Improving Refactoring with Alternate Program Views

Emerson Murphy-Hill

Portland State University, Department of Computer Science
P.O. Box 751, Portland, OR 97207-0751
emerson@cs.pdx.edu

Abstract. Refactoring is the process of changing the structure of code without changing its behavior. Refactoring can be semi-automated with the help of tools, but many existing tools do a poor job of communicating errors triggered by the programmer. This poor communication causes programmers to refactor slowly, conservatively, and incorrectly. In this paper, I demonstrate the problems with current refactoring tools, characterize three new alternative program views to assist in refactoring, and describe a user study that compares these new views against existing tools. The results of the study show that both the speed and accuracy of refactoring can be increased using these new views. The new views exhibit several desirable properties for future refactoring tools.

1 Introduction

This paper describes how tools support programmers in a broad swath of day-to-day programming activities called refactoring. In this section, I will define what refactoring is, describe why it is important, and detail an essential refactoring called Extract Method. I will depict how tools are meant to assist in refactoring and describe an exercise that demonstrates the deficiencies of current tools.

1.1 Refactoring

Refactoring is simply defined as the process of changing the structure of code without changing the way a program behaves [1]. Many activities fall under the heading of refactoring: changing variable names, moving class members up and down a class hierarchy, changing visibility, generalizing and specializing types, substituting one algorithm for another, and removing dead code, to name a few.

Refactoring is important for several reasons:

- **Refactoring makes understanding programs easier.** Well-chosen names and meaningful module boundaries make code more digestible for the programmer. For instance, a variable named "hasHelmet" is likely to be more understandable than a variable named "b." Programmers rarely get names right the first time, and refactoring allows names to be changed as the program evolves.

- **Refactoring makes adding new features easier.** Adding new functionality may be difficult, depending on how a program is structured. For instance, printing to the console every time a variable is set is difficult when the variable is set in many places. If the program is refactored so that all sets to that variable go through a single setter method, then printing to the console would only require the addition of one line.
- **Refactoring keeps development agile.** Software customers are famous for changing requirements, so software developers must adapt. Programmers cannot predict what will change, nor should they have to. Refactoring allows programs to be changed throughout the development cycle.

But the power of refactoring does not come for free. Performing 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. Even apparently simple refactoring operations have many rules, or preconditions, that must be satisfied.

For several refactorings, Opdyke showed that program behavior is preserved when certain preconditions are satisfied [2]. Later, Roberts and colleagues developed the first tool that automatically checks preconditions before refactoring [3]. Today, refactoring tools are implemented in most mainstream integrated development environments for object-oriented languages, including Smalltalk, C++, Java, Python, and Visual Basic. Refactoring tools have also been built for other, non-object-oriented languages, such as Haskell [4] and Prolog [5].

1.2 Extract Method Refactoring

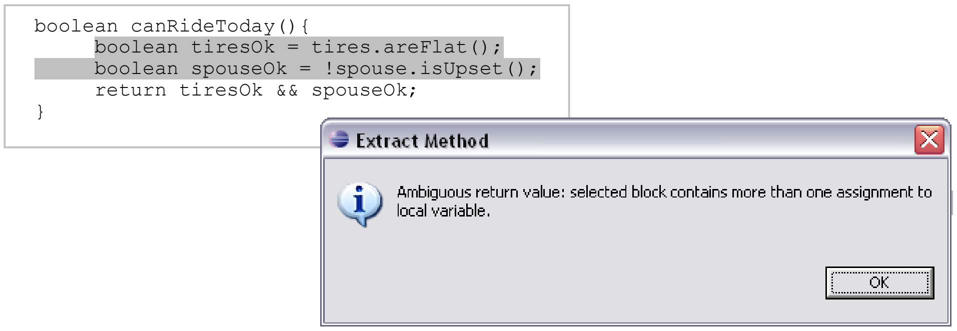
One refactoring that has enjoyed widespread tool support is called Extract Method [1]. A tool that performs the Extract Method refactoring essentially takes a sequence of statements, copies them into a new method, and then replaces the original statements with a call to the new method. This refactoring is useful when duplicated code should be factored out (Figure 1) 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 he performs [1]. Later, in the article "Crossing Refactoring's Rubicon," Fowler says Extract Method is "a key refactoring. If you can do Extract Method, it probably means you can go on [to] more refactorings" [6].

```
void hopOverLog(){
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
     rotatePedals(0);//stop pedaling
     liftRearWheel();
}

void showOff(){
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
     rotatePedals(50);//ease up a bit
     changeExpression(Faces.GRIN);
}
```

```
void hopOverLog(){
     popWheelie();
     rotatePedals(0);//stop pedaling
     liftRearWheel();
}

void showOff(){
     popWheelie();
     rotatePedals(50);//ease up a bit
     changeExpression(Faces.GRIN);
}

void popWheelie() {
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
}
```

Figure 1. An application of the Extract Method refactoring to remove duplicated code in a Bicyclist class. On the left, the original code, on the right, the refactored code.



