largely a matter of improving representation of compilation errors themselves, which has already been a subject of extensive research [8, 30, 56, 80, 87]. For refactoring, **Relationality** and **locatability** can be improved by showing which compilation errors will have an impact on the correctness of a refactoring, and by prioritizing those that are most likely to interfere with correctness. For example, when inlining a private method, fixing compilation errors within the method's containing class would increase the likelihood of maintaining behavior, but fixing compilation errors on all other classes will not improve correctness, in general. **Estimability** may be improved by displaying how many compilation errors exist, especially on relevant program elements, but again, this depends on the usability of the underlying tool producing the compilation errors.

– Internal error occurs when the refactoring tool encounters an error for an unknown reason.

Example: Internal Error Errors

Tool	**Refactoring**	**Message**
JDT	EXTRACT INTERFACE	An internal error occurred during precondition checking. See the error log for more details.
CDT	HIDE METHOD	Can not load translation unit.
HaRe	Multiple	HaRe: Error in addLocInfo!

Internal errors, like those in the inaccurate analysis category, are bugs in the tool, and thus the toolsmith could not provide any useful representation for the error.

– Inconsistent state occurs when some program element changes between when a programmer initiated a refactoring and when the refactoring tool tries to analyze it.

Example: Inconsistent State Errors

Tool	**Refactoring**	**Message**
JDT	Multiple	The workspace has been modified since the refactoring change object has been created
CDT	Multiple	File not found.

Several improvements can be made to errors in this category and subcategories. **Locatability** may be improved by showing which program elements have changed. **Estimability** may be improved by displaying how many program elements have changed. **Relationality** may be improved by showing the relationship between the original and the changed program element. This category is subdivided into unsaved and deleted.

∗ Unsaved occurs when the refactoring tool tries to analyze or modify code that is contained inside of an open editor buffer.

Example: Unsaved Errors

Tool	Refactoring	Message
JDT	RENAME TYPE	Type ··· does not exist in the saved version of ···
CDT	HIDE METHOD	Editor is not saved

In addition to the improvements described in inconsistent state, **comprehensibility** of unsaved errors may be improved by showing the programmer the difference between the saved and unsaved code using a difference viewer.

* Deleted occurs when the refactoring tool tries to analyze or modify code that has been deleted, possibly because the programmer chose code for refactoring in a view that has not been updated since the deletion.

Example: Deleted Errors

Tool	Refactoring	Message
JDT	RENAME FIELD	The selected field has been deleted from ...

In addition to the improvements described in inconsistent state, **comprehensibility** may be improved if the refactoring tool can describe how and when the code was deleted, if that information is known.

- Misselection is an error that indicates a problem with the programmer's selection. This category is subdivided into selection not understood and improper quantity.

  – Selection not understood occurs when the programmer selects some piece of code that the refactoring tool doesn't expect as input.

Example: Selection Not Understood Errors

Tool	Refactoring	Message
JDT	EXTRACT METHOD	The selection does not cover a set of statements or an expression. Extend selection to a valid range using the 'Expand Selection To' actions from the 'Edit' menu.
CDT	HIDE METHOD	No proper Selection!
RDT	MOVE METHOD	The caret needs to be inside of a class.
HaRe	REMOVE PARAMETER	The selected identifier is not a function/simple pattern name, or is not defined in this module

Like other violations, this kind of violation can be avoided with tools such as Box View (Section 6.2.2) or refactoring cues (Section 6.6.3). When such a tool is not used, one precondition guideline may help improve the **comprehensibility** of selection not understood violations. The tool can indicate what kind of program element it does expect, can give concrete examples of such program elements in the programmer's current editor, and can allow the programmer to change her selection to the nearest such examples.

– Improper quantity occurs when the programmer did not select the expected number of program elements.

Example: Improper Quantity Errors

Tool	Refactoring	Message
CDT	EXTRACT CONSTANT	Too many literals selected.
HaRe	MERGE DEFINITIONS	Only two functions may be fused together!

Comprehensibility of improper quantity violations may be improved in a similar way as selection not understood, by suggesting some additions or reductions to the selection that would make it a valid selection, and then

making it quick and easy to change to one of those options. **Comprehensibility** may also be improved by saying exactly how many program elements are expected. **Locatability** and **estimability** may be improved by highlighting each of the selected program elements and allowing the programmer to individually deselect or reselect elements in the existing and adjacent highlights.

- Misconfiguration is a violation that occurs as the programmer chooses options when configuring a refactoring. Violations in this category could likely be pushed into subcategories, if the messages were more specific.

Example: Misconfiguration Errors

Tool	**Refactoring**	**Message**
JDT	RENAME PACKAGE	Choose another name.

This category is subdivided into illegal name, unconventional name, identity configuration, and unbound configuration.

– Illegal name occurs when the programmer enters the name of a program element to be created, but the name violates the rules of the programming language.

Example: Illegal Name Errors

Tool	**Refactoring**	**Message**
JDT	Multiple	Type name cannot contain a dot (.).
CDT	Multiple	· · · contains an unidentified mistake.
RDT	RENAME	Please enter a valid name for the variable.
HaRe	RENAME	The new name should be an operator!

Illegal name violation representations can be improved in terms of **comprehensibility** by displaying what characters (or character combination)

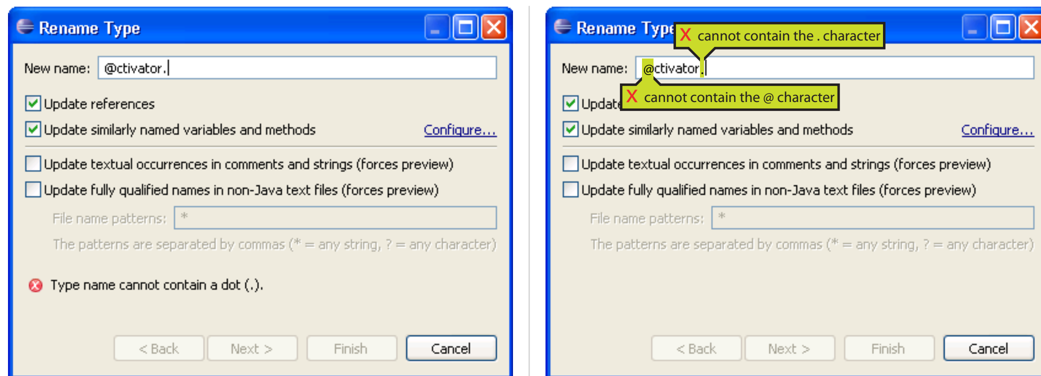


Figure 9.4: Illegal name violations, displayed normally in Eclipse (at left), and how such violations would be implemented following the guidelines (at right). The green violation indicators at right indicate that two invalid characters were typed into the new name text field.

are invalid (such as in the first error message) and, if possible, what characters are valid. **Locatability** may be improved by pointing at which entered character or characters are invalid. **Estimability** may be improved by pointing at each and every invalid character. Figure 9.4 shows an example of what such a user interface might look like.

– Unconventional name occurs when the programmer enters the name of a program element to be created, but that name violates some convention.

Example: Unconventional Name Errors

Tool	Refactoring	Message
JDT	EXTRACT LOCAL VARIABLE	This name is discouraged. According to convention, names of local variables should start with a lowercase letter.

Unconventional name violation representations can be improved in the same way as those for illegal name, although the representations for unconventional name should be displayed in a manner that conveys less urgency.

– Identity configuration occurs when the programmer enters input that results in a refactoring that does not not modify the underlying source code.

Example: Identity Configuration Errors

Tool	**Refactoring**	**Message**
JDT	CHANGE SIGNATURE	Method signature and return type are unchanged.
RDT	RENAME	Nothing to refactor.
HaRe	RENAME	The new name is same as the old name

The guidelines do not appear to dictate any particular way to improve identity configuration violation representations. It may be more fruitful to address identity configuration in the "interpret results" step of the refactoring process (Section 2.5).

– Unbound configuration occurs when the programmer types configuration information into a refactoring tool and that information should, but does not, refer to some programming element.

Example: Unbound Configuration Errors

Tool	**Refactoring**	**Message**
JDT	CHANGE SIGNATURE	··· is not a valid return type.
CDT	Multiple	No return-value (void).
HaRe	ADD FIELD	Please enter a field name (_ to omit) and a field type!

Unbound configuration may be prevented by providing a default value for the input in question and, as discussed in Chapter 6, allowing only valid input to be selectable. For instance, the first CHANGE SIGNATURE error message could be eliminated if a widget were used where the programmer could select only from valid types. When preventing unbound

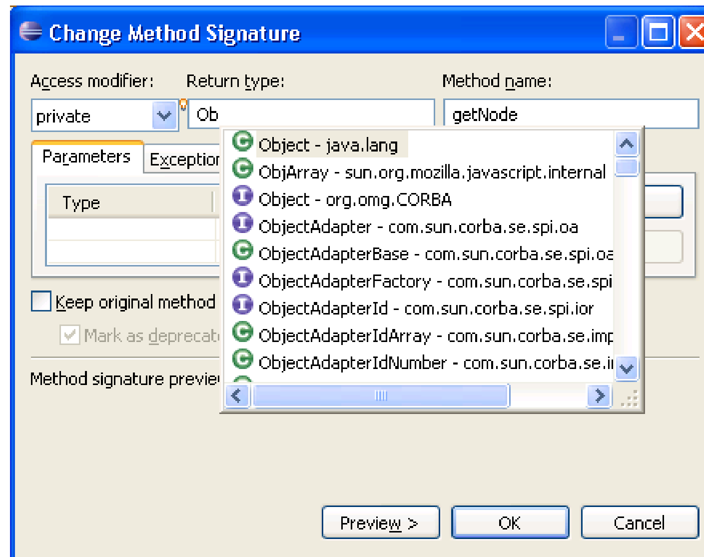


Figure 9.5: Eclipse offering a quick-assist of all available return types in a refactoring dialog.

configuration violations is not desirable, comprehensibility may likewise be improved by allowing the programmer to see an example of input that is bound to some existing program element. Eclipse currently offers this functionality using a quick-assist to choose from available types for several refactorings (Figure 9.5), although it still accepts arbitrary input into the text box, only to later display an error message if that text does not indicate an existing type.

- Unchangeable violations are those that occur because some programming element cannot be modified. Error messages that appear in this category should be placed in subcategories, but were too vague to be classified appropriately.

