

Tools for a Successful Refactoring

Emerson Murphy-Hill and Andrew P. Black

Portland State University

1900 SW 4th Avenue

Portland, OR 97201

+1-503-725-4036

{emerson, black}@cs.pdx.edu

Abstract

Although refactoring tools have been integrated into numerous development environments over the past ten years, we have seen little variation in the human interfaces to refactoring tools – when a refactoring fails, most tools present a textual error message. Existing interfaces can cause programmers to refactor slowly, conservatively, and in an error-prone manner. In this demonstration, we will show how our tools can help programmers overcome barriers to a successful refactoring.

Categories and Subject Descriptor

D.2.3 [Software Engineering]: Coding Tools and Techniques;

D.2.6 [Software Engineering]: Programming Environments.

General Terms Design, Reliability, Human Factors

Keywords Refactoring, tools, usability, environments

1. Introduction

Refactoring is the process of restructuring code without changing the way a program behaves. Because preserving behavior can be difficult when restructuring by hand, Roberts and colleagues proposed a tool that refactors semi-automatically [1]. To perform a refactoring, the programmer selects the program element(s) to be refactored, supplies some parameters (such as a new identifier), and applies the refactoring. If successful, the programmer may review the changes; if unsuccessful, the refactoring tool usually gives a brief explanation.

Since this first refactoring tool, very little has changed. Although nearly every professional development environment provides refactoring tools, each environment interfaces with the programmer in the same way, plus or minus a few dialog boxes.

As a case study, we have investigated how programmers perform the Extract Method refactoring [2]; we found the this human interface is inadequate.

2. Code Selection

The first major problem that we observed during refactoring is that programmers have difficulty in selecting program statements using the keyboard or mouse. Statements can be difficult to select in code when it is not formatted to the programmer’s personal preference (often, in code the programmer did not write). Even if code is well-formatted, very long statements, such as long if statements, can be difficult to select because the beginning and the

end of the statement may not be on the same screen.

Selection is not a problem unique to the Extract Method refactoring – indeed, all refactorings require the identification of some program construct, and selection appears to be the most direct way of doing so. Existing techniques, such as various forms of parenthesis matching, are inadequate because parenthesized expressions are not the only meaningful selections. Other techniques, such as Eclipse’s hotkeys for statement selection, force the programmer to use the keyboard and remember the right key combination.

To address the difficulty of statement selection, we have built two new code selection tools for Java in Eclipse. These tools are available at <http://multiview.cs.pdx.edu/refactoring>.

2.1 SelectionAssist

The SelectionAssist tool gives the programmer a visual cue to the extent of a statement. When a programmer places the cursor in front of a statement, that statement is overlaid with a highlight. The highlight remains visible while the programmer selects the statement. The next statement is highlighted when the programmer extends the selection into that statement. In this way, programmers can select a list of statements appropriate for the Extract Method refactoring.

```
boolean chistoso(int x){
    if (getA() < 10)
        if (getB() < 10) {
            return false;
        }
    return x > 30;
}
```

Figure 1. SelectionAssist

SelectionAssist exhibits some desirable attributes of a selection tool. It doesn’t slow down the common case or change the way a statement is physically selected; it merely provides a cue. The tool is easy to learn because it is similar to existing tools, such as Eclipse’s “Mark Occurrences” highlighter.

2.2 BoxView

The BoxView tool provides an alternate view of program code that simply shows how statements are ordered and nested. BoxView represents each statement as a clickable rectangle. Since statements are nested in program code, boxes are nested in BoxView. When a programmer clicks on a rectangle in BoxView, the corresponding statement is selected in the program code. When a programmer selects part of a statement in the code, the

corresponding rectangle is shown in orange in BoxView. When a programmer selects a whole statement in the code, the corresponding rectangle is shown in cyan in BoxView. By using the keyboard, a programmer can use BoxView just like Eclipse's hotkeys for statement selection.