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

Extract Method

Ambiguous return value: selected block contains more than one assignment to local variable.

Figure 2. A code selection (in grey) that a tool cannot extract into a new method. Most tools that perform this refactoring will display an error message similar to the one shown above.

While Extract Method tools are important, the human interface to such tools remains stagnant. The original Refactoring Browser for Smalltalk assists the user in performing Extract Method by prompting the user for a new name for the method. The browser then presents the user with a generic textual error message if there is a problem [3]. Figure 2 displays an example of such an error message in the Eclipse environment [7]. My review of 16 tools that perform the Extract Method refactoring shows very little variation on this user interface. In this paper, I demonstrate that this user interface can be significantly improved.

1.3 An Exercise in Refactoring

From personal experience, I found that existing tools' presentations of error messages are typically non-specific and unhelpful in diagnosing problems. However, I was unsure how often these problems arise in practice and whether other programmers also find the error messages unhelpful.

As an exercise, I 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, while three were commercial software developers. I asked the programmers to perform Extract Method refactorings wherever they thought it was appropriate, using the Eclipse Extract Method Wizard. Each session with a programmer lasted around 30 minutes, and programmers successfully extracted between 2 and 16 methods.

I asked the participants to refactor several code bases:

- Azureus, a peer-to-peer file-sharing client [8];
- GanttProject, a project scheduling application [9];
- JasperReports, a report generation library [10];
- Jython, a Java implementation of the Python programming language [11];
- the Java 1.4.2 libraries [12].

Each code base is large, active, and mature. Each code base uses different coding conventions and varies in the degree to which those conventions are adhered to. I gave programmers a tool to find unusually large methods, an indicator that Extract Method might be useful.

The exercise 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.

These observations suggest that (1) programmers fairly frequently attempt to apply Extract Method to code that is not immediately extractable, and (2) some experienced programmers have developed coping strategies to avoid code that cannot easily be extracted.

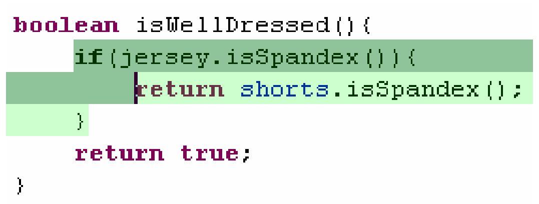
The errors that programmers encountered while trying to apply Extract Method arose from violations of preconditions. Opdyke defined several preconditions to the

0. The selection 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.
4. Outside of the selection, there must be no branches to code inside of the selection. Java has no such language feature, but GOTO would be an example from other languages.
5. The code must compile before it is refactored.

Figure 3. Preconditions to the Extract Method refactoring, based on Opdyke's preconditions [2]. I have omitted preconditions regarding the naming, visibility, and inheritance relationships of the new, extracted method.

Extract Method refactoring [2], summarized in Figure 3. During the exercise, I observed the following about the error messages that the tools presented:

- Error messages regarding the syntactic selection occurred about as frequently as any other type of error message (violating precondition 0, Figure 3). 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 flowed down multiple screens.
- The tool reported only the first violation that it found. With varying degrees of success, programmers scanned the source code to find other violations of preconditions.
- Programmers never selected **break** or **continue** statements without their corresponding targets. Generally, programmers were quite conservative about this type of irregular control flow, and thus avoided refactoring code with **break** or **continue** statements.
- The error messages often discouraged the programmer 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, the next time a programmer came across any assignments to local variables, the programmer didn't try the extraction at all.
- The errors were sometimes misinterpreted. The error messages were all presented as text boxes and sometimes with similar wording. At times, programmers interpreted one error message as an unrelated error message.
- The errors were sometimes insufficiently descriptive. Especially among programmers who previously had not used refactoring tools, 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.



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

Figure 4. The Selection Assist tool in the Eclipse environment, shown covering the entire **if** statement, in green. The user's mouse selection is partially overlaid, slightly darker.

This exercise revealed that there are two types of improvements to Extract Method tools. First, programmers need support in making a valid selection before the Extract Method refactoring can take place. Second, programmers need more expressive, distinguishable, and understandable messages that convey the meaning of precondition violations.

2 New Views for Extract Method

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

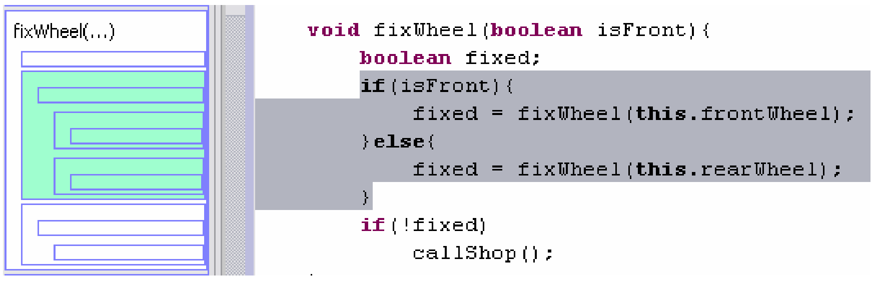
I have built Selection Assist and Box View to improve on the standard mouse and keyboard for selecting code. I have also built Refactoring Annotations to improve on the standard dialog-based Extract Method tools.