Example: Unchangeable Errors

Tool	**Refactoring**	**Message**
JDT	MULTIPLE	· · · is read only

Comprehensibility may be improved if the type of the unchangeable element

is displayed, such as whether it is binary or has been tagged as read-only. **Locatability** may be improved if the unchangeable element or elements are displayed to the programmer. **Estimability** may be improved if each unchangeable program elements are shown, or if the total number of unchangeable program elements is shown. This category is subdivided into unchangeable reference, unchangeable source, and unchangeable target.

– Unchangeable reference violations occur when a reference to a refactored program element cannot be modified.

Example: Unchangeable Reference Errors

Tool	**Refactoring**	**Message**
JDT	INTRODUCE FACTORY	Constructor call sites in binary classes cannot be replaced by factory method calls.
JDT	MOVE METHOD	The method invocations to the moved method in resource ··· cannot be updated.

Apart from the improvements stated for the category unchangeable, unchangeable reference violation representations may improve **relationality** by showing the relationships between the referencer and the referenced program element.

– Unchangeable source violations occur because the original program element to be refactored cannot be modified.

Example: Unchangeable Source Errors

Tool	Refactoring	Message
JDT	CHANGE TYPE	Type of selected declaration cannot be changed
JDT	PULL UP	Pull up is not allowed on members of binary types.
JDT	Multiple	Moving of members declared in read-only types is not supported

Unchangeable source may be improved in the same way as unchangeable.

– Unchangeable destination violations occur when the programmer requests to have some program element moved to a location that cannot be modified

Example: Unchangeable Destination Errors

Tool	Refactoring	Message
JDT	INTRODUCE INDIRECTION	Cannot place new method in a binary class.
JDT	Multiple	The selected destination is not accessible

Unchangeable destination may be improved in the same manner as unchangeable.

Unbinding occurs when the refactoring tool tries to modify some code that contains references, but doing so would cause those references to become unresolved.

Example: Unbinding Errors

Tool	**Refactoring**	**Message**
JDT	PUSH DOWN	Pushed down member ··· is referenced by ···
HaRe	ADD DEFINITION	The free identifiers: ··· which are used by the definition to be moved are not in scope in the target module ··· !

Error messages that appear in this category are too vague to be pushed down into a subcategory, but I suspect that most such error messages in context could be placed in a subcategory. In general, unbinding violations refer to both "source" and "destination" program elements. For example, the source in the first error message in the table is the class that originally contained the member, and the destination is the subclass that the member would be pushed down to. Moreover, referencing program elements, either to the source or the target, may be relevant. Thus, **Locatability** may be improved by displaying source, target, and referencing program elements, and **relationality** may also be improved by showing the relationship between these elements. Moreover, because relationships are broken in unbinding violations, changes to that relationship should be shown. **Estimability** may be improved by displaying how many relationships are broken.

This category is subdivided into control unbinding, data unbinding, name unbinding, and inheritance unbinding.

– Control unbinding occurs when the refactoring tool tries to modify some code that contains control flow references (method calls, function calls, or inter-procedural control flow), but doing so would break those references.

Example: Control Unbinding Errors

Tool	**Refactoring**	**Message**
JDT	MOVE MEMBERS	Accessed method ··· will not be visible from ...
CDT	EXTRACT METHOD	Extracting break statements without the surrounding loop is not possible. Please adjust your selection.
RDT	EXTRACT METHOD	Extracting methods not possible when the selected code contains super calls
HaRe	MOVE DEFINITION	The 'main' function defined in a 'Main' module should not be renamed!

Improving inter-procedural **Relationality** may be done using control-flow techniques already described using Refactoring Annotations for EXTRACT METHOD. **Relationality** between methods and method calls is slightly more difficult because such control flow does not necessarily have any intuitive direction. However, if you extend the "top is value in-flow and bottom is value out-flow" metaphor used in Refactoring Annotations for EXTRACT METHOD, then you have "top is method/function callers and bottom is method/function callees." Thus, any lines coming in the top of function represent callers and lines coming out the bottom represent callees. Xs can then be placed where such relationships are broken, and the programmer can click on the unattached end the relationship to go the source. Figure 9.6 on the next page is an example mockup for an unsuccessful MOVE METHOD refactoring.

– Data unbinding occurs when the refactoring tool tries to modify some code that contains references to or from variables, and the modification would break those references.

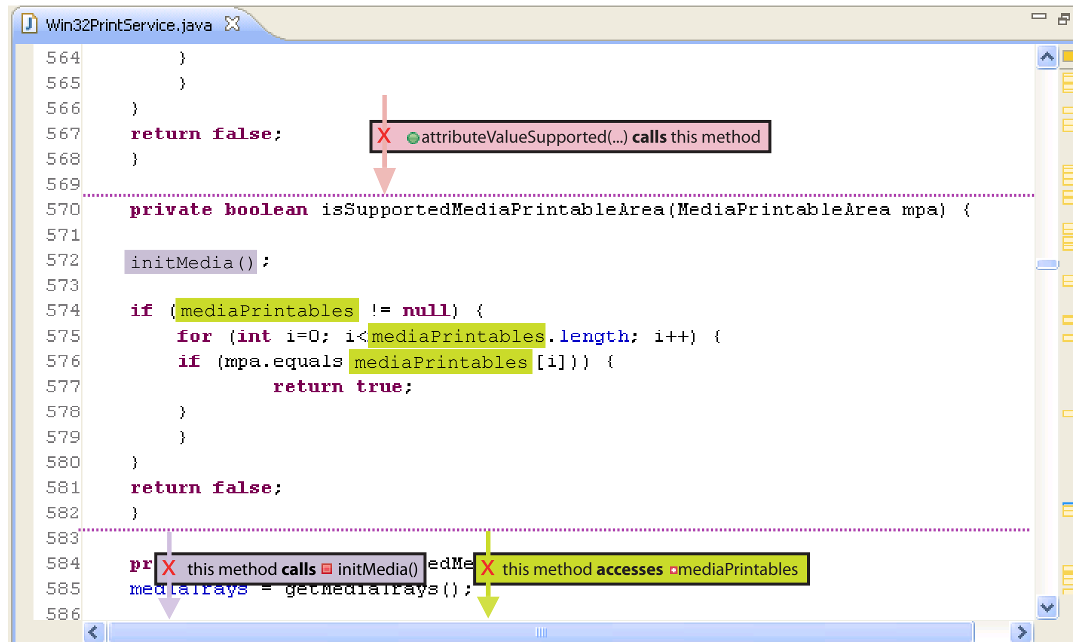


Figure 9.6: A mockup of how the guidelines inform the display of control unbinding (top and bottom left) and data unbinding (bottom right) for an attempted MOVE METHOD refactoring. The purple top annotation indicates that `isAttributeValueSupported(...)` calls this method, which is a problem because this method would not be visible outside in the destination. The `initMedia()` annotations indicate that this method calls the `initMedia()` method, which would not be visible from the destination. The `mediaPrintables` annotations indicate that this method uses the `mediaPrintables` field, which would not be visible from the destination.

Example: Data Unbinding Errors

Tool	Refactoring	Message
JDT	CONVERT ANONYMOUS TO NESTED	Class accesses fields in enclosing anonymous type. The refactored code will not compile.
CDT	MOVE FIELD	Field is used in a Subclass. Maybe the target class is not visible in the subclass and the refactoring will result in compiler errors.
RDT	MOVE METHOD	The method ··· contains the class field ···. Moving it might affect the functionality of the class ···.
HaRe	REMOVE PARAMETER	This parameter can not be removed, as it is used!

Data unbinding may be improved in the same way as control unbinding. Figure 9.6 on the preceding page shows an example mockup.

– Name unbinding occurs when the refactoring tool tries to modify some code that contains references to or from named entities (such as types), but doing so would break those references or cause them to be illegal in some other sense.

Example: Name Unbinding Errors

Tool	Refactoring	Message
JDT	MOVE MEMBER	Accessed type ··· will not be visible from ···
HaRe	MOVE FUNCTION	Moving the definition to module ··· will cause mutually recursive modules!

Name unbinding may be improved in the same way as control unbinding. Figure 9.7 on the next page is an example mockup for an unsuccessful MOVE METHOD refactoring.

– Inheritance unbinding occurs when the refactoring tool tries to modify some code that contains inheritance relationships, but doing so would break those references.

Example: Inheritance Unbinding Errors

Tool	Refactoring	Message
JDT	RENAME FIELD	Cannot be renamed because it is declared in a supertype
RDT	INLINE CLASS	The inlne [sic] target is subclassed and thus cannot be inlined.

Inheritance unbinding may be improved in the same way as control unbinding. Extending the top/bottom metaphor, if a name is used inside

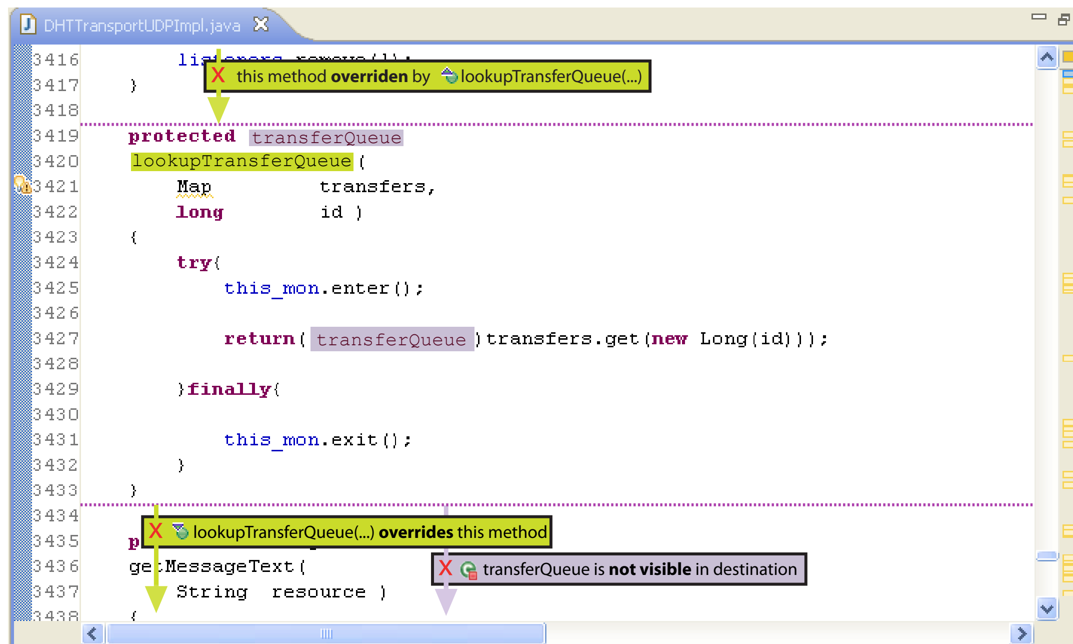


Figure 9.7: A mockup of how the guidelines inform the display of name unbinding (in purple) and inheritance unbinding (in green) for an attempted Move Method refactoring, where the destination class is the class of `this_mon`. The purple `tranferQueue` annotations indicate that this method relies on a class `transferQueue`, which will not be accessible in the destination. The green `lookupTransferQueue` annotations indicate that the current method overrides a superclass method (top) and some subclass method (bottom), so the method cannot be moved.

a code block, then lines come out the bottom of that block; if that block defines a name that is used elsewhere, then lines come in the top. Figure 9.7 shows an example mockup for an unsuccessful Move Method refactoring.

