# Breaking the Barriers to Successful Refactoring

Emerson Murphy-Hill and Andrew P. Black
Portland State University, P.O. Box 751
Portland, OR 97201-0751
{emerson,black}@cs.pdx.edu

## ABSTRACT
Refactoring, the process of changing the structure of code without changing its behavior, can be semi-automated with the help of tools. However, many tools do a poor job of communicating errors triggered by the refactoring process. This poor communication causes programmers to refactor slowly, conservatively, and incorrectly. In this paper we demonstrate problems with current refactoring tools, characterize three new tools to assist in refactoring, and describe a user study that compares these new tools against existing tools. The results of the study show that the speed, accuracy, and user satisfaction can be significantly increased The new tools have inspired a set of usability recommendations that we hope will help build a new generation of programmer-friendly refactoring tools.

## Categories and Subject Descriptors
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 changing the structure of code without changing the way the program behaves [7]. Many activities fall under the heading of refactoring: changing variable names, moving methods or fields up and down a class hierarchy, substituting one algorithm for another, and removing dead code, to name a few. Refactoring is important to software development because it aids in program understanding and makes it easier to add new features; thus, refactoring helps programmers to adapt their software to changing requirements.

But performing a refactoring is not trivial, even for seemingly simple refactorings such as changing variable names. After changing a variable name, you must be sure to change every reference to the new name, but not when the name appears in string literals, in the middle of other variable names, or in comments (unless the comment directly refers to the variable, except when in casual use), and not when the name is shadowed by a variable of the same name in a subclass, or by a local variable of the same name. Moreover, even apparently simple refactoring operations have complex rules, or preconditions, that must be satisfied before we can be sure that a refactoring is safe to apply.


*ICSE'08*, May 10–18, 2008, Leipzig, Germany.


For several refactorings, Opdyke showed that program behavior is preserved when certain preconditions are satisfied [17]. Later, Roberts and colleagues developed the first tool that automatically checks preconditions before refactoring [20], automating this error-prone and time-consuming task. Roberts' thesis describes his experience building and using the Refactoring Browser, the original refactoring tool [19]. Although Roberts extolled the virtues of using refactoring tools, he noted that the original tool was so unpopular that the designers did not even use it themselves. Upon reflection, Roberts noted three usability recommendations that every good refactoring tool should have: speed, undo support, and tight IDE integration.

Most tools appear to have implemented Roberts' usability recommendations; our review of 16 refactoring tools shows very little variation from the Refactoring Browser's user interface. However, despite their prevalence in modern development environments, programmers do not use refactoring tools as often as they should [15]. Why not? What usability problems do modern refactoring tools have that we can observe empirically? In addition to Roberts' three usability recommendations, what further guidelines will help improve the adoption and usage rates of refactoring tools? To answer these questions, we decided to start by studying a non-trivial refactoring.

### 1.1 Extract Method Refactoring
One refactoring that has enjoyed widespread tool support is called Extract Method [9]. A tool that performs the Extract Method refactoring essentially takes a sequence of statements, copies them into a new method, and then replaces the original statements with a call to the new method. This refactoring is useful when duplicated code should be factored out and when a method contains code segments that are conceptually separate.

In his influential book on refactoring, Fowler reports that Extract Method is one of the most common refactorings that he performs [7]. Later, in the article "Crossing Refactoring's Rubicon," Fowler says that Extract Method is "a key refactoring. If you can do Extract Method, it probably means you can go on [to] more refactorings" [6]. Because the Eclipse environment [4] implements Extract Method and because little user-interface variation exists between refactoring tools, we reason that Eclipse is a representative, non-trivial refactoring tool worthy of study.

While Extract Method tools have become widespread, the human interface to such tools remains stagnant. To use refactoring tool, the programmer first selects code to be refactored, then configures

```
boolean canRideToday(){
        boolean tiresOk = !tires.areFlat();
        boolean spouseOk = !spouse.isUpset();
        return tiresOk && spouseOk;
}
```

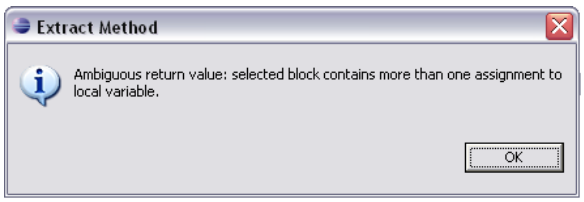


**Figure 1. A code selection (above, in grey) that a tool cannot extract into a new method.**

the refactoring via a "refactoring wizard" or dialog box, then presses "OK" to execute the refactoring. The browser then presents the user with a generic textual error message if there is a problem. Figure 1 displays an example of such an error message in Eclipse. In this paper we demonstrate that user-interface changes to refactoring tools can both reduce the number of errors encountered by programmers and improve the programmers' ability to understand the remaining errors.