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 4). 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 those found in other development environments. Dr. Scheme, for example, highlights the area between two parentheses in a similar manner [13]. EMACS and other text editors have similar mechanisms for bracket matching [14]. However, brackets do not surround most program statements, so these tools are not very 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. The Selection Assist allows the programmer to use either the mouse or the keyboard for selection tasks.

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



```
void fixWheel(boolean isFront){
    boolean fixed;
    if(isFront){
        fixed = fixWheel(this.frontWheel);
    }else{
        fixed = fixWheel(this.rearWheel);
    }
    if(!fixed)
        callShop();
```

Figure 5. Box View tool in the Eclipse environment, to the left of the program code.

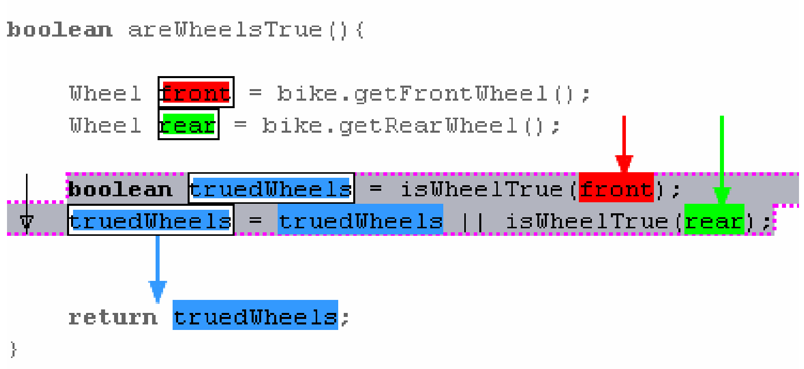
2.2 Box View

Another tool that assists with selection is Box View, which displays a simplified abstract syntax tree 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 5). At the top level, Box View represents a class as a box with labeled methods 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 shows up in orange. When the programmer selects a whole statement in the editor, the corresponding box shows up in green. When the programmer selects a box, Box View selects the corresponding program statement in the program code.

Box View was inspired by a similar tool in Adobe GoLive [15] that displays an outline of an HTML table. 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 green). Using the keyboard, the programmer can select sibling, parent and child statements.

2.3 Refactoring Annotations

Refactoring Annotations convey the consequences of an Extract Method refactoring. Annotations overlay program text to express information about a specific Extract Method refactoring (Figure 6). 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 a parameter 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.



```
boolean areWheelsTrue(){

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

    boolean truedWheels = isWheelTrue(front);
    truedWheels = truedWheels || isWheelTrue(rear);

    return truedWheels;
}
```

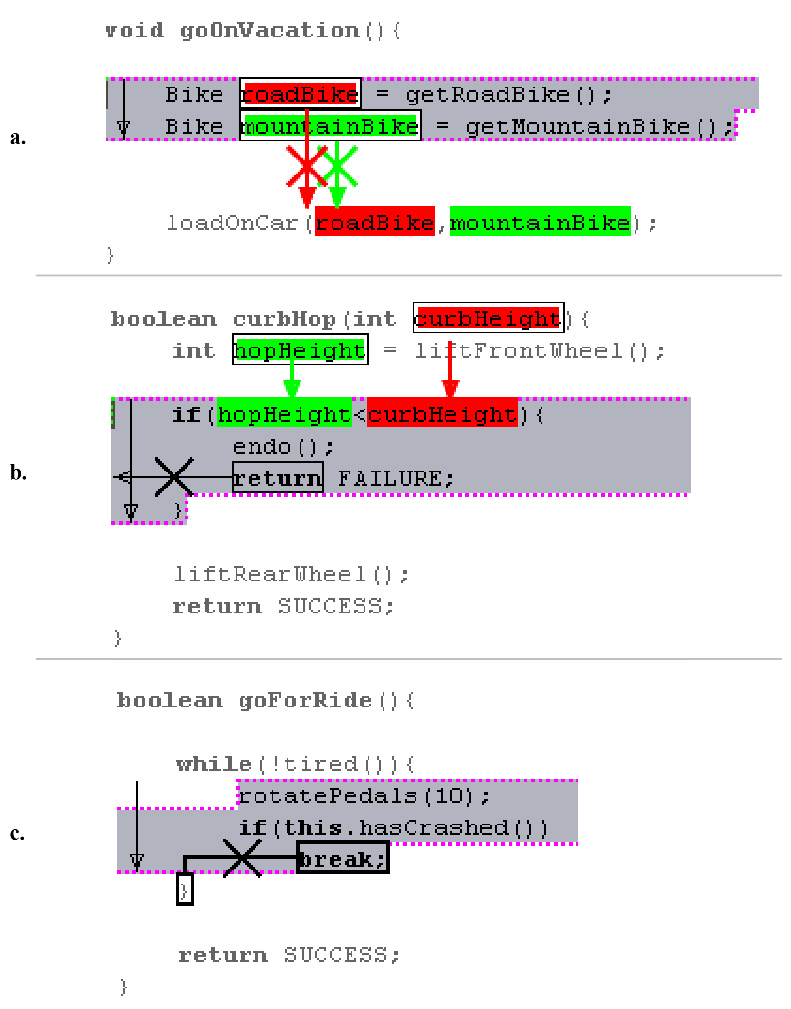
Figure 6. Refactoring Annotations overlaid on program code. The programmer has selected two lines (in grey) to extract. Here, Refactoring Annotations show variable use: `front` and `rear` will be parameters, `truedWheels` will be returned.