- Clash refers to a conflict that would be introduced if the refactoring is executed. Violations in this category should be pushed down to subcategories when they can be made more concrete.

Example: Clash Errors

Tool	Refactoring	Message
JDT	Multiple	Problem in ⋯. Another name will shadow access to the renamed element
CDT	Multiple	⋯ is already defined in this scope.
HaRe	RENAME	Name ⋯ already existed

Locatability of clash violations may be improved by displaying the program elements that clash, both the original and the new element.

This category is subdivided into control clash, data clash, name clash, and inheritance clash.

– Control clash occurs when a refactoring tool needs to create a new method or function that clashes with an existing method or function.

Example: Control Clash Errors

Tool	Refactoring	Message
JDT	Multiple	If you proceed, the method ⋯ in ⋯ will have a constructor name.
CDT	EXTRACT METHOD	Name already in use.
RDT	Multiple	New method name is not unique.
HaRe	MERGE DEFINITIONS	the use of the name: ⋯ is already in scope!

Control clashes should be displayed the same as general clashes. An example is shown in Figure 9.8 on the next page.

– Data clash occurs when a refactoring tool needs to create a new data element, but the new data element clashes with an existing data element.

```
  */
  public native void isValid() throws SyncFailedException;
/
 * Tests if this file descriptor object is valid.
 *
 * @return  <code>true</code> if the file descriptor object represents a
 *          valid, open file, socket, or other active I/O connection;
 *          <code>false</code> otherwise.
 */
public boolean isValid() {
return ((handle != -1) || (fd != -1));
}
```

X isValid already exists

Figure 9.8: A mockup of how the guidelines inform the display of control clash for an attempted RENAME METHOD refactoring, where the method at bottom has just been renamed to `isValid()` using Eclipse's in-line rename refactoring tool. At top, the existing method that the newly renamed method conflicts with, in a floating editor that can be used to perform recursive refactorings, such as renaming the original `isValid()` method.

Example: Data Clash Errors

Tool	Refactoring	Message
JDT	RENAME FIELD	A field with name ··· is already defined in ···.
RDT	RENAME	The chosen variable name already exists! Please go back and change it.
HaRe	ADD PARAMETER	The new parameter name will cause name clash or semantics change, please select another name!

Data clashes should be displayed the same as general clashes.

– Name clash occurs when a refactoring tool needs to create a new program element, but the new program element's name clashes with an existing program element's name. These program elements are typically types or modules.

Example: Name Clash Errors

Tool	Refactoring	Message
JDT	EXTRACT INTERFACE	A type named ⋯ already exists in package ⋯
CDT	Multiple	File already exists: ⋯
HaRe	MOVE DEFINITION BETWEEN MODULES	The pattern names: ⋯ are already defined in module ⋯ !

Name clashes should be displayed the same as general clashes.

– Inheritance clash occurs when the refactoring tool attempts to create a program element that will clash with another program element in the inheritance hierarchy.

Example: Inheritance Clash Errors

Tool	Refactoring	Message
JDT	PULL UP	Field ⋯ declared in type ⋯ has a different type than its moved counterpart

Inheritance clashes should be displayed the same as general clashes.

- Inherent is a category of violations where some input to the refactoring tool is inherently not suitable for refactoring. This category is subdivided into context, structure, and property.

  – Context occurs when the program element to be refactored, or the resulting program element after refactoring, is not appropriate because of the program context in which it occurs.

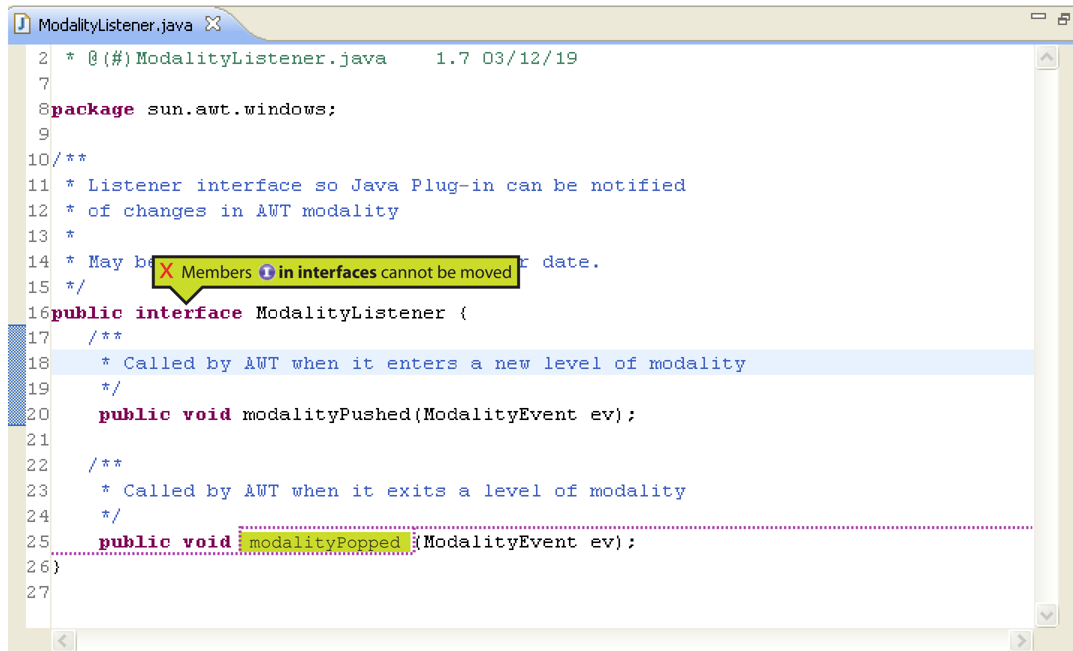


Figure 9.9: A mockup of how the guidelines inform the display of context for an attempted MOVE METHOD refactoring, pointing out that the method `modalityPopped(...)` cannot be moved because interface methods cannot be moved. The original Eclipse modal error message states "Members in interfaces cannot be moved."

Example: Context Errors

Tool	Refactoring	Message
JDT	EXTRACT METHOD	Cannot extract increment part of a 'for' statement.
RDT	TEMP TO FIELD	There is no enclosing class to insert fields.
HaRe	CLEAN IMPORTS	The selected identifier is not a top level identifier!

Locatability and **Relationality** may be improved by displaying the context, or lack thereof, that is causing the problem. Figure 9.9 shows an example. This category is contains the more specialized own parent category.

* Own parent violations occur when the programmer requests to move

a program element to its parent.

Example: Own Parent Errors

Tool	Refactoring	Message
JDT	MOVE	A file or folder cannot be moved to its own parent.
JDT	MOVE	A package cannot be moved to its own parent.

Own parent is similar to identity configuration, in that it is not clear that this should be a violation at all, but instead information that tells the programmer that nothing has changed during the "interpret results" step (Section 2.5).

– Structure violations occur when a refactoring cannot proceed because of the structure of the program element being refactored.

Example: Structure Errors

Tool	Refactoring	Message
JDT	INLINE METHOD	Method declaration contains recursive call.
RDT	MERGE WITH EXTERNAL CLASS PARTS	There is no class in the current file that has external parts to merge
HaRe	MERGE DEFINITIONS	The guards between the two functions do not match!

Locatability may be improved by making the structure of the program element to be refactored more explicit. **Relationality** may be improved by relating relevant pieces of the structure with one another. Figure 9.10 on the next page and 9.11 on page 182 shows two examples. To improve **Comprehensibility**, supplementary explanation may be required in Figure 9.11 on page 182 to explain what a "blank final" is.

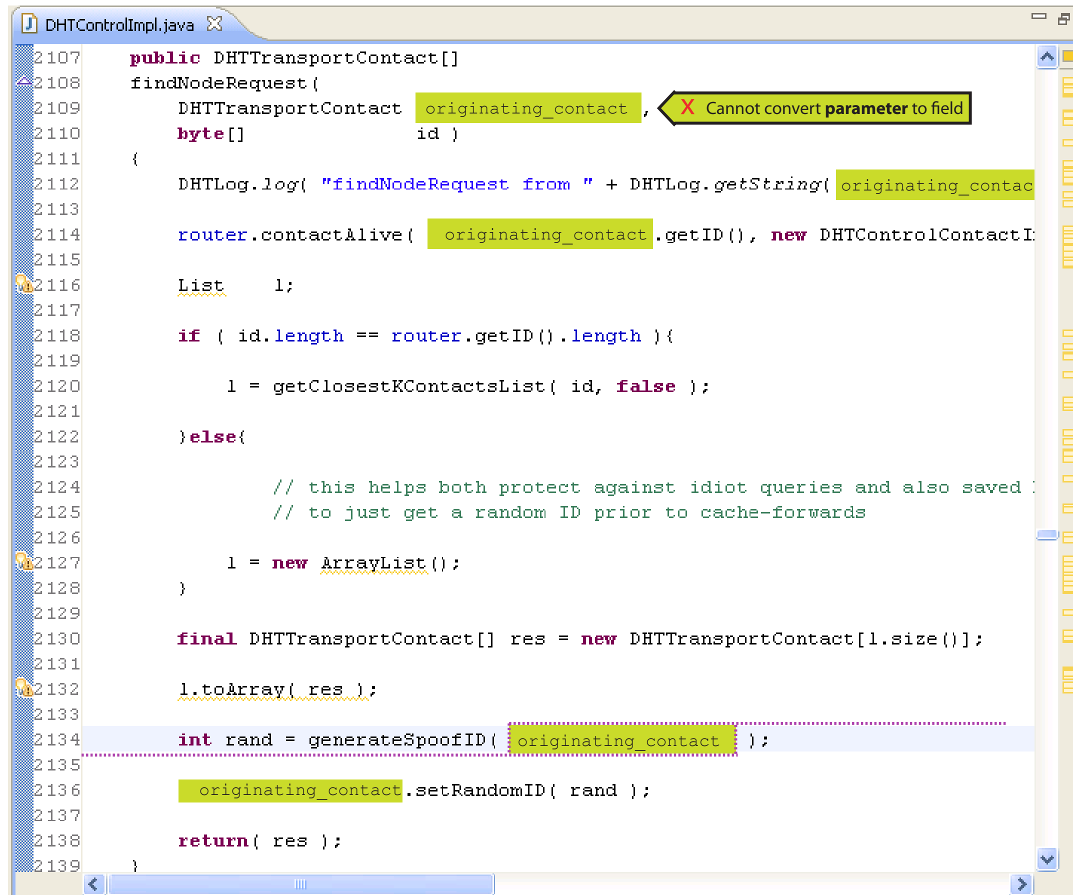


Figure 9.10: A mockup of how the guidelines inform the display of structure for an attempted CONVERT LOCAL TO FIELD refactoring, pointing out that the selected variable `originating_contact` is a parameter, which cannot be inlined. The original Eclipse modal error message states "Cannot convert method parameters to fields."