## 1.2 An Formative Study in Refactoring

In our experience, error messages emitted by existing tools' are non-specific and unhelpful in diagnosing problems. We decided to undertake a formative study to determine how often these messages arise in practice and whether other programmers also find them unhelpful.

We observed 11 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.

We asked the participants to use the Eclipse Extract Method Wizard to refactor parts of several large, open-source projects:

- *Azureus*, a peer-to-peer file-sharing client [3];
- *GanttProject*, a project scheduling application [22];
- *JasperReports*, a report generation library [10];
- *Jython*, a Java implementation of the Python programming language [9];
- The *Java 1.4.2 libraries* [21].

We picked these projects because of their size and maturity, not because they were particularly in need of refactoring.

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 are usually good candidates for refactoring. However, the programmers decided on which projects to run the long-method tool, and which candidates to refactor. Each session with a programmer was limited to 30 minutes, and programmers successfully extracted between 2 and 16 methods during that time.

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.

**Figure 2. Preconditions to the Extract Method refactoring, based on Opdyke's preconditions [17]. We have omitted preconditions that were not encountered during the refactoring exercise.**

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 some of 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.
- Error messages regarding syntactic selection occurred about as frequently as any other type of error message (violating precondition 0, Figure 2). 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 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, Figure 2).
- 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 our observations of programmers struggling with refactoring error messages, we conjecture as follows:

- Error messages were insufficiently descriptive. Especially among programmers who had not used refactoring tools previously, a new error message may not be understandable. When asked to explain what an error message was saying and where the problem was located, several programmers gave explanations unrelated to the problem.
- Error messages were misinterpreted. The errors were all 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 1), the next time a programmer came across any assignments to local variables, the programmer didn't try to refactor, even if no preconditions were violated.

This study revealed two types of improvements to Extract Method tools. First, to prevent a large number of errors in the first place, programmers need support in making a valid selection. Second, to help programmers successfully recover from violated preconditions, programmers need expressive, distinguishable, and understandable feedback that conveys the meaning of precondition violations.

# 2. NEW TOOLS FOR EXTRACT METHOD

In the following section, we describe three tools[1] that we have built for the Eclipse environment that address the problems demonstrated in the formative study. Although built for the Java programming language, the techniques embodied in these tools apply to other object-oriented and imperative programming languages.

## 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 (Figure 3). Using the green highlight as a guide, a programmer can then select the statement normally with the mouse or keyboard.

This tool bears similarities to tools found in other development environments. Dr. Scheme, for example, highlights the area between two parentheses in a similar manner [5], although that highlighting disappears whenever cursor selection begins, making it inappropriate as a selection cue. Vi and other text editors have mechanisms for bracket matching [11], but brackets do not surround most statements, so these tools are not always useful for selecting statements. Some environments, such as Eclipse, have special keyboard commands to select statements, but during this project, nearly every programmer under observation seemed to prefer the mouse. Selection Assist allows the programmer to use either the mouse or the keyboard for selection tasks, accommodating both varieties of programmer.

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

**Figure 3. The Selection Assist tool in the Eclipse environment, shown covering the entire `if` statement, in green. The user's selection is partially overlaid, darker.**

## 2.2 Box View

We designed a second tool to assist with selection, called Box View, that displays nested statements as a series of nested boxes. Box View is a window shown adjacent to program text that displays a uniform representation of the code (Figure 4). At the top level, 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.

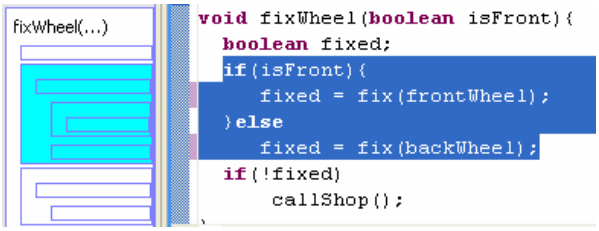


**Figure 4. Box View tool in the Eclipse environment, to the left of the program code.**

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 scales fairly well as the level of statement nesting increases. In methods with less than 10 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.

Box View was inspired by a similar tool in Adobe GoLive [1] that displays an outline of an HTML table.

[1] The tools and a short screencast are available at: http://www.multiview.cs.pdx.edu/refactoring.