These annotations are intended to be most useful when preconditions are not met (Figure 7)[2]. When the selection contains assignments to more than one variable, arrows are drawn from the bottom as multiple return values (Figure 7a). 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 7b). When the selection contains a branch statement, a line is drawn from the branch statement to its corresponding target (Figure 7c). If any of these preconditions are not met, Xs are displayed, 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 change the selection to allow the extraction of a method. One solution is to reduce or enlarge the selection. Other solutions include changing program logic to eliminate **break** and **continue** statements, another kind of refactoring.

The 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 may be easier than correcting six assignments. Secondly, Refactoring Annotations give specific, spatial cues to problem points. For instance, when a branch is selected without its target, boxes are drawn around, and lines are drawn between, both the branch and the target.

[2] Refactoring Annotations currently only handle preconditions 1-3. Precondition 4 is irrelevant, as this tool currently works only with Java. Precondition 5 is already handled by the Eclipse environment, which underlines compilation errors.



a.

```
void goOnVacation(){

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

    loadOnCar(roadBike,mountainBike);
}
```

b.

```
boolean curbHop(int curbHeight){
    int hopHeight = liftFrontWheel();
    if(hopHeight<curbHeight){
        endo();
        return FAILURE;
    }

    liftRearWheel();
    return SUCCESS;
}
```

c.

```
boolean goForRide(){

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

    return SUCCESS;
}
```

Figure 7. Refactoring Annotations display an instance of a violation of refactoring precondition 1 (a), precondition 2 (b), and precondition 3 (c), described in Figure 3.

The control flow annotations appear quite similar to Control Structure Diagrams [16]. 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, depending on whether they are used both inside and outside of the selection. In Refactoring Annotations, the arrows drawn as parameters and return values are similar to the arrows drawn in the Dr. Scheme environment [13]. In Dr. Scheme, arrows are drawn 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, when needed. As a whole, Refactoring Annotations are simply a collection of existing program annotation techniques, tailored specifically for the Extract Method refactoring.

3 User Study

Having demonstrated that there are usability problems with Extract Method tools and having proposed new tools as solutions, I will now present an experiment that has helped to determine whether the new tools overcome these usability problems. The experiment has two parts. In the first part, programmers use the standard mouse and keyboard, Selection Assist, and Box View to select program statements. In the second part, programmers use the standard Eclipse Extract Method Wizard and Refactoring Annotations to identify problems in a selection that violate Extract Method preconditions. In both parts, I evaluate answers for speed and correctness.

3.1 Human 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. All subjects were at least moderately familiar with Java, C, or C++. Most students had around 5 years of programming experience and three had about 20 years. Fifteen students were computer science majors; one was an electrical engineering major. Six students were college seniors, eight were Masters students, one was working towards a Ph.D., and two were not degree seeking. At the time of the experiment, half of the students held jobs that related to computer science.

About half the students typically used integrated development environments such as Eclipse, while the other half typically used editors such as vi [17]. While all students were at least somewhat familiar with the practice of refactoring, only two used automated tools. These users estimated using tools to perform refactoring 20% and 60% of the time.

No subjects had previously used Selection Assist, Box View, or Refactoring Annotations, as they were not available prior to the study.

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 filled out a brief questionnaire regarding their background. They were then trained and allowed to practice using each selection tool. The subjects then began the actual test by selecting a variety of **if** statements with each selection tool. They were then trained and allowed to practice with the Extract Method tools. The subjects then attempted to extract several methods with each tool. Finally, they filled out a post-study questionnaire regarding their subjective impressions of the tools.

I first compared three selection tools: mouse and keyboard, Selection Assist, and Box View. I varied the order of selection tools across subjects to minimize the effect of cross training between tools. Then I compared the two Extract Method tools in fixed order: first, the Eclipse Extract Method Wizard, then Refactoring Annotations. I did not vary order for these tools because Extract Method could only be explained to subjects with a tool that performs the refactoring; Refactoring Annotations do not actually transform code.

I selected all code for this experiment from the open source projects described in Section 1.3.

3.2.1 Training

For each selection tool, I explained the tool and demonstrated it on an example class, and then allowed the subject to practice with the tool to his satisfaction. The same procedure was followed for each Extract Method tool, except that I spent more time explaining how each tool worked because the tools were more complex. Training and practice time for each selection tool was generally less than 5 minutes. Training and practice time for each Extract Method tool was generally 5 to 10 minutes.