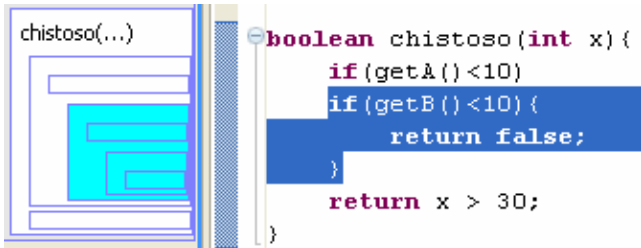


Figure 2. BoxView (at left)

BoxView exhibits several desirable attributes. BoxView exposes the underlying structure of code, regardless of code formatting. It allows the programmer to select statements at a constant rate, regardless of the size of the statement. Like SelectionAssist, BoxView allows the programmer to use either the keyboard or the mouse.

3. Error Recovery

An often overlooked aspect of the refactoring process is error recovery. Refactoring tools will sometimes present error messages when a refactoring precondition is violated [3]. For example, one precondition of the Extract Method refactoring is that you cannot have a return statement inside of a conditional as part of the code to be extracted. To perform the refactoring and recover from this error successfully, the programmer must first perform another refactoring to change the program logic to remove the conditional return.

However, informing the programmer what the problem is and where it is located can be problematic. Eclipse, like most other refactoring tools, gives the user a vague and cryptic error message. For example, Eclipse might say, "Ambiguous return value: selected block contains more than one assignment to local variable." We have demonstrated that programmers have problems understanding and recovering from these messages [2]. The problem of poor representation of violated preconditions goes beyond the Extract Method refactoring; indeed, all refactorings have preconditions that must be expressed by refactoring tools.

3.1 Refactoring Annotations

To address the issue of how to express violations of refactoring preconditions so that the programmer can recover, we have created Refactoring Annotations. Refactoring Annotations are meant to be used before a refactoring is applied, so that the programmer gets an idea of what will happen, and will be informed of the extent of any violated preconditions. Refactoring Annotations are overlaid on program code, indicating the location and cause of violated preconditions graphically. In Figure 3, Refactoring Annotations indicate that two variables will be passed

into the extracted method and two variables must be returned – a precondition violation for Extract Method. Refactoring Annotations currently express precondition violations for Extract Method, although the techniques they use can be employed for other refactorings.

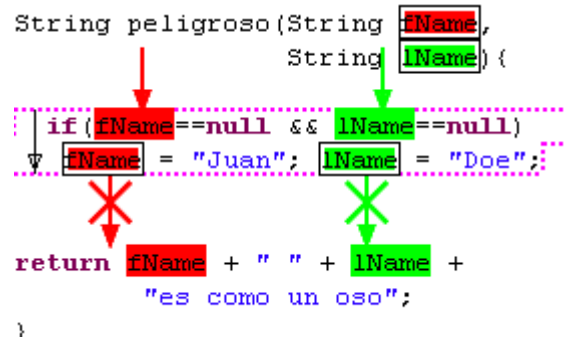


Figure 3. Refactoring Annotations

Refactoring Annotations are an improvement over typical refactoring error messages for a number of reasons. They indicate the position(s) of the violations, reducing the time to locate and fix the problem. Unlike Eclipse's error messages, they indicate every violated precondition. The programmer can easily distinguish violations from one another, unlike with textual error messages, which are easily conflated and misinterpreted.

4. Conclusion

We have discussed the motivation and functionality of several tools intended to help programmers successfully refactor. Our experience with a small set of programmers in an experiment shows that these tools can increase the speed and accuracy at which programmers can perform refactoring tasks [2]. While the tools were built to solve problems encountered during the Extract Method refactoring, we believe that similar problems will be encountered during other refactorings and that the techniques embodied in the tools presented here will be helpful in building other refactoring tools in the future.

Acknowledgements

Our thanks go to Mark P. Jones, the study participants, and the National Science Foundation for funding this research.

References

- [1] Roberts, D., Brant, J. and Johnson, R. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3, 4, 1997, pp. 253-263.
- [2] Murphy-Hill, Emerson. *Improving Refactoring with Alternate Program Views*. Research Proficiency Exam, Portland State University, Portland, OR, 2006.
- [3] Opdyke, W. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.