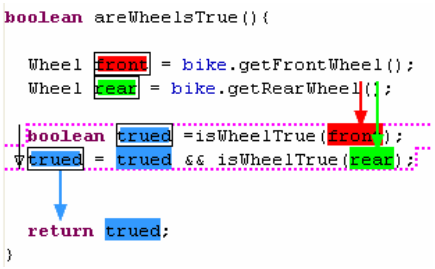


**Figure 5. Refactoring Annotations overlaid on program code. The programmer has selected two lines (between the dotted lines) to extract. Here, Refactoring Annotations show variable use: `front` and `rear` will be parameters, and `trued` will be returned.**

## 2.3 Refactoring Annotations

Refactoring Annotations communicate to the programmer control- and data-flow for the Extract Method refactoring. Annotations overlay program text to express information about a specific extraction (Figure 5). Each variable is assigned a distinct color, and each occurrence is highlighted. 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. When the selection contains assignments to more than one variable, multiple arrows are drawn from the bottom showing multiple return values (Figure 6, 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 6, middle). When the selection contains a branch statement, a line is drawn from the branch statement to its corresponding target (Figure 6, bottom). In each case, Xs are displayed over the arrows, indicating the location of the offending code.

When code does not meet a precondition, Refactoring Annotations are intended to give the programmer an idea of how to correct the violation. Although refactoring while violating a precondition may change program behavior, 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, another kind of refactoring.

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

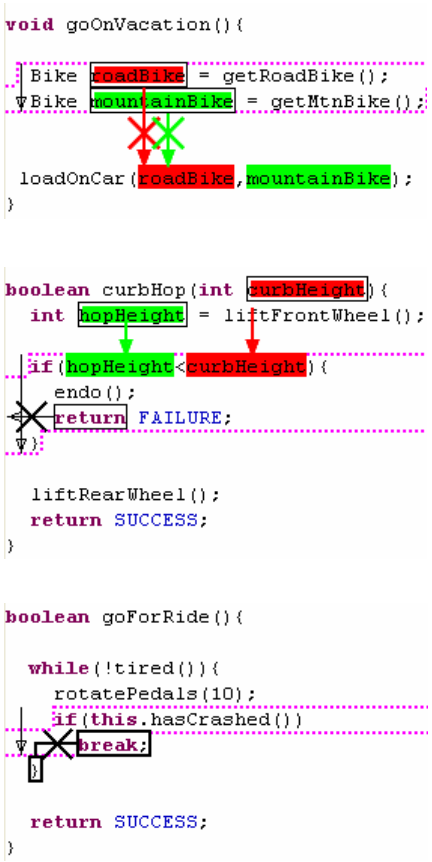


**Figure 6. Refactoring Annotations display an instance of a violation of refactoring precondition 1 (`goOnVacation`), precondition 2 (`curbHop`), and precondition 3 (`goForRide`), described in Figure 2.**

or assigns many variables, the annotations become visually complex. However, we reason that this is desirable: the more variables that are passed in or returned, the less cohesive the extracted method. Thus, we 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 finding these solutions in two ways. Firstly, because Refactoring Annotations can indicate multiple precondition violations simultaneously, the annotations give the programmer an idea of the severity of the problem. Correcting for a conditional return alone will be easier than correcting for a conditional return, and a branch, and multiple assignments. Likewise, correcting two assignments is likely easier than correcting six assignments. Secondly, Refactoring Annotations give specific, spatial cues to problem points to help the programmer diagnose the violated preconditions accurately.

**Table 1. Total number of correctly selected and mis-selected `if` statements and mean correct selection time, with time normalized to mouse/keyboard selection time, over all subjects for each tool.**

| | Total Mis-Selected If Statements | Total Correctly Selected If Statements | Mean Selection Time | Selection time as Percentage of Mouse/Keyboard Selection Time |
|---|---|---|---|---|
| **Mouse/Keyboard** | 37 | 303 | 10.2 seconds | 100% |
| **Selection Assist** | 6 | 355 | 5.5 seconds | 54% |
| **Box View** | 2 | 357 | 7.8 seconds | 76% |

Refactoring Annotations were inspired by a variety of prior ideas. Our control flow annotations are visually similar to Control Structure Diagrams [8]. However, unlike Control Structure Diagrams, Refactoring Annotations depend on the programmer's selection, and include less noise. Variable highlighting is much 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. Furthermore, variables are highlighted automatically, when they are used both inside and outside of the selection. In Refactoring Annotations, the arrows drawn on parameters and return values are similar to the arrows drawn in the Dr. Scheme environment [5], which draws arrows between a variable declaration and each variable reference. Unlike the arrows in Dr. Scheme, Refactoring Annotations draw only one arrow per parameter and per return value, as needed.