– Property violations occur when the selected program element to be refactored has some property that makes it unsuitable for refactoring.

ProcessEnvironment.java

```
187     // Allow `=' as first char in name, e.g. =C:=C:\DIR
188     static final int MIN_NAME_LENGTH = 1;
189
190     private static final NameComparator nameCompara
191     private static final EntryComparator entryComp
192     private static final ProcessEnvironment theEnvironment;
193     private static final Map<String,String> theUnmodifiableEnvironment;
194     private static final Map<String,String> theCaseInsensitiveEnvironment;
195
196     static {
197     nameComparator  = new NameComparator();
198     entryComparator = new EntryComparator();
199      theEnvironment  = new ProcessEnvironment();
200     theUnmodifiableEnvironment
201         = Collections.unmodifiableMap(theEnvironment);
202
203     String envblock = environmentBlock();
204     int beg, end, eql;
205     for (beg = 0;
206          ((end = envblock.indexOf('\u0000', beg  )) != -1 &&
207           // An initial `=' indicates a magic Windows variable name -- OK
208           (eql = envblock.indexOf('='     , beg+1)) != -1);
209          beg = end + 1) {
```

X cannot inline **blank** finals

Figure 9.11: A mockup of how the guidelines inform the display of structure for an attempted INLINE CONSTANT refactoring, pointing out that the selected constant `theEnvironment` is blank, meaning that it is not assigned to at its declaration. The original Eclipse modal error message states "Inline Constant cannot inline blank finals."

Example: Property Errors

Tool	**Refactoring**	**Message**
JDT	EXTRACT LOCAL	Operation not applicable to an array initializer.
CDT	MOVE FIELD	Move Field can't move a Method use Move Method instead.
RDT	INLINE LOCAL	Cannot inline method parameters.

Property violations are too varied to suggest any general way to apply my guidelines.

- Vague includes error messages that I could not classify from the message alone.
- Unknown similarly includes error messages that I simply did not understand.

Several caveats are worth stating before drawing conclusions about how the guidelines apply to the taxonomy of refactoring preconditions:

- The taxonomy shown is a best-effort attempt to classify real-world error messages; similar error messages may appear in different categories. The size of the error message corpus (891 messages in total) combined with the textual inconsistencies between messages mean that doing a completely accurate classification is nearly impossible. For example, several completely different EXTRACT METHOD error messages appear to refer to the same violation: "No statement selected," "There is nothing to extract," and "Cannot extract a single method name." While these messages may have identical causes, they have no words in common. Indeed, messages are difficult to classify because they can be stated in ways that are positive or negative, constructive or declarative, programmer-oriented or tool-oriented.

- Likewise, some messages could be placed in more than one category. For example, the error message "The elements in project ⋯ referenced from ⋯ cannot be updated, since the project is binary" could appear in either in unchangeable or in unbinding. As another example, the error message "Removed parameter ⋯ is used in method ⋯ declared in type ⋯" could appear in either inheritance unbinding or in data unbinding. However, I put each message into one and only one category. Moreover, because I created the taxonomy as I inspected the error messages, which category each message went into is based on what categories were known when I inspected each message. Notwithstanding this ambiguity, not every category can overlap with another category. For example, given a selection not understood violation, that violation cannot also be an unbinding or clash message, because unbinding and clash messages appear only when the refactoring tool understands the pro-

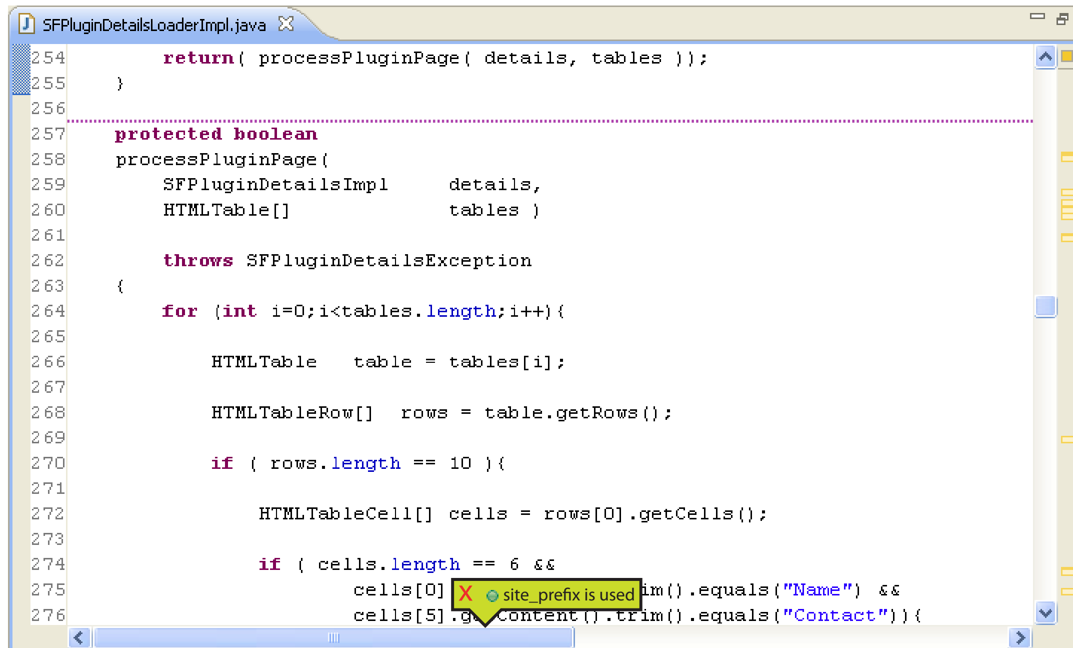


Figure 9.12: A mockup of how the non-local violations can be displayed in the program editor. Here, the variable `site_prefix` is referenced somewhere further down the editor.

grammer's selection.

- Thus far, in this section on violation generalization, I have chosen examples in the figures that fit onto one editor screen. What happens if a violation is not located on screen in the editor? In that case, the guideline for **locatability** (showing the violation close to relevant program elements) conflicts with the guideline for **completeness** (showing all preconditions at once). Figure 9.12 gives an example of a refactoring where one precondition is violated, but the editor annotation that would normally be displayed onscreen is indicating that part of the violation is displayed somewhere further down the editor. Figure 9.8 on page 177 gives another example of how this problem can be addressed using a floating editor. Using these techniques, such annotations can maintain **locatability** without sacrificing **completeness**.

9.6.1.3 Application of the Remaining Guidelines to the Taxonomy

As I postulated in Section 9.4, **completeness** (showing every refactoring precondition violation) is an important guideline to a usable user-interface for representing violations of refactoring preconditions. **Completeness** may be achieved in the same way as by Refactoring Annotations in EXTRACT METHOD: by overlaying each message on top of program code or in the configuration user interface. For example, in Figure 9.6 on page 173, three different Refactoring Annotations are represented at once.

Likewise, I postulated **perceptibility** (enabling the programmer to distinguish easily violations from non-violations) as a usability guideline. Again, as with Refactoring Annotations for Extract Method, Refactoring Annotations in general may achieve **perceptibility** by drawing Xs for every violated precondition.

The last guideline that I postulated was **distinguishability**, allowing the programmer to easily distinguish between different types of precondition violations. With standard Refactoring Annotations, I provided a representation for two different categories in the taxonomy, control unbinding, the black lines in the margins, and data unbinding, the colored lines. In this section, I've proposed more than two dozen categories of violated refactoring preconditions; how can the programmer distinguish all of them?

Traditional error messages use two different mechanisms to help programmers distinguish between different kinds of messages:

- The text of the error message. As I have mentioned, however, programmers sometimes do not read these messages (Section 4.5).
- The timing of the error message. Different messages may appear at different times; for example, selection not understood always occurs near the beginning of the use of a refactoring tool, while name clash only occurs after the

programmer has done some configuration. Because the original Refactoring Annotations all appeared at the same time during refactoring, it is unclear how effective timing is in helping programmers to distinguish between precondition violations.

Furthermore, there are at least three other ways to help programmers distinguish between different kinds of messages:

- An iconic representation of the violation. For example, a series of 1's and 0's is a familiar iconic representation of "binary," and thus may be useful for representing the unchangeable category when the unchangeable element is binary, such as a library. Likewise, in Figures 9.6 on page 173 and 9.7 on page 175, Refactoring Annotations use field, method, and class icons to distinguish between data unbinding, control unbinding, and name unbinding, respectively. Even if the meaning of the icons is not intuitively grasped immediately, if the icons are encountered with enough frequency, programmers may be able to recognize problems from icons more quickly than by reading text.

- Shapes and patterns. Like icons, shapes and patterns may be used to help programmers distinguish between different kinds of precondition violations. For example, consider the straight arrows used in Figure 9.6 on page 173 and the curved arrows used in Figure 9.8 on page 177; differently shaped arrows represent different kinds of violations. Although the shapes do not intuitively imply different violations, I hope that the difference would help programmers distinguish between the violations, if only at a subconscious level.

- Word emphasis. Some error messages are overly wordy, making them difficult to understand. This difficulty arises because the cause of the problem may not be apparent, as it is hidden in prose that is not central to the problem.