3.2.2 Selection Tasks

In the first half of the experiment, subjects were asked to select **if** statements, including the **then**, **else** and **else if** clauses that might go along with them. I chose this task because **if** statements can be difficult to select for the reasons I observed in the pilot study: they can be long, have inconsistent formatting, and have optional curly brackets. I also chose **if** statements because they occur frequently, occur in the same syntax in many languages, and are easy for programmers to identify.

By hand, I chose nine methods containing **if** statements, with varying length and anticipated selection difficulty. I added a three-line **if** statement to the beginning of each method to indicate the beginning of each test. Otherwise, the original code remained unmodified.

I asked subjects to select each **if** statement, then press the ESC key to “mark” the selection. The programming environment measured the time to select a statement as the elapsed time between consecutive presses of ESC. I told subjects that I was measuring accuracy first, then speed, although subjects were given as much time as needed to complete the task. All subjects used each of the three tools.

Table 1. Combined 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.

	Combined Mis-Selected If Statements	Combined Correctly Selected If Statements	Mean selection time (seconds)	Selection time as Percentage of Mouse/Keyboard Selection Time
Mouse/Keyboard	37	303	10.4	100%
Selection Assist	6	355	5.5	54%
Box View	2	357	7.8	71%

3.2.3 Extract Method Tasks

In the second half of the experiment, I gave subjects a pre-selected piece of code and told them to try to perform the Extract Method refactoring. The subjects first used the Eclipse Wizard for four methods, and then used the Refactoring Annotations for the next four methods. I told participants that every piece of code they would try to extract would violate one or more preconditions. Their task was to mark the pieces of code that were causing the precondition violation or violations. If the violation was a return inside the conditional, I told the subjects to mark every **return** keyword. If the violation was a branch, I told the subjects to mark every offending **break** or **continue** keyword. If the violation was assignments to multiple local variables, I told the subjects to mark all assignments to the variables that would have to be returned in the extracted method. Subjects were measured for accuracy and speed, and were given as much time as they needed to complete the task.

4 Results of the Study

Here I 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.

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

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 and median time to correctly identify all violated preconditions, in seconds. Smaller numbers indicate better performance.

	Missed Violation	Irrelevant Code	Missed Break/ Continue	Missed Variable	Mean Identification Time (seconds)	Median Identification Time (seconds)
Eclipse Wizard	11	28	2	12 (0)	164	127
Refactoring Annotations	1	6	2	2 (10)	46	40

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 29%.

The top of Figure 8 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 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 were more error prone using the mouse and keyboard than with Selection Assist. Furthermore, subjects' mean speed varied widely using the mouse and keyboard (standard deviation of 3.02 seconds), but was more consistent using Selection Assist (standard deviation of 0.99 seconds).

The bottom of Figure 8 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 are less error prone with Box View. Subjects' mean speed varied only slightly less with Box View versus the mouse and keyboard (a standard deviation of 2.85 versus 3.02 seconds).

Table 2 displays what kinds of problems 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 irrelavent to the violated precondition, such as marking a **break** statement when the problem was a conditional **return**. "Missed Break/Continue" means that a subject failed to mark a **break** or **continue** statement that was violating a precondition. "Missed Variable" means that a subject failed to mark any occurrence of a variable that is assigned to within the code to be extracted. The parenthesized number next to "Missed Variable" is the number of times a subject correctly marked at least one assignment to such a variable, but failed to mark every assignment. All subjects found every conditional return, so this precondition violation is omitted from the table.

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 every assigned variable in the selection. Subjects were also much less likely to misidentify the precondition violations. Neither tool was more helpful in helping the subjects find **break** and **continue** statements. Subjects correctly identified variables that would need to be returned more often with Refactoring Annotations than with the Eclipse Wizard.