# 3. USER STUDY

Having demonstrated that there are usability problems with Extract Method tools and having proposed new tools as solutions, we conducted a study that has helped to ascertain whether the new tools overcome these usability problems. The study has two parts. In the first part, programmers used the mouse and keyboard, Selection Assist, and Box View to select program statements. In the second part, programmers used the standard Eclipse Extract Method Wizard and Refactoring Annotations to identify problems in a selection that violated Extract Method preconditions. In both parts, we evaluated their responses for speed and correctness.

## 3.1 Human Subjects

We 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 [11]. All students were at least somewhat familiar with the practice of refactoring.

## 3.2 Experiment Design

The experiments were performed over the period of a week, and lasted between ½ and 1½ hours per subject. The subjects first used three selection tools: mouse and keyboard, Selection Assist, and Box View (the "selection experiment"), then later the Eclipse Extract Method Wizard and Refactoring Annotations (the "precondition experiment"). For the selection experiment, subjects were randomly assigned to one of five blocks; a different random code presentation and tool usage order was used for each block. For the precondition experiment, subjects were randomly assigned to one of two blocks; a different random code presentation order was used for each block. In both experiments, we selected code from the open source projects described in Section 1.2. Each subject used every tool.

When a subject began the selection experiment, the test administrator showed her how to use one of the three selection tools, depending on which block she was assigned to. The administrator demonstrated the tool for about a minute, told the subject that her task was to select all **`if`** statements in a method, and allowed her to practice the task using the selection tool until she was satisfied that she could complete the task (usually less than 3 minutes). The subject then was told to perform the task in 3 different methods from different classes, about two dozen **`if`** statements total. This experiment was then repeated for the two other tools on two different code sets.

After the selection experiment was complete, the subject performed the precondition experiment. The test administrator first showed the programmer how the Extract Method refactoring works using the standard Eclipse refactoring tool, the Eclipse Extract Method Wizard. The administrator then demonstrated and explained each error message produced by the Eclipse Wizard for preconditions 1 through 3 in Figure 2, lasting about 5 minutes. The subject was then told her task was to identify each and every violated precondition in a given code selection, assisted by the tool's diagnostic error message. The subject was then allowed to practice using the tool until she was satisfied that she could complete the task (usually less than 5 minutes). The subject was then told to perform the task on 4 different Extract Method candidates from different classes. The experiment was then repeated for Refactoring Annotations on a different code base.

# 4. RESULTS OF THE STUDY

Here we present the results of the study, including measurements of the accuracy in completing the tasks, the time taken to complete a task, and subjects' perceptions of the tools[2].

## 4.1 Measured Results

Table 1 shows the combined number of **`if`** statements that subjects selected correctly and incorrectly for each tool. Table 1 also shows the mean time in seconds to select an **`if`** statement

[2] Preliminary results were presented in an extended abstract at the 2007 ACM Student Research Competition [14].

across all participants, and the time normalized as a percentage of the selection time for the mouse and keyboard.

From Table 1, we can see that there were far more mis-selections using the mouse and keyboard than using Selection Assist, and that Box View had the fewest mis-selections. Table 1 also indicates that Selection Assist improved selection speed by 46%, and that Box View improved selection speed by 24%. Both speed increases are statistically significant ($p < 0.001$, using a t-test with a logarithmic transform to normalize long selection-time outliers).

The top graph in Figure 7 shows individual subjects' mean times for selecting **`if`** statements using the mouse and keyboard against Selection Assist. Here we can see that all subjects but one (labeled 'a') were faster using the Selection Assist than using the mouse and keyboard (subjects below the dotted line). We 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 was statistically significant ($p < 0.01$, using a Wilcoxon signed ranks test).

The bottom graph in Figure 7 compares the mouse and keyboard against Box View. Here we see that 11 of the 16 subjects are faster using Box View than using the mouse and keyboard. We can also see that all subjects except one (labeled 'c') are less error prone with Box View. The error-rate difference was statistically significant ($p < 0.01$, using a Wilcoxon signed ranks test).

Table 2 shows two kinds of problems that subjects encountered during the Extract Method task. "Missed Violation" means that a subject failed to recognize that one or more preconditions were being violated. "Irrelevant Code" means that a subject marked some piece of code that was irrelevant to the violated precondition, such as marking a **`break`** statement when the problem was a conditional **`return`**.