For example, the following error message may be overly wordy: “Extracting methods is not possible when the selected Method contains internal methods.” Assuming that the programmer remembers the refactoring that she just performed and that she just selected some method, all she needs to know from an error message is *why*. In the example above, the only information pertinent to *why* can be summarized in 3 words: “contains internal methods.” Thus, emphasizing the *why* words may help improve not only **comprehensibility**, but also **distinguishability** because the programmer would have fewer words to compare when wondering “is this error message like the last one that I saw?” As a positive example, notice that in Figure 9.11 on page 182, a single word is bolded to emphasize why a constant is not suitable to be inlined.

I have presented this list of methods of distinguishing between preconditions to show that several methods exist (hopefully a sufficient number). I am not suggesting a way to apply them to the precondition taxonomy. Knowing the frequency with which these violations arise in practice and which pairs of violations are conflated by programmers should help to define what categories should be made most distinguishable. This remains future work.

9.6.2 Evaluation

In the previous section (Section 9.6.1), I discussed how my guidelines for improving the representation of precondition violations can be applied to such representations in general. The guidelines clearly *can* be applied, but it is not yet clear that the guidelines *improve* the representation of precondition violations. In this section, I describe an evaluation which suggests that that the guidelines do indeed improve usability.

To summarize, the experiment described in this section is similar to the previous experiment (Section 9.3): I ask programmers to use Refactoring Annotations and

error messages to diagnose violations of refactoring preconditions. I compare their ability to correctly identify the source of those violations, and report on their personal opinions of both tools. Unlike the previous evaluation, it is a paper-based evaluation of two hypothetical tools and explores a variety of refactorings and refactoring preconditions.

9.6.2.1 Subjects

I drew subjects from three upper-level graduate and undergraduate courses: Scholarship Skills, Advanced Programming, and Languages and Compiler Design. I encouraged students to participate in the experiment by offering 10 dollar gift cards from the Portland State bookstore to participants, requiring only that the participants have programmed in Java. Thirteen students volunteered to participate, but two had scheduling conflicts and one was somewhat familiar with this research: all three were excused. As a result, a total of 10 students participated.

Subjects reported a mean of about 6 years of programming experience and 19 hours per week of programming over the last year. Eight of the subjects used Integrated Development Environments at least part of the time when programming: these environments included Visual Studio, Eclipse, Netbeans, and Xcode. Six subjects were at least somewhat familiar with the concept of refactoring, and two of them had used refactoring tools. All subjects were at least somewhat familiar with Java and C++.

9.6.2.2 Methodology

I randomly placed subjects into one of four groups to ensure that average task difficulty was balanced, as shown in Table 9.3 on the following page. Half of the subjects used Refactoring Annotations to help them diagnose violated preconditions on 8 individual refactorings during the first phase of the experiment, then used error

	Group 1		Group 2		Group 3		Group 4	
	Tool	Order	Tool	Order	Tool	Order	Tool	Order
Phase 1	Refactoring Annotations	A	Refactoring Annotations	B	Error Messages	A	Error Messages	B
Phase 2	Error Messages	B	Error Messages	A	Refactoring Annotations	B	Refactoring Annotations	A

Table 9.3: In which order the four different groups of subjects used the two refactoring tools over the two code sets.

messages to help diagnose violated preconditions on 8 other individual refactorings in the second phase. The other half used error messages in the first phase, then used Refactoring Annotations in the second phase. Additionally, half of the subjects analyzed violations in one order (call it "A") in the first phase, then in another order (call it "B") in the second phase, and vice-versa for the other half of subjects. Of the 10 subjects who participated, I assigned two to Group 1, three to Group 2, two to Group 3, and three to Group 4.

I chose three refactorings for subjects to analyze to try to balance having a sufficient variety of refactorings and having few enough refactorings that subjects would not find it difficult to remember how the refactorings work. I selected the refactorings RENAME, EXTRACT LOCAL VARIABLE, and INLINE METHOD because they are currently among the most popular refactorings performed with tools (see Table 3.1 on page 23).

I selected violations of refactoring preconditions for the 3 refactorings by the following criteria:

- The chosen violations should span several precondition categories (Figure 9.2 on page 159), so that I am evaluating a substantial cross section of the precondition taxonomy.

- Preference should be given to violation categories to which the guidelines ap-

Refactoring Kind	**Refactoring Number**	**Category**	**Message**
Rename	1	Data Clash	A field with this name is already defined.
	2	Illegal Name	Type name cannot contain a dot (.).
Inline Method	3	Stucture	Method declaration contains recursive call.
		Inheritance Unbinding	Method to be inlined implements method from interface $\cdots$.
	4	Property	Cannot inline abstract methods.
	5	Stucture	Cannot inline a method that uses qualified this expressions.
Extract Local	6	Illegal Name	$\cdots$ is not a valid identifier.
	7	Data Clash	A variable with name $\cdots$ is already defined in the visible scope.
	8	Context	Cannot extract assignment that is part of another expression.

Table 9.4: Refactorings and precondition violations used in the experiment.

ply, so that the results of the experiment highlight the difference between the two violation representations. For example, I did not select internal errors because the guidelines do not apply to errors in this category.

- Some violation categories should appear twice, but for different refactorings, so that subjects might notice similarities between violations.

- Some refactorings should exhibit two different violated preconditions at the same time to simulate when the refactoring tool only informs the developer of the first violation that it finds (Section 4.5.3).

Table 9.4 displays the precondition violations that I chose. The **Refactoring** column refers to one of the three chosen refactorings, **Category** refers to the category in which that violation occurs, and **Message** lists a specific error message that the

Eclipse refactoring tool would display had that violation occurred. Refactoring number 3 (in the third row of Table 9.4 on the preceding page) contains two different violations; Eclipse version 3.2 reports only the upper one to the programmer.

I first randomized the order of appearance of the three kinds of refactorings (RENAME, INLINE METHOD, and EXTRACT LOCAL) and then randomized the order of each kind. In othe words, I did not mix the different kinds of refactorings to reduce the likelihood that programmers would be confused about which refactoring was being performed. The two orders in which subjects were asked to diagnose violations were A=(1,2,7,6,8,5,3,4) and B=(1,2,4,3,5,8,6,7). Finally, I selected example code for subjects to refactor (and thus to encounter the violations) from PyUnicode, a unicode Java class from the Jython project (described in Section 4.5.2), revision number 5791. This class contains 5 private inner classes that can be used to iterate over collections. This code was selected because 5 of the 9 precondition violations shown in Table 9.4 on the previous page could naturally arise when refactoring that code. I manually changed the code in two ways:

- I inserted code that would cause the programmer to violate the remaining four preconditions when refactoring, and
- I changed code to avoid making the cause of a violation trivially apparent. For example, if the error message says “A field with this name is already defined,” then the field with the same name should not appear directly adjacent to the renamed field.

This code spanned 202 lines, small enough to fit on three 8.5×11 inch sheets of paper, but large enough to contain code that could generate two violations of each refactoring precondition in Table 9.4 (once for error messages, once for Refactoring Annotations).

9.6.2.3 Example Experiment Run

When a subject arrived to participate in the experiment, I offered her a refreshment and gave her a letter of informed consent. I then asked her to complete a pre-experiment questionnaire in which she noted her programming and refactoring experience.

I gave the subject a brief overview of the experiment, and a short review of the 3 refactorings. I then told her that she was going to see 16 attempted refactorings on the same code base, but that none of these refactorings would be successful. Instead, the tool would produce either an error message in a dialog box or a graphical representation on top of the code, to indicate why the refactoring could not be performed. I told the subject that her task was to diagnose the violated precondition, and indicate the pieces of relevant source code.

Assume, for example, that a subject is assigned to Group 3 (Table 9.3 on page 189), and thus uses error messages first with precondition ordering A, then uses Refactoring Annotations with precondition ordering B. For the first task, I give the subject the code in the form of 3 pieces of 8.5×11 inch paper, placed vertically on a flip chart. Figure 9.13 on the next page depicts a simulated experiment situation. I point out what the programmer selected, which refactoring was attempted, and the error message that the programmer encountered. I then told the subject to place small sticky notes next to the code, or the refactoring tool configuration window, that they felt was responsible for the violation. In this case, the first error message encountered was number 1 (Table 9.4 on page 190), and a correct answer was to place a sticky note next to the field that the new name clashes with. The subject then indicated that she was satisfied with her response, and moved on to the next task. This task was repeated with 7 other individual refactorings, and then repeated again, in this case, with Refactoring Annotations on the same code base (but on slightly different individual refactorings) with precondition ordering B. I recorded the aggregate time

Figure 9.13: An example of an experiment run. The experiment participant (at left), considers where to place a sticky note on the code responsible for the violation. The experiment administrator (at right), records observations about the participant's reasoning regarding where he places the note.

it took for the subject to complete the 8 tasks, making one timing measurement with Refactoring Annotations and one timing measurement with error messages.

After the tasks were complete, I gave the subject a post-experiment questionnaire to help her to express her opinions of the tools; this was followed by a brief interview. The subject was then thanked and released.

	Missed Location	Irrelevant Location	Total Time
Error Messages	54	19	666 seconds
Refactoring Annotations	23	4	876 seconds*

Table 9.5: The number and type of mistakes when diagnosing violations of refactoring preconditions, for each tool. The right-most column lists the total mean amount of time subjects spent diagnosing preconditions for all 8 refactorings. The asterisk (*) indicates that a timing was not obtained for one subject, so I could not include it in the mean. Subjects diagnosed errors in a total of 80 refactorings with each tool. Smaller numbers indicate better performance.

9.6.2.4 Results

Eight out of 10 subjects reported that Refactoring Annotations helped them understand violations better than error messages. The difference between subject ratings is statistically significant ($p = .046$, $df = 9$, $z = 2.00$)[2]. The measured results confirm this opinion; Table 9.5 shows the total number of program locations that subjects missed, as well as the number of irrelevant code fragments that subjects selected. The difference in missed locations was statistically significant ($p = .007$, $df = 9$, $z = 2.70$) as was the difference in choosing irrelevant locations ($p = .017$, $df = 9$, $z = 2.39$).

Six out of 10 subjects reported that they would be more likely to use Refactoring Annotations than error messages and 4 out of 10 said that they would be equally likely to use either. The difference between subject ratings is statistically significant ($p = .026$, $df = 9$, $z = 2.23$).

