

# Improving Usability of Refactoring Tools

Emerson Murphy-Hill  
Portland State University  
1900 SW 4th Avenue  
Portland, OR 97201  
+1-503-725-4036  
emerson@cs.pdx.edu

## Abstract

While the integration of refactoring tools into many development environments has increased, the usability of these tools has remained stagnant. Specifically, when refactorings fail, tools communicate the failure to the programmer poorly, causing the programmer to restructure slowly, conservatively, and without preserving behavior. In the accompanying poster, I show how refactoring correctness, speed, and user satisfaction can be measurably increased by improving the usability of refactoring tools. In the long run, better usability will aid in the adoption and utilization of refactoring tools.

### 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 a process of restructuring code without changing the way it behaves [1]. Refactoring can be semi-automated with the help of tools, such as those that are integrated into the Eclipse environment. If these tools help programmers refactor with fewer errors, in less time, and with greater overall satisfaction than refactoring by hand, then programmers are likely to adopt such tools.

In the accompanying poster, I take the Extract Method refactoring as a case-study. Extract Method is an important refactoring [2], widely implemented in many environments, and almost always implemented as a wizard-and-error-message-like interface. Because of its non-trivial implementation and user-interface similarity to other refactoring tools, I believe the lessons learned from Extract Method tools also apply to other refactoring tools.

## 2. Problems with Current Tools

As an exercise, I observed 11 graduate and commercial programmers perform the Extract Method refactoring on whatever code they thought appropriate in several large codebases. Each programmer successfully extracted between 2 and 16 methods in approximately 30 minutes using the Eclipse environment. I observed two main difficulties encountered by subjects while trying to refactor.

First, on many occasions I observed that programmers were unable to select a list of statements, due to very long statements and inconsistent formatting. Second, I observed that programmers have trouble understanding and interpreting error messages produced by the tools, such as “Ambiguous return value: selected block contains more than one assignment to local variable.” These error messages are generally caused by violated refactoring preconditions.

## 3. New Refactoring Tools

I have created three tools for Eclipse that address the selection problems and error message problems described in the last section. The curious reader can find further information about these tools, view animated demos, and download prototypes at <http://multiview.cs.pdx.edu/refactoring>.

### 3.1 SelectionAssist

SelectionAssist provides the programmer with a visual cue to the extent of a statement, to assist in accurate statement selection. When the cursor is placed in the whitespace in front of a statement, a green highlight appears over the text range of that statement (Figure 1).

```
int limonada(int p){
  while (p<10)
    p += getA();
  return p;
}
```

Figure 1. SelectionAssist.

### 3.2 BoxView

BoxView is as a series of nested boxes adjacent to the program code (Figure 2). When a rectangle is clicked, the corresponding statement is selected. When a statement is selected, the corresponding rectangle is shown in cyan.

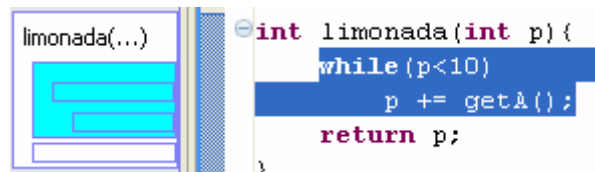


Figure 2. BoxView (at left).

### 3.3 Refactoring Annotations

Refactoring Annotations tell the programmer why a code segment cannot be extracted by overlaying program code with representations of violated refactoring preconditions. In Figure 3,

the programmer has selected and wishes to extract two statements. Refactoring Annotations tell the programmer that one parameter must be passed in to the extracted method, and that two values must be returned, violating an Extract Method precondition. Refactoring Annotations can also inform the programmer about precondition violations involving control flow.

```
double nada (int a) {
    double b = 1/a; int c = getA();
    return a*b;
}
```

Figure 3. Refactoring Annotations.

#### 4. Experimental Validation

To demonstrate that the three tools presented in the last section are more usable than existing tools, I performed two experiments. The experiments were performed with 16 subjects drawn from an object-oriented programming class. Subjects were college seniors, graduate students, and commercial developers, all at least moderately familiar with either Java or C/C++.

The experiments were intended to simulate the two problematic stages of refactoring described in Section 2.

##### 4.1 Selection Experiment

In the first experiment, I simulated the selection of statements during the Extract Method refactoring by having subjects select a large number of if statements. Subjects used the mouse/keyboard, SelectionAssist, and BoxView, with varied order among subjects.

For all subjects combined, Table 1 indicates that subjects mis-selected (e.g., missed a curly bracket) fewer statements with SelectionAssist and BoxView than with the mouse and keyboard, and were also faster with SelectionAssist and BoxView. Subjects’ expressed a preference for both BoxView and SelectionAssist over the keyboard/mouse, for the tasks given.

Table 1. Results of selection experiment.

	Mis-Selected	Correctly Selected	Mean Selection time
Mouse/Keyboard	36	303	10.4 seconds
SelectionAssist	6	355	5.5 seconds
BoxView	2	357	7.8 seconds

##### 4.2 Extraction Experiment

In the second experiment, I simulated the task of encountering and recovering from violated preconditions by having subjects identify the location(s) of violated preconditions in several pieces of code to be extracted. Subjects used the Eclipse Extract Method Wizard and Refactoring Annotations to help diagnose the violation.

In Table 2, the numbers for “Failed Identification” count the cases where the subject was unable to identify the reasons that a refactoring failed. The numbers for “Incorrect Identification” count the cases where the subject incorrectly identified a piece of code as violating a precondition. The results show that subjects identified the reasons for refactoring failures more accurately and more rapidly using Refactoring Annotations than with the Eclipse Extract Method Wizard. Eleven of the 16 subjects found Refactoring Annotations more helpful than the Eclipse Wizard, while the remaining 5 subjects said both tools were equally helpful. All participants said that they would like to use Refactoring Annotations again.

Table 2. Results of extraction experiment.

	Failed Identification	Incorrect Identification	Mean Identification Time
Eclipse Wizard	11	28	164 seconds
Refactoring Annotations	1	6	46 seconds

#### 5. Conclusion and Future Work

Using the Extract Method refactoring as a case-study, I have demonstrated usability problems in refactoring tools, proposed solutions to those problems, and shown that these solutions improve programmers’ speed and correctness.

In the future, I would like to explore how the visual techniques used in Refactoring Annotations extend to other refactorings. Furthermore, it would be interesting to investigate how these techniques apply to other software engineering tools, such as compilers. For example, type error messages are notoriously hard for beginners to understand and for experts to track down [3]. Can annotations help improve understanding of type errors?

Research has generated many apparently useful software tools. Ultimately, though, a tool will not be adopted if it is not usable. This research represents a small step towards increasing the usability of software engineering tools.

#### Acknowledgements

My thanks go to my advisor, Andrew P. Black, Mark P. Jones, the study participants, and the National Science Foundation for funding this research.

#### References

- [1] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [2] Fowler, M. Crossing Refactoring’s Rubicon. <http://www.martinfowler.com/articles/refactoringRubicon.html>, 2001.
- [3] Yang, J., Wells, J., Trinder, P., Michaelson, G. Improved Type Error Reporting. *Proceedings of the 12<sup>th</sup> International Workshop on Implementation of Functional Languages*, 2000, pp. 71-86.