Table 2 tells us that programmers made fewer mistakes with Refactoring Annotations than with the Eclipse Wizard. Using Refactoring Annotations, subjects were much more likely to recognize all precondition violations and identify the assigned variable in the selection. Subjects were also much less likely to misidentify the precondition violations. The difference in error-rate was statistically significant ($p < 0.01$, using a Wilcoxon signed ranks test).

Table 2 also shows the mean time to find all precondition violations correctly, across all participants. On average, subjects recognized precondition violations more than three times faster using Refactoring Annotations than using the Eclipse Wizard. The recognition time improvement was statistically significant ($p < 0.001$ using a t-test with a logarithmic transform to remedy long recognition time outliers).

Figure 8 shows the mean time to identify all precondition violations correctly for each tool and each user. Note that we omitted two participants from the plot, because they did not correctly identify precondition violations for any code using the Eclipse Wizard. Again, note that the dotted line represents equal mean speed using either tool. In Figure 8, we notice that all users are faster with Refactoring Annotations. We also notice that most users were more accurate using Refactoring Annotations.

In all, 45 out of 64 uses of Refactoring Annotations helped the subjects to mark every precondition violation. Only 26 out of 64 uses of the Eclipse Wizard allowed the subjects to identify every precondition violation.

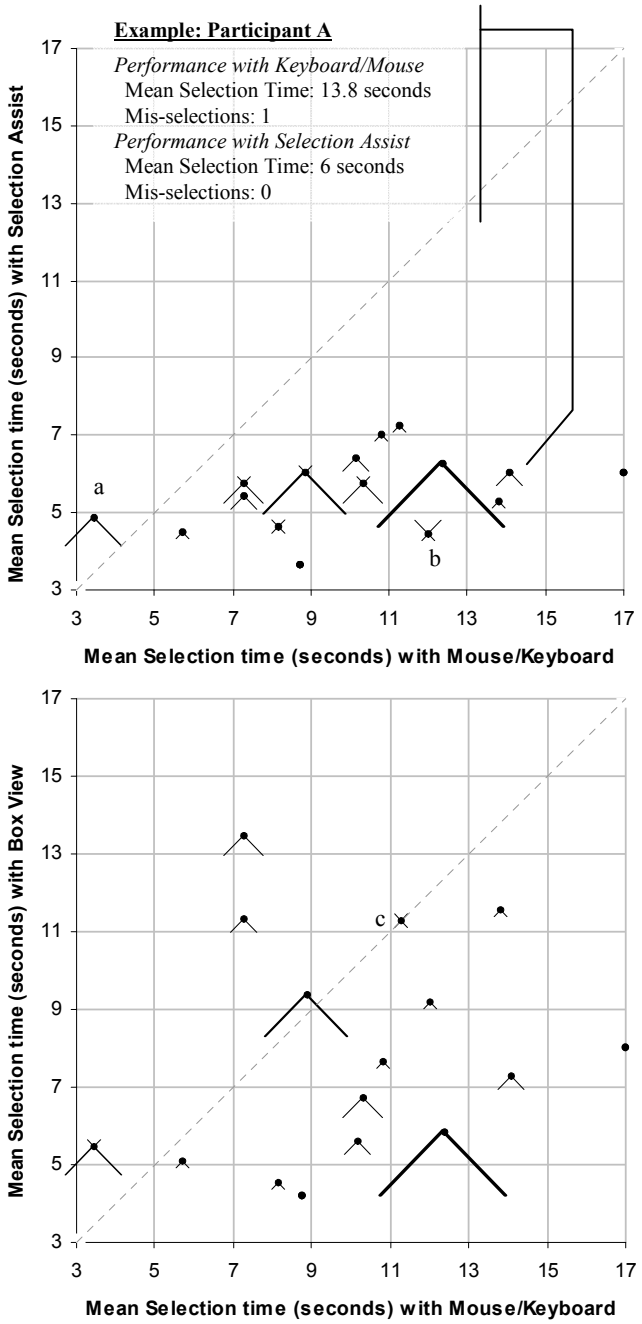


**Figure 7. Mean time in seconds to select `if` statements using the mouse and keyboard versus Selection Assist (top) and Box View (bottom). Each subject is represented as a whole or partial X. The distance between the bottom legs represents the number of mis-selections using the mouse and keyboard. The distance between the top arms represents the number of mis-selections using Selection Assist (top) or Box View (bottom). Points without arms or legs represent subjects who did not make mistakes with either tool.**