Nine out of 10 subjects reported that they felt that Refactoring Annotations helped them figure out what went wrong faster than with error messages. The difference between subject ratings is statistically significant ($p = .026$, $df = 9$, $z = 2.23$). The measured results confirm this opinion; subjects took on average 24% less time with Refactoring Annotations compared to error messages (Table 9.5), but the size of the effect depended on which tool the subject used first. When a subject used Refactoring

[2]This and the the remaining significance tests in this chapter were performed using a using a Wilcoxon matched-pairs signed-ranks test.

Annotations first and then used error messages, on average she took about 3% less time using Refactoring Annotations. When a subject used error messages first, on average she took about 41% less time.

Five out of 10 subjects reported that they felt that Refactoring Annotations made them more confident of their judgements than error messages, and 4 out of 10 said that they were equally confident with either tool. The difference between subject ratings is not statistically significant ($p = .084$, $df = 9$, $z = 1.73$). The one subject that said that she was less confident using Refactoring Annotations said that the reason was that "they give such great information, I feel like, 'Am I missing something?"'

The subjects' ability to identify the causes of violations, and the speed at which they could do it, varied from refactoring to refactoring:

- Refactoring numbers 1 and 7 (Table 9.4 on page 190), in which the programmer attempted to make a new program element that element conflicted with an existing program element, was usually understood with both error messages and Refactoring Annotations. However, subjects were sometimes unable to find the existing program element. With Refactoring Annotations, only one programmer did not correctly identify the conflicting program element, whereas with error messages, it was not identified 5 times. Moreover, using error messages, subjects incorrectly identified some irrelevant program element as conflicting 3 times. It appeared that this task was difficult because subjects generally performed linear searches through the given code, which not only took a significant amount of time, but often required repeated passes over the same code until they found the conflicting element. Several subjects mentioned that they would usually enlist the help of a find tool, such as Unix `grep`, to find candidates and then sort through those candidates manually to find the conflicting element. While useful, this technique would likely include false positives. Two subjects mentioned that they would use Eclipse's "Open Declaration" tool

to help in the task, but this would only be useful once they found a reference to the conflicting element in the code.

- Refactoring number 2 (Table 9.4 on page 190), in which an illegal identifier was typed that contained two dots, and refactoring number 6, where an illegal identifier was typed that contained a # or @ sign and began with a number, were sometimes problematic for subjects, especially those using error messages. With Refactoring Annotations, only once did a programmer select some irrelevant code when she thought there was a problem with the original code selection (she apparently saw a == sign, interpreted that as assignment, and thought that there was a violation similar to that shown in refactoring number 8). With error messages, 3 subjects noticed that the # or @ was a problem, but failed to also notice that the identifier started with a digit. Moreover, 2 subjects erroneously thought that Java identifiers could not contain *any* digits, and thus incorrectly said that digits inside of an identifier were a problem. One programmer said that she would use Google to find out whether # and @ were legal characters in Java identifiers. However, if typed literally, Google ignores these characters.

- Refactoring number 3 (Table 9.4 on page 190), in which INLINE METHOD was attempted on a method that contained two recursive calls and also implemented a method from an `Iterator`, was rarely answered perfectly. With both Refactoring Annotations and error messages, most subjects appeared to understand why recursion was a problem, but finding each recursive call was much more difficult with error messages. With error messages, 7 subjects missed at least one recursive call, but only 1 programmer missed them with Refactoring Annotations. As for the method implementing a method from an interface, no programmer using error messages noticed this problem. This appeared to be

because the refactoring tool didn't tell them about it; it told them only about a recursive call. When using Refactoring Annotations, five subjects noticed the problem with the interface, although only one of them could provide a coherent explanation as to why it was a problem.

- Refactoring number 4 (Table 9.4 on page 190), inlining an abstract method, appeared to be generally understood by all subjects, regardless of whether they used Refactoring Annotations or error messages. Subjects located the abstract method 9 out of 10 times using Refactoring Annotations, and 7 out of 10 times with error messages.

- Refactoring number 5 (Table 9.4 on page 190) was about inlining a method that contains a qualified `this` expression (such as `ClassName.this.member`). In it, subjects performed about equivalently with both Refactoring Annotations and error messages. With both tools, 7 out of 10 subjects were able to locate the problem, but apparently neither representation of the error was descriptive enough to help subjects understand why refactoring that code was a problem. This may be because the qualified `this` notation is relatively obscure. Only one programmer was able to explain the problem correctly.

- On refactoring number 8 (Table 9.4 on page 190), extracting a local variable from an expression containing an assignment, subjects performed about equivalently with both Refactoring Annotations and error messages. However, it appeared that neither representation helped any programmer to understand the cause of the problem.

When I interviewed subjects after the experiment, all seemed to prefer Refactoring Annotations, although to differing degrees. Subjects described them as "unobtrusive," "a little more helpful," "more informative," giving "more specific information," being "across the board helpful," "sav[ing me] some seconds," showing "more errors,

more like a compiler," showing "where the problem is," and " mak[ing] refactoring part of my usual error fixing strategy: read all, fix all at once." Subjects disliked error messages because they "cover too much area," "tend to get in the way," required the programmer to scan the code manually, and because "they are modal [and they say]: 'Tackle Me before You Do Anything Else!"'

9.6.2.5 Discussion

The results of the evaluation suggest that these generalized Refactoring Annotations are preferred by programmers, help them find the causes of precondition violations more accurately, and improve the speed at which they find those causes when compared to error messages. Because Refactoring Annotations were designed according to the guidelines discussed in Section 9.4, these results suggest that the guidelines can help improve the usability of refactoring tool precondition violations. As a consequence, implementing tools to represent violations for future refactoring tools according to those guidelines appears to be warranted. Materials from this experiment, including raw results, questionnaires, and my experimenter's notebook, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

For refactorings 5 and 8, Refactoring Annotations were not sufficiently detailed to help subjects understand why the tool could not perform the refactoring. It appears that a more detailed textual explanation of the problem might be helpful, including descriptions of language semantics. For instance, a help document might be useful for explaining what a qualified `this` statement is, and what the consequences would be if its containing method were inlined. However, the fact that programmers sometimes do not spend much time trying to understand a single sentence about a refactoring error (Section 4.5.3) suggests that they may not spend any time reading a detailed help document.

Reflecting upon the experiment, it appears that refactoring 5, which I originally

classified as a structure violation, might better be classified as a name unbinding violation. Why? A qualified `this` statement cannot be moved largely because it literally references a containing class, and that class reference will not be resolved at the method call sites. Thus, future replications of this experiment might show that subjects understand this violation better if it is represented as a name unbinding. This is because my guidelines suggest that name unbinding (unlike structure) should display what program element the refactored code is being unbound from; this information may help programmers better understand the violation.

9.6.2.6 Threats to Validity

While encouraging, there are several threats to validity for this experiment. First, I chose code by hand and modified it so that it would cause a violation of at least one refactoring precondition. The code may not be representative of code found in the wild. Second, subjects were volunteers from several classes at Portland State University, and likewise may not be representative of the average programmer. Third, the task given to subjects was on paper, and would likely vary somewhat if it were in a real development environment. Fourth, when placing sticky notes on code to indicate the location of violations, subjects may have simply been parroting what Refactoring Annotations (and to a lesser extent, error messages) were telling them, without any real understanding of the violation. To mitigate this threat, I had subjects briefly explain why they put a sticky note where they did, and tried to discern to what extent they understood the problem. Even if a programmer does not understand a violation, locating the pieces of code that are part of the problem is valuable because it focuses the programmer's attention on the relevant code.

9.7 Future Work

More study of the unknown and vague categories may reveal whether their members can be reclassified into other categories, or whether new categories emerge. Likewise, study of refactoring tools in other languages and paradigms may reveal new categories, or may simply reinforce existing categories.

A study of refactoring errors in the wild could determine which categories are encountered with the highest frequency by programmers. The results would help to determine which categories are deserving of further research. A deeper empirical evaluation of which precondition violations are most commonly conflated by programmers would also help to drive future research.

In addition to helping define how my guidelines apply to other kinds of error messages, the precondition taxonomy may also help design refactoring tools themselves. For example, while the clash category spans many refactorings, the guidelines dictate that a representation of a clash violation should show the relationship between the clashing program elements. Rather than a string describing the violation, a violation display routine in the development environment could take as arguments both of the program elements, and display their relationship, regardless of what refactoring generated the violation. This might help amortize the cost of implementing a graphical error system such as Refactoring Annotations.

Refactoring Annotations may also prove to be an enormous opportunity to help the programmer not just to understand precondition violations, but also to resolve them. This is because the graphics provide convenient "hooks" for implementing further functionality for highly-focused, detailed programmer interaction. For example, consider again Figure 9.6 on page 173. It may be useful to allow the programmer to interact with the two bottom arrows and messages to:

- open those elements in the existing editor, a new editor, or an embedded editor,

- make one or both of those members `public` so that they can be accessed in the destination class, or
- collect one or both of those members into the refactoring, so that they are moved along with the method.

In this way, specific graphical representations of precondition violations provide not only a way to enhance understanding of the precondition violations, but perhaps also a method of resolving them.

9.8 Conclusions

In this chapter, I have addressed the usability of tools that help programmers during the *error interpretation* step if the refactoring process (Section 2.5). I have presented guidelines and a tool called Refactoring Annotations, both designed to help programmers understand and recover from violated refactoring preconditions. The results of my evaluation suggest that Refactoring Annotations do indeed improve usability, in terms of speed, correct understanding, and programmer preference. Violations of refactoring preconditions continue to be difficult for programmers to understand, but the work presented in this chapter begins to aid that understanding.

Chapter 10

Conclusion

In this dissertation, I have presented data that suggests that programmers underuse refactoring tools, and that poor usability is one cause of that underuse. For five stages of the refactoring process, I have demonstrated how usability can be improved by way of usability guidelines, in the hope that this improvement will increase tool adoption and programmer efficiency.

10.1 Summary of Contributions

While I have laid out my specific contributions in each chapter of this dissertation, these contributions can be summarized as follows:

- The distinction between root-canal and floss refactoring (Section 2.3). An analysis shows that floss refactoring is the more popular refactoring tactic (Section 3.4.5). I outline what is necessary to build tools that are suitable for floss refactoring (Section 4.7.2).

- A study that confirms, and disconfirms, several previously held assumptions and conclusions about how programmers refactor in the wild (Section 3).

- Guidelines for building more usable refactoring tools in the future, based on current research about how programmers refactor (Chapters 5–9).