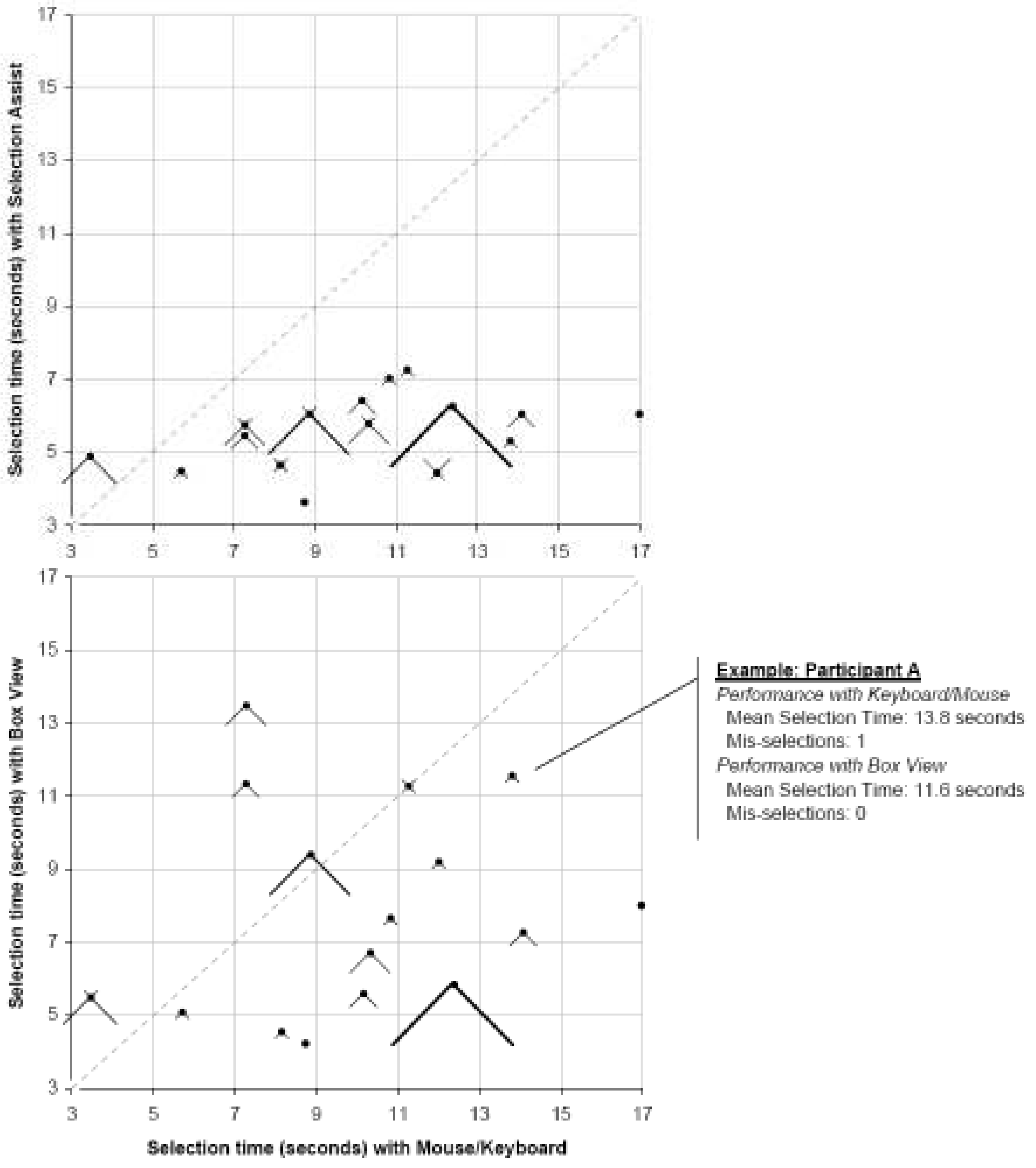


Figure 8. Mean time in seconds to select **if** blocks using the mouse and keyboard vs. Selection Assist (top) and Box View (bottom). Each subject is represented as a whole or partial X. The distance between the bottom legs represents the number of mis-selections using the mouse and keyboard. The distance 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.

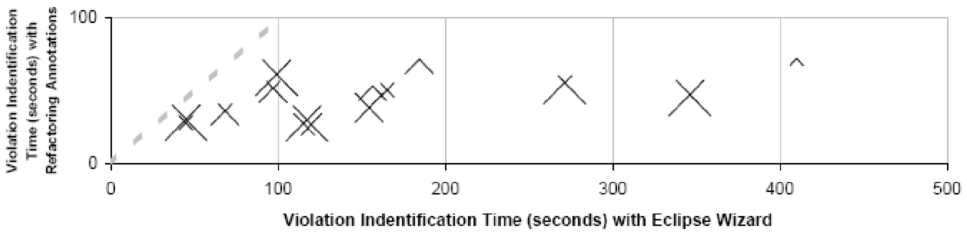


Figure 9. Mean time to identify precondition violations correctly using the Eclipse Wizard versus Refactoring Annotations. Each subject is represented as an X, where the size of the bottom half represents the number of imperfect identifications using the Eclipse Wizard and the size of the top half represents the number of imperfect identifications using Refactoring Annotations.

Figure 9 shows the mean time to identify all precondition violations correctly for each tool and each user. Note that I 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 9, we notice that all users are faster with Refactoring Annotations. We also notice that all users except one were at least as 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.

In terms of speed, Table 2 also shows the mean and median 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.

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

4.2 Questionnaire Results

The post-test questionnaire allowed the subjects to express their preferences for the five tools they tried. For each tool, the subject indicated the tool's helpfulness for the assigned task, indicated likeliness to use the tool in the future if it was available for their preferred environment, and optionally wrote free form comments and suggestions. The complete numeric results of the questionnaire appear in the Appendix.

Most users did not find the keyboard or mouse alone helpful in selecting **if** statements, and generally rated the mouse and keyboard lower than either Box View or Selection Assist.