**Table 2. At left, number and type of mistakes when finding problems during the Extract Method refactoring over all subjects, for each tool. At right, the mean time to correctly identify all violated preconditions, in seconds. Smaller numbers indicate better performance.**

| | Missed Violation | Irrelevant Code |
|---|---|---|
| **Eclipse Wizard** | 11 | 28 |
| **Refactoring Annotations** | 1 | 6 |

| Mean Identification Time |
|---|
| 164 seconds |
| 46 seconds |

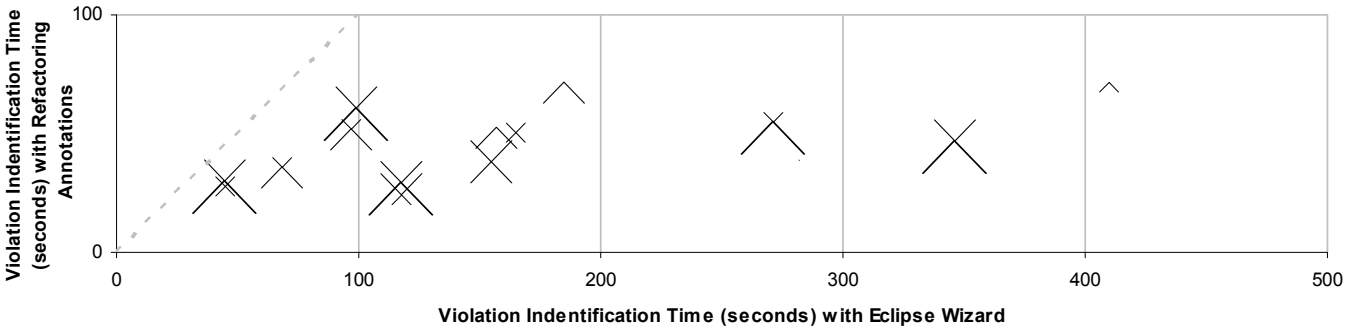


**Figure 8. For each subject, mean time to identify precondition violations correctly using the Eclipse Wizard versus Refactoring Annotations. Each subject is represented as an X, where the distance between the bottom legs represents the number of imperfect identifications using the Eclipse Wizard and the distance between the top arms represents the number of imperfect identifications using Refactoring Annotations.**

Overall, compared against traditional tools, subjects performed better in terms of speed and accuracy for all three tools that we have created: Selection Assist, Box View, and Refactoring Annotations.

## 4.2 Questionnaire Results

We administered a post-test questionnaire that allowed the subjects to express their preferences for the five tools they tried. The survey itself and a summary of the responses can be found in our technical report [13]. Significance levels are reported with $p < 0.01$, using a Wilcoxon 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 or Selection Assist. The difference preferences for both Box View and Selection Assist over the keyboard and mouse were statistically significant. 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 they were likely to use it again.

Subjects were unanimously positive on the helpfulness of Refactoring Annotations and all subjects said they were likely to use them again, while the reviews of standard Eclipse Extract Method Wizard were mixed. Differences in helpfulness and likeliness to use again were both statistically significant. Concerning the standard Eclipse Extract Method Wizard, 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 message the Eclipse tool produced, one subject quipped, "who reads alert boxes?"

Overall, the subjects' responses showed that they found the Selection Assist, Box View, and Refactoring Annotations 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 of the context of the study.

## 4.3 Limitations of Findings

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

In the selection experiment, each subject used every tool on each code set. Unfortunately, a flaw in the design of our study caused the distribution of tools to code sets to be uneven. In the most extreme instance, one code set was traversed only twice with the mouse and keyboard while another code set was traversed eight times using the Selection Assist. However, because each code set was chosen to be of roughly equal content and difficulty, we do not believe this biased the results in favor of any particular tool.

In the precondition diagnosis experiment, every subject first used the Eclipse Extract Method Wizard then used Refactoring Annotations. We originally reasoned that the fixed order was necessary to educate programmers about how Extract Method is performed because our 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, we should have made more of an effort to vary the tool usage order. However, we believe that the magnitude of the differences of errors and speed, coupled with the strong subject preference, suggest that Refactoring Annotations are preferable to refactoring error dialog boxes.

In both experiments, we tested the core of selection and precondition recognition tasks, but one must consider their context in a real-world programming situation. For example, while Box View is more accurate than Selection Assist, Box View takes up more screen real estate and requires switching between the editor and a separate view, 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. We 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.

# 5. DISCUSSION

During this study, we have observed that new tools can improve programmer accuracy and speed in refactoring.

An effective statement selection tool is critical to a successful Extract Method refactoring. Programmers can use both Box View and Selection Assist to improve code selection. 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 required and statements are less than a few lines long.