- Tools that show how those guidelines can be reified (Chapters 5–9), which is important because guidelines alone may be too ambiguous for toolsmiths to implement (Section 4.4).
- Evaluations of those tools, demonstrating improved usability (Chapters 5–9).

10.2 Limitations

Apart from the threats to validity discussed in each individual study, several additional limitations should be considered:

- First, while I have argued that underuse of refactoring tools is a significant problem, I have not provided direct evidence that usability will actually lead to increased adoption. Increased adoption can only be demonstrated over the long term, and, even if more usable tools see higher adoption and usage rates, other factors (such as marketing) may have confounding effects.
- Second, while I have addressed several steps of the refactoring process (Section 2.5), several other steps remain unaddressed. Specifically, I have not discussed how to improve usability for the following steps: Execute, Interpret results, Undo, Refactor (recursive), and Clean Up.
- Third, in improving the usability of refactoring tools, I have attempted to show improvements in efficiency, errors, and satisfaction, but have not addressed learnability or (with the exception of Section 7.5.2) memorability, two other key usability properties (Section 4.2).
- Fourth, while I have directly demonstrated improved usability of my user interfaces, I have shown only indirectly that the usability guidelines that informed their design are responsible. More detailed studies might apply each guideline individually in an attempt to explore how that guideline affects usability.

However, it was impractical to conduct such studies as part of this dissertation research.

10.3 Future Work

Several areas remain open for future work:

- Longer term study is needed of how improvements in the user interface affect refactoring and refactoring tool adoption.
- More study is needed outside of the Eclipse Java environment [18] because many of the results in this dissertation rely on data from programmers using only Eclipse. Although Eclipse for Java is a popular environment, future work should examine how the results might differ in other environments, especially non-object-oriented ones.
- A study should be performed on how programmers can be made aware that they are refactoring, and how a tool could help them in this task. This is another avenue to improve the adoption of refactoring tools.
- This research may be expanded outside of the context of refactoring tools. Although my research focused on the usability of refactoring tools, some of this work can be extended to other tools that employ static program analysis. For instance, my guidelines on usability of refactoring tool error messages may also apply to compilation errors.
- Future work on usability of refactoring tools could investigate not just usability *within* steps of my model, but usability *between* steps. A tool could make it easier to transition from step-to-step. For example, a smell detector might not only tell you about a smell, but also select the code that smells on your behalf, anticipating that you will want to refactor it with a tool.

- Future research must examine what other guidelines might be necessary to improve the usability of refactoring tools, in addition to the guidelines that I have presented in this dissertation. Indeed, I fully expect new guidelines to emerge as researchers learn more about how programmers refactor, and as programmers' behavior changes in the presence of new refactoring tools and programming paradigms.

10.4 The Thesis Statement

My thesis statement, as introduced in the first chapter, is:

> Applying a specified set of user-interface guidelines can help build more usable refactoring tools.

Throughout this dissertation, I have postulated these guidelines and applied them to several refactoring tool user interfaces. The guidelines and tools are listed in Table 10.1 on page 209. The evaluations suggest that the tools are indeed more usable, and as I have argued, this implies that the guidelines are responsible. Thus, generally speaking, I have supported my thesis.

However, the guidelines are numerous and varied, and the amount of confidence that you can have that each guideline will improve usability varies between them. For example, consider the Expressiveness guideline for the *Identify* step (Table 10.1 on page 209). Programmers directly rated this guideline *less* important than other guidelines, which somewhat reduces its relative importance in terms of inspiring improved usability. In contrast, consider the Task-centricity guideline, which was demonstrated as important to usability in four separate steps of the refactoring process: *Identify*, *Select*, *Initiate*, and *Configure*. The fact that it appeared repeatedly makes it more likely that it contributes to improved usability of refactoring tools. Having more support for some guidelines and less support for others has the positive

effect of helping toolsmiths prioritize which guidelines to follow in new refactoring tools, based on which ones are more likely to improve usability.

Programmers need usable refactoring tools: otherwise, they will not be used. This underuse is a missed opportunity; rather than harnessing the speed and correctness that refactoring tools afford, programmers who do not use tools may instead be slower and introduce more bugs. In this dissertation, I have presented guidelines to help improve usability, and provided evidence that the guidelines do so. I hope that, with these guidelines in hand, toolsmiths can build more usable refactoring tools that allow programmers to re-take this missed opportunity and enjoy the productivity benefits that were originally promised.

Step	Guideline	Tools
Identify	*Scalability*. A smell detector should not display smell information in such a way that a proliferation of code smells overloads the programmer. (p.65)	Stench Blossom (p.76)
Identify	*Relationality*. A smell detector should display smell information relationally when related code fragments give rise to smells. (p.65)	Stench Blossom (p.76)
Identify	*Bias*. A smell detector should place emphasis on smells that are more difficult to recognize without a tool. (p.65)	Stench Blossom (p.76)
Identify	*Task-centricity*. A smell detector should not distract from the programmer's primary task, if the need for refactoring is weak. (p.66)	Stench Blossom (p.76)
Identify	*Estimability*. A smell detector should help the programmer estimate the extent of the smell spread throughout the code. (p.66)	Stench Blossom (p.76)
Identify	*Availability*. A smell detector should make smell information as available as soon as possible, with little effort on the part of the programmer. (p.67)	Stench Blossom (p.76)
Identify	*Unobtrusiveness*. A smell detector should should not stop the programmer from programming while gathering, analyzing, and displaying information about smells. (p.68)	Stench Blossom (p.76)
Identify	*Context-Sensitivity*. A smell detector should first and foremost point out smells relevant to the current programming context. (p.69)	Stench Blossom (p.76)
Identify	*Expressiveness*. A smell detector should go further than simply telling the programmer that a smell exists; it should help the programmer find the source(s) of the problem by explaining *why* the smell exists. (p.69)	Stench Blossom (p.76)
Select	*Task-centricity*. A tool that assists in selection should not distract from the programmer's primary task. (p.102)	Selection Assist, Box View, Refactoring Cues (p.96 & 131)

Step	Guideline	Tools
Select	*Uniformity*. A tool that assists in selection should help the programmer to overcome unfamiliar or unusual code formatting. (p.102)	Selection Assist, Box View, Refactoring Cues (p.96 & 131)
Select	*Atomicity*. A tool that assists in selection should help eliminate as many selection errors as possible by separating character-based selection from program-element selection: the only possible selections should be those that are valid inputs to the refactoring tool. (p.103)	Selection Assist, Box View, Refactoring Cues (p.96 & 131)
Select	*Task-specificity*. A tool that assists in selection should be task specific. (p.103)	Selection Assist, Box View, Refactoring Cues (p.96 & 131)
Select	*Explicitness*. A refactoring tool should allow the programmer to explicitly see what the tool expects as input before selection begins. (p.106)	Refactoring Cues (p.131)
Select	*Multiplicity*. A tool that assists in selection should allow the programmer to select several program elements at once. (p.106)	Refactoring Cues (p.131)
Initiate	*Task-centricity*. Initiation of a refactoring tool should not distract from the programmer's primary task. (p.112)	Pie Menus for Refactoring (p.117)
Initiate	*Identifiability*. Initiation of a refactoring tool should not rely exclusively on the names of refactorings, but rather use a mechanism that more closely matches how programmers think about refactoring, such as structurally or spatially. (p.113)	Pie Menus for Refactoring (p.117)
Configure	*Bypassability*. Refactoring tool configuration should not force the programmer to view or enter unnecessary configuration information. (p.128)	Refactoring Cues (p.131)
Configure	*Task-centricity*. Refactoring tool configuration should not interrupt the programmer's primary task. (p.129)	Refactoring Cues (p.131)

Step	**Guideline**	**Tools**
Configure	*Accessibility*. Refactoring tool configuration should not obstruct a programmer's access to other tools, including the source code editor itself. (p.129)	Refactoring Cues (p.131)
Interpret Error	*Expressiveness*. Representations of refactoring errors should help the programmer to comprehend the problem quickly by clearly expressing the details: the programmer should not have to spend significant time understanding the cause of an error. (p.152)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Locatability*. Representations of refactoring errors should indicate the location(s) of the problem. (p.152)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Completeness*. Representations of refactoring errors should show all problems at once. (p.153)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Estimability*. Representations of refactoring errors should help the programmer estimate the amount of work required to fix violated preconditions. (p.153)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Relationality*. Representations of refactoring errors should display error information relationally, when appropriate. (p.153)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Perceptibility*. Representations of refactoring errors should allow programmers to easily distinguish precondition violations (showstoppers) from warnings and advisories. (p.153)	Refactoring Annotations (p.148 & 187)
Interpret Error	*Distinguishability*. Representations of refactoring errors should allow the programmer to easily distinguish between different types of violations. (p.154)	Refactoring Annotations (p.148 & 187)

Table 10.1: The guidelines postulated in this dissertation. **Step** indicates a step in the refactoring process (Section 2.5). **Guideline** states a postulated guideline and the page number where it was motivated. **Tools** lists my tools that implement that guideline and the page number where the tool was evaluated.