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. Some subjects reported it was “easy to use,” “fast for highlighting the whole block,” and the “best tool.” One subject explained his positive review by saying he had the “opportunity to view code blocks without being bothered by code specifics.” Other subjects, however, reported it was “too complicated to digest” and “counterintuitive.”

Selection Assist scored the highest of the selection tools, with 15 of 16 users reporting it was helpful and they were likely to use it again. Subjects reported the tool is “easy to operate,” “intuitive,” and “simple, unobtrusive, [and] easy to understand.” The only widespread disappointment was that “we still have to do the highlighting by ourselves.”

Subjects were unanimously positive on the helpfulness of Refactoring Annotations, and almost all of them preferred Refactoring Annotations to the standard Eclipse Extract Method Wizard. Users reported that Refactoring Annotations were “easy to use and very intuitive,” “extremely helpful for large blocks of code,” and “a sweet addition to any IDE.” Some users reported difficulty in distinguishing between colors and one user reported Refactoring Annotations seemed “rather complex” on first impression. 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 the new tools would be helpful outside of the context of the experiment.

4.3 Shortfalls of the Experiment

While the results of this study indicate that the new refactoring tools are beneficial to performing Extract Method, the study has several shortfalls:

- The code used in both the selection and Extract Method workloads may not be representative of a typical refactoring situation. Although the code comes from real projects, I selected code by hand to stress different aspects of the tools.
- Subjects learned how to use each tool in an extremely short amount of time. Given more practice, I expect better programmer performance with every tool.
- Some procedural irregularities occurred during the study. During the selection test, while every participant tried each tool on each code set, a flaw in the study design caused distribution of tools to code sets to be uneven. In the most extreme case, one code set was traversed only twice with the mouse and keyboard while another code set was traversed eight times using the Selection Assist. Furthermore, during the selection test, two subjects were given only partial code sets, so they tried two tools for significantly shorter durations than the third tool. These study irregularities were confined to the selection tasks.

- Since the order of the tools during the Extract Method test was fixed, I would automatically expect somewhat better performance from Refactoring Annotations because the subjects' experience with Extract Method increased as the experiment went on.
- Box View displayed two bugs during the study. Participants either briefly had to revert to using the mouse, or did not notice the mistakes caused by Box View. I have omitted selection mistakes related to bugs with Box View from these results.

Overall, I believe that despite the shortcomings and irregularities of this study, the results are nevertheless useful in comparing the tools against one another.

5 Discussion

5.1 Interpretation of Results

During this study, I have observed that new, alternative view tools can improve programmer accuracy and speed in 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. An effective statement selection tool is critical to a successful Extract Method refactoring.

Refactoring Annotations are preferable to a wizard-based approach to show precondition violations during the Extract Method refactoring. The results of this study indicate that Refactoring Annotations effectively communicate the location and severity of precondition violations. 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.

5.2 Recommendations

Creating and studying views for Extract Method revealed a number of desirable properties for tools that assist with refactoring in general. 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 constructs a programmer would want to select.

Tools that assist in displaying 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 the violations. A tool should tell the programmer what the compiler already knows, rather than needing "to basically compile the whole snippet in my head," as one Eclipse bug reporter mentioned [18].
- Show every violated precondition. This helps the programmer in accessing 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 the code can be refactored with a few minor changes, or the code is nearly hopeless.
- Display the violation relationally, when appropriate. Violations often are 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 message for another and waste time tracking down and trying to fix a violation that does not exist.

6 Related Work

Many tools provide support for the Extract Method refactoring, but few deviate from the wizard-and-error-message interface described in this paper. 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 [19]. 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 [20]. Again, this may or may not be what you intended, and is not the only solution to the problem.

StarDiagram for Eclipse performs refactoring using a non-text-based representation [21]. O'Connor and colleagues implement Extract Method using a graph notation to help the programmer recognize and eliminate code duplication, 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.

A variety of authors have suggested using UML diagrams to refactor programs [22,23,24,25], mainly by manipulating class diagrams. However, as Don Roberts notes, "most refactorings have to manipulate portions of the system that are below the method level" [26], so representing refactoring preconditions using class diagrams appears difficult. In one sense, UML is a poor representation for performing refactorings because a refactoring must alter syntax, yet UML always abstracts away syntax.

Previous research has suggested some desiderata for refactoring tools. Roberts' thesis describes his experience building and using the Refactoring Browser, the original refactoring tool and the first to implement Extract Method [26]. Roberts notes that the original tool was so unpopular that the designers did not even use it themselves. Upon reflection, Roberts illustrated three characteristics that every good refactoring tool should have: speed, undo support, and tight IDE integration. Most refactoring tools seem to have taken this message to heart.

7 Conclusions

Refactoring is an important part of software development and refactoring tools are critical to making refactoring fast and behavior preserving. However, when a refactoring is not immediately possible, most modern refactoring tools do a poor job of communicating problems with the programmer. Programmers are unlikely to correctly identify the root problems when a refactoring goes wrong, and are therefore unlikely to successfully correct problems. Programmers may waste time trying to correct problems that do not exist, or worse, manually alter code and inadvertently change behavior in the process. Moreover, the experience of a failed refactoring may cause a programmer to avoid refactoring code in the future.