Refactoring Annotations are preferable to an error-message-based approach for showing precondition violations during the Extract Method refactoring. The results of this study indicate that Refactoring Annotations communicate the precondition violations effectively. When a programmer has a better understanding of refactoring problems, we believe the programmer is likely to be able to correct the problems and successfully perform the refactoring.

## 5.1 Recommendations for Future Tools

The tools described in this paper 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, we reason that the interaction techniques embodied in these tools are applicable to all refactoring tools. Every refactoring tool requires the programmer to select a piece of code to be refactored and every refactoring tool requires the programmer to interpret the meaning of a violated precondition. By studying how programmers use existing refactoring tools and the new tools that we have described in this paper, we have deduced a number of desirable general properties for all refactoring tools.

Tools that assist in the selection of code should:

- Be lightweight: users can normally select code quickly and efficiently, and any tool to assist selection should not add overhead to slow down the common case.
- Help the programmer overcome unfamiliar or unusual code formatting.
- Allow the programmer to select code in a manner specific to the task they are performing. While bracket matching can be helpful, bracketed statements are not the only meaningful program construct that a programmer needs to select.

Tools that display violations of refactoring preconditions should:

- Be lightweight: the full, round-trip time to complete a tool-assisted refactoring should not take longer than a manual refactoring.
- Indicate the location(s) of precondition violations. A tool should tell the programmer what it just discovered, rather than needing "to basically compile the whole snippet in my head," as one Eclipse bug reporter complained regarding an Extract Method error message [2].
- Show all violated preconditions at once. This helps the programmer in assessing the severity of the violations.
- Help programmers distinguish precondition violations (showstoppers) from warnings and advisories. Programmers should not have to wonder whether there is a problem with the refactoring.
- Give some indication of the amount of work required to fix the problem. The programmer should be able to tell whether a violation means that the code can be refactored with a few minor changes, or that the refactoring is nearly hopeless.
- Display the violation relationally, when appropriate. Violations are often not caused at a single character position, but arise from a number of related pieces of source code. Relations can be represented using arrows and colors, for example.
- Use different, distinguishable representations for different types of violations. Programmers should not confuse one error for another and waste time tracking down and trying to fix a violation that does not exist.

While these recommendations may seem self-evident, they are rarely implemented in contemporary refactoring tools.

# 6. RELATED WORK

Many tools provide support for the Extract Method refactoring, but few deviate from the wizard-and-error-message interface described in Section 1.2. 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 [15]. 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 [23]. Again, this tool makes strong assumptions about what the programmer wants.

O'Connor and colleagues implement Extract Method using a graph notation to help the programmer recognize and eliminate code duplication [18], but they do not specify what happens when a precondition is violated. This approach avoids selection mistakes by presenting program structure as an abstract syntax tree, where nodes are the only valid selections.

Mealy and colleagues [12] have compiled a list of 38 usability guidelines for building refactoring tools. Unlike our research, the authors' guidelines are derived theoretically by refining existing guidelines and using general human-computer interaction models. While their goal is to build tools that support all of the refactoring process, our goal is to empirically find and remedy usability

deficiencies in existing refactoring tools to make them more palatable to the end-programmer.

## 7. FUTURE WORK

In the future, we plan on generalizing our selection tools and Refactoring Annotations. While we have shown that these tools are useful for one particular refactoring, they are only worth programmers' time to learn if they are applicable in all refactorings. We are currently investigating how Box View can be made applicable to all refactorings and overlaid on code like Selection Assist. We will also be using techniques similar to Refactoring Annotations to communicate violations of preconditions for other refactorings.

After generalizing our tools to other refactorings, we should be able to cross-validate our recommendations for future tools. For instance, it will be useful to determine what other violated preconditions should be displayed relationally. We expect that new recommendations will emerge as well.

We also plan to expand our recommendations by addressing other stages of the programmers' refactoring process. For example, we plan on investigating how to improve the process of configuring refactorings.

Finally, we would like to evaluate our tools in a larger case study. Our small experiments are useful in evaluating some aspects of our tools, but a long-term case study can help us evaluate how programmers' behavior changes with more usable tools. In the long term, we hope more usable tools foster increased adoption and use.

## 8. CONCLUSIONS

In this paper, we have presented three tools that help programmers avoid selection errors and understand refactoring precondition violations.

With Selection Assist and Box View, we were able to reduce code selection errors several fold. Likewise, with Refactoring Annotations, we were able to improve the refactoring precondition diagnosis by several fold. For each of our new refactoring tools, speed and user satisfaction was significantly increased. We were surprised to see that such simple improvements to existing refactoring tools yielded such dramatic usability improvements.