References

[1] Thorbjörn Ravn Andersen. Extract method: Error message should indicate offending variables. Bug Report, 4 2005. https://bugs.eclipse.org/89942. [cited at p. 152]

[2] Aptana. Ruby development tools, 2008. Computer Program, http://rubyeclipse.sourceforge.net. [cited at p. 157]

[3] Martin Bisanz. Pattern-based smell detection in TTCN-3 test suites. Master's thesis, Georg-August-Universität Göttingen, Dec 2006. [cited at p. 67]

[4] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM. [cited at p. 104]

[5] Fabrice Bourqun and Rudolf K. Keller. High-impact refactoring based on architecture violations. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and ReEngineering*, pages 149–158, Washington, DC, USA, 2007. IEEE Comp. Soc. [cited at p. 18, 33]

[6] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 95–100, New York, NY, USA, 1988. ACM. [cited at p. 112, 114, 115, 117]

[7] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In Danny Dig and Michael Cebulla, editors, *Proceedings of the 2nd Workshop on Refactoring Tools*, 2008. [cited at p. 50]

[8] Richard W. Conway and Thomas R. Wilcox. Design and implementation of a diagnostic compiler for PL/I. *Communications of the ACM*, 16(3):169–179, 1973. [cited at p. 161]

[9] Steve Counsell, Youssef Hassoun, Roger Johnson, Keith Mannock, and Emilia Mendes. Trends in Java code changes: the key to identification of refactorings? In *PPPJ '03: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 45–48, New York, NY, USA, 2003. Computer Science Press, Inc. [cited at p. 29]

[10] Steve Counsell, Youssef Hassoun, George Loizou, and Rajaa Najjar. Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 288–296, New York, NY, USA, 2006. ACM. [cited at p. 42]

[11] Brian de Alwis and Gail C. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 51–54, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 129]

[12] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Andrew P. Black, editor, *ECOOP*, Lecture Notes in Computer Science, pages 404–428. Springer, 2006. [cited at p. 42]

[13] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society. [cited at p. 61, 64, 69, 89]

[14] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994. [cited at p. 148]

[15] Dev Express. Refactor!Pro, 2008. Computer Program, http://www.devexpress.com/Products/NET/Refactor. [cited at p. 59]

[16] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12:369–388, 2002. [cited at p. 95, 147]

[17] Institute for Software. C++ refactoring for CDT, 2008. Computer Program, http://r2.ifs.hsr.ch/cdtrefactoring. [cited at p. 157]

[18] The Eclipse Foundation. Eclipse, 2008. Computer Program, http://www.eclipse.org. [cited at p. 7, 20, 50, 71, 94, 144, 204]

[19] The Eclipse Foundation. Eclipse C/C++ development tooling, 2008. Computer Program, http://www.eclipse.org/cdt. [cited at p. 157]

[20] The Eclipse Foundation. Eclipse Java development tools, 2008. Computer Program, http://www.eclipse.org/jdt. [cited at p. 156]

[21] The Squeak Foundation. Squeak, 2008. Computer Program, http://www.squeak.org. [cited at p. 7]

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [cited at p. 4, 6, 13, 18, 43, 60, 61, 62, 72, 113, 123]

[23] William G. Griswold. *Program restructuring as an aid to software maintenance*. PhD thesis, University of Washington, Seattle, WA, USA, 1992. [cited at p. 4, 155]

[24] Tovi Grossman, Pierre Dragicevic, and Ravin Balakrishnan. Strategies for accelerating on-line learning of hotkeys. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1591–1600, New York, NY, USA, 2007. ACM. [cited at p. 113]

[25] Code Guide. Omnicore, 2008. Computer Program, http://www.omnicore.com/en/codeguide.htm. [cited at p. 155]

[26] Shinpei Hayashi, Motoshi Saeki, and Masahito Kurihara. Supporting refactoring activities using histories of program modification. *IEICE - Transactions on Information and Systems*, E89-D(4):1403–1412, 2006. [cited at p. 67]

[27] T. Dean Hendrix, James H. Cross II, Saeed Maghsoodloo, and Matthew L. McKinney. Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java. In *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 382–386, New York, NY, USA, 2000. ACM. [cited at p. 147]

[28] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: A taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 99–108, New York, 2008. ACM. [cited at p. 29, 42]

[29] Jason I. Hong and James A. Landay. SATIN: A toolkit for informal ink-based applications. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 7, New York, NY, USA, 2006. ACM. [cited at p. 114]

[30] James J. Horning. What the compiler should tell the user. In *Compiler Construction, An Advanced Course, 2nd ed.*, pages 525–548, London, UK, 1976. Springer-Verlag. [cited at p. 161]

[31] Apple Inc. Xcode, 2008. Computer Program, http://developer.apple.com/tools/xcode. [cited at p. 7]

[32] JetBrains. IntelliJ IDEA, 2008. Computer Program, http://www.jetbrains.com/idea. [cited at p. 130]

[33] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996. [cited at p. 132, 135]

[34] William Joy and Mark Horton. An introduction to display editing with vi. University of California, Berkeley, 1984. [cited at p. 95, 96]

[35] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 576, Washington, DC, USA, 2002. IEEE Computer Society. [cited at p. 13]

[36] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 159–168, New York, NY, USA, 2005. ACM. [cited at p. 76]

[37] Gordon Kurtenbach and William Buxton. The limits of expert performance using hierarchic marking menus. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, pages 482–487, New York, NY, USA, 1993. ACM. [cited at p. 114, 116, 117]

[38] Martin Lippert. Towards a proper integration of large refactorings in agile software development. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, Proceedings of 5th International Conference (XP 2004)*, volume 3092 of *Lecture Notes in Computer Science*, pages 113–122, Garmisch-Partenkirchen, Germany, June 2004. Springer. [cited at p. 13]

[39] Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, pages 24–31, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 141]

[40] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. The Fabrik programming environment. In *IEEE Workshop on Visual Languages*, pages 222–230, Pittsburgh, PA, USA, Oct 1988. IEEE Computer Society Press. [cited at p. 114]

[41] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, 1994. [cited at p. 65]

[42] Jennifer Mankoff, Anind K. Dey, Gary Hsieh, Julie Kientz, Scott Lederer, and Morgan Ames. Heuristic evaluation of ambient displays. In *CHI '03: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 169–176, New York, NY, USA, 2003. ACM. [cited at p. 69]

[43] Mika V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 287–296, November 2005. [cited at p. 62, 65]

[44] Mika V. Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 297–306, New York, NY, USA, 2006. ACM. [cited at p. 37]

[45] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving usability of software refactoring tools. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*, pages 307–318, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 13, 48, 49, 68, 87]

[46] Microsoft. Visual Studio, 2008. Computer Program, http://msdn.microsoft.com/vstudio. [cited at p. 7]

[47] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006. [cited at p. xv, 20, 37, 39, 113, 116]

[48] Emerson Murphy-Hill. Improving refactoring with alternate program views. Technical Report TR-06-086, Portland State University, Portland, OR, 2006. http://multiview.cs.pdx.edu/publications/rpe.pdf. [cited at p. 50, 55, 100]

[49] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, NY, USA, 2008. ACM. [cited at p. 46, 93, 143]

[50] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), September-October 2008. [cited at p. 3, 29, 46]

[51] Emerson Murphy-Hill and Andrew P. Black. Seven habits of a highly effective smell detector. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 36–40, New York, NY, USA, 2008. ACM. [cited at p. 60]

[52] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009. [cited at p. 18]

[53] Netbeans. Sun Microsystems, Inc., 2008. Computer Program, http://www.netbeans.org/. [cited at p. 27]

[54] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. [cited at p. 47]

[55] Jakob Nielsen. Ten usability heuristics. Internet, 2005. http://www.useit.com/papers/heuristic/heuristic_list.html. [cited at p. 47, 80, 94, 106, 113, 154]

[56] Bruce Oberg and David Notkin. Error reporting with graduated color. *IEEE Software*, 9(6):33–38, November 1992. [cited at p. 161]

[57] Alexis O'Connor, Macneil Shonle, and William Griswold. Star diagram with automated refactorings for Eclipse. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, New York, NY, USA, 2005. ACM. [cited at p. 103]

[58] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992. [cited at p. xi, 49, 51, 62, 154, 155]

[59] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA '90: Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990. [cited at p. 3]

[60] Chris Parnin and Carsten Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 76]

[61] Chris Parnin and Carsten Görg. Design guidelines for ambient software visualization in the workplace. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 18–25, June 2007. [cited at p. 70]

[62] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 2008 ACM Symposium on Software Visualization*, 2008. [cited at p. xv, 64, 69]

[63] Markus Pizka. Straightening spaghetti-code with refactoring? In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 846–852. CSREA Press, 2004. [cited at p. 7, 33]

[64] Stuart Pook, Eric Lecolinet, Guy Vaysseix, and Emmanuel Barillot. Control menus: excecution and control in a single interactor. In *CHI '00: Extended Abstracts on Human Factors in Computing Systems*, pages 263–264, New York, NY, USA, 2000. ACM. [cited at p. 125]

[65] Jef Raskin. *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. [cited at p. 47, 66, 88, 129]

[66] Jacek Ratzinger. *sPACE: Software Project Assessment in the Course of Evolution*. PhD thesis, Vienna University of Technology, Austria, 2007. [cited at p. 29, 30, 31, 42]

[67] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 35–38, New York, 2008. ACM. [cited at p. 29, 42]

[68] Romain Robbes. Mining a change-based software repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 15–23, Washington, DC, USA, 2007. IEEE Comp. Soc. [cited at p. 21]

[69] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. [cited at p. 49]

[70] Ben Shneiderman. *System Message Design: Guidelines and Experimental Results*, chapter 3, pages 55–78. Human/Computer Interaction. Ablex Publishing Corporation, 1982. [cited at p. 67, 69, 80, 154]

[71] Ben Shneiderman. *Designing the User Interface (2nd ed.): Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. [cited at p. 47, 48, 68, 103, 106, 113, 129]

[72] James Shore. Design debt. *Software Profitability Newsletter*, February 2004. http://jamesshore.com/Articles/Business/Software Profitability Newsletter/Design Debt.html. [cited at p. 6]

[73] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Main-*

tenance and Reengineering, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society. [cited at p. 64, 69]

[74] Stephan Slinger. Code smell detection in Eclipse. Master's thesis, Delft University of Technology, March 2005. [cited at p. 67]

[75] Omnicore Software. X-develop, 2008. Computer Program, http://www.omnicore.com/xdevelop.htm. [cited at p. 27, 58, 130]

[76] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*, pages 1–6, Washington, DC, USA, 2007. IEEE Computing Society. [cited at p. 29, 42]

[77] Linda Tetzlaff and David R. Schwartz. The use of guidelines in interface design. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 329–333, New York, NY, USA, 1991. ACM. [cited at p. 48]

[78] The Eclipse Foundation. Usage Data Collector Results, August 29th, 2008. Website, http://www.eclipse.org/org/usagedata/reports/data/commands.csv. [cited at p. 21]

[79] Simon Thompson, Claus Reinke, Huiqing Li, Chris Brown, Jonathan Cowie, and Nguyen Viet Chau. Hare: The Haskell refactorer, 2008. Computer Program, http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html. [cited at p. 157]

[80] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering Methodology*, 10(1):5–55, 2001. [cited at p. 161]

[81] Mark A. Toleman and Jim Welsh. Systematic evaluation of design choices for software development tools. *Software - Concepts and Tools*, 19(3):109–121, 1998. [cited at p. 22]

[82] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *CSMR*, pages 329–331. IEEE Computing Society, 2008. [cited at p. 67]

[83] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM. [cited at p. 156]

[84] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, New York, NY, USA, 2006. ACM. [cited at p. 19, 22, 42, 43]

[85] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported — an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 18, 35, 42, 43]

[86] Xref-Tech. Xrefactory, 2008. Computer Program, http://www.xref.sk. [cited at p. 155]

[87] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86, 2000. [cited at p. 161]