However, the contribution of this research is not the tools themselves, but the qualities embodied in the tools that produce the demonstrated benefits. Therefore, to increase the usability of new refactoring tools, we have distilled our observations into a set of usability recommendations. We hope that builders of future refactoring tools will heed our recommendations and build tools that help programmers refactor quickly, errorlessly, and pleasurably.

## 9. ACKNOWLEDGMENTS

For their reviews and advice, we would like to thank Barry Anderson, Robert Bauer, Paul Berry, Iavor Diatchki, Tom Harke, Brian Huffman, Mark Jones, Jim Larson, Chuan-kai Lin, Ralph London, Philip Quitslund, Suresh Singh, Tim Sheard, and Aravind Subhash. Special thanks to participants in the user study and our anonymous reviewers for detailed, insightful criticism. Also, thanks to the National Science Foundation for partially funding this research under grant CCF-0520346.

## 10. REFERENCES

[1] Adobe Systems Incorporated. 2007. Adobe GoLive. http://www.adobe.com/products/golive.

[2] Andersen, T.R. 2005. "Extract Method: Error Message Should Indicate Offending Variables." https://bugs.eclipse.org/bugs/show_bug.cgi?id=89942.

[3] Azureus Incorporated. 2005. Azureus. http://azureus.sourceforge.net.

[4] The Eclipse Foundation. 2007. Eclipse. http://eclipse.org.

[5] Findler, R., Clements, J., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. 2002. "DrScheme: A Programming Environment for Scheme." Journal of Functional Programming, vol. 12, pp. 159-182.

[6] Fowler, M. 2001. "Crossing Refactoring's Rubicon," http://martinfowler.com/articles/refactoringRubicon.html.

[7] Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc.

[8] Hendrix, T. D., Cross, J. H., Maghsoodloo, S., and McKinney, M. L. 2000. Do visualizations improve program comprehensibility? experiments with control structure diagrams for Java. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education.* (Austin, Texas, United States, March 07 - 12, 2000). ACM Press, New York, NY, 382-386.

[9] Hugunin, J. and Warsaw, B. 2005. Jython, http://www.jython.org.

[10] JasperSoft Corporation. 2005. JasperReports. http://jasperreports.sourceforge.net.

[11] Joy, W. and Horton, M. 1984. "An Introduction to Display Editing with Vi."

[12] Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. 2007. Improving Usability of Software Refactoring Tools. In *Proceedings of the 2007 Australian Software Engineering Conference* (April 10 - 13, 2007). ASWEC. IEEE Computer Society, Washington, DC, 307-318.

[13] Murphy-Hill, E. 2006. Improving Refactoring with Alternate Program Views. Research Proficiency Exam, TR-06-086, Portland State University, http://multiview.cs.pdx.edu/publications/rpe.pdf, Portland, OR.

[14] Murphy-Hill, E. "Improving Usability of Refactoring Tools." ACM Student Research Competition. http://www.acm.org/src/subpages/murphy-hill/acm_src_final.html

[15] Murphy-Hill, E. and Black, A. 2007. Why don't people use refactoring tools? In *Proceedings of the 1st Workshop on Refactoring Tools*. ECOOP '07. TU Berlin, ISSN 1436-9915.

[16] Omnicore Software. 2007. CodeGuide. http://www.omnicore.com.

[17] Opdyke, W. F. 1992. *Refactoring Object-Oriented Frameworks*. Technical Report. UMI Order Number: UIUCDCS-R-92-1759., University of Illinois at Urbana-Champaign.

[18] O'Connor, A., Shonle, M., and Griswold, W. 2005. Star diagram with automated refactorings for Eclipse. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange* (San Diego, California, October 16 - 17, 2005). ETX '05. ACM Press, New York, NY, 16-20.

[19] Roberts, D. B. 1999 *Practical Analysis for Refactoring*. Technical Report. UMI Order Number: UIUCDCS-R-99-2092., University of Illinois at Urbana-Champaign.

[20] Roberts, D., Brant, J., and Johnson, R. 1997. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems* 3, 4 (October 1997), 253–263.

[21] Sun Microsystems Incorporated. 2005. Java 1.4.2 Standard Libraries, http://java.sun.com/j2se/1.4.2/.

[22] Thomas, A. and Bareshev, D. 2005. GanttProject, http://ganttproject.sourceforge.net.

[23] Xref-Tech. 2005. Xrefactory, http://www.xref-tech.com.