In this paper, I have presented three alternate program views that help programmers avoid selection errors and understand refactoring precondition violations. Through a user study, I have demonstrated that these views exhibit several qualities that improve the experience of refactoring, help programmers correctly identify problems with refactoring, and increase speed during several parts of the refactoring process. Alternate views appear to be a promising approach to increasing usability of refactoring tools.

8 Acknowledgements

Thanks to Suresh Singh and Jim Larson for help during experimental design. For their reviews and advice, I would like to thank Robert Bauer, Paul Berry, Andrew Black, Iavor Diatchki, Tom Harke, Brian Huffman, Mark Jones, Chuan-kai Lin, Ralph London, Philip Quitslund, Tim Sheard, and Aravind Subhash. Special thanks to colleagues and students who participated in testing and made this research possible.

9 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. Opdyke, W.: *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign (1992)
3. Roberts, D., Brant, J. and Johnson, R.: A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253-263, (1997)
4. Li, H., Reinke, C. and Thompson, S.: Tool Support for Refactoring Functional Programs. In *Proc. ACM SIGPLAN Workshop on Haskell*. pp. 27-38. ACM, Uppsala, Sweden (2003)
5. Schrijvers, T., Serebrenik, A. and Demoen, B.: Refactoring Prolog Code. In *Proc. 18th Workshop on (Constraint) Logic Programming*, Potsdam, Germany (2004)
6. Fowler, M. Crossing Refactoring's Rubicon. http://www.martinfowler.com/articles/refactoringRubicon.html, (2001)
7. The Eclipse Foundation: Eclipse, http://www.eclipse.org, accessed November 2005
8. Azureus Incorporated: Azureus, http://azureus.sourceforge.net, accessed November 2005
9. Thomas, A. and Bareshev, D.: GanttProject. http://ganttproject.sourceforge.net, accessed November 2005
10. JasperSoft Corporation: JasperReports. http://jasperreports.sourceforge.net/, accessed November 2005
11. Hugunin, J. and Warsaw, B.: Jython, http://www.jython.org, accessed November 2005
12. Sun Microsystems Incorporated: Java 1.4.2 Standard Libraries. http://java.sun.com/j2se/1.4.2/download.html, accessed November 2005
13. Findler, R., Clements, J., Matthew, Krishnamurthi, S., Steckler, P. and Felleisen, M.: DrScheme: A Progamming Environment for Scheme. *Journal of Functional Programming*, 12(2):159-182, (2002)
14. Free Software Foundation: Parentheses - GNU Emacs Manual. http://www.gnu.org/software/emacs/manual/html_node/Parentheses.html, accessed April 2006
15. Adobe Systems Incorporated: Adobe GoLive. http://www.adobe.com/products/golive, accessed November 2005
16. Hendrix, D., Cross, J., Maghsoodloo, S. and McKinney, M.: Do Visualizations Improve Program Comprehensibility? Experiments with Control Structure Diagrams for Java. Haller, S. (ed.): In *Proc. Thirty-First SIGCSE Technical Symposium on Computer Science Education*, Vol. 32. pp. 382-386. ACM, Austin, Texas (2000)
17. Joy, W. and Horton, M.: An Introduction to Display Editing with Vi. http://www.unixprogram.com/manuals/usd.15.vi.pdf, (1984)
18. Andersen, T.R.: Extract Method: Error Message Should Indicate Offending Variables. https://bugs.eclipse.org/bugs/show_bug.cgi?id=89942, accessed November 2005
19. Omnicore Software: CodeGuide. http://www.omnicore.com/, accessed November 2005
20. Xref-Tech: Xrefactory. http://www.xref-tech.com, accessed November 2005
21. O'Connor, A., Shonle, M. and Griswold, W.: Star Diagram with Automated Refactorings for Eclipse. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange*. ACM, San Diego, California (2005)
22. Astels, D.: Refactoring with UML. In *Proc. 3rd Int'l Conference on eXtreme Programming and Flexible Processes in Software Engineering*. pp. 67-70, Alghero, Italy (2002)
23. Boger, M., Sturm, T. and Fragemann, P.: Refactoring Browser for UML. In *Proc. NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. pp. 366-377. Springer-Verlag, Erfurt, Germany (2003)
24. Sunyé, G., Pollet, D., Le Traon, Y. and Jezéquél, J.-M.: Refactoring UML Models. In *Proc. 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools*. pp. 134-148. Springer-Verlag, Toronto, Ontario, Canada (2001)
25. Van Gorp, P., Stenten, H., Mens, T. and Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. *Lecture Notes in Computer Science*, 2863:144-158, (2003)
26. Roberts, D.: *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign (1999)

Appendix: Posttest Questionnaire (N=16)

Please answer the following questions based on your experience during this test. If you feel it necessary, feel free to write notes in the margin to explain your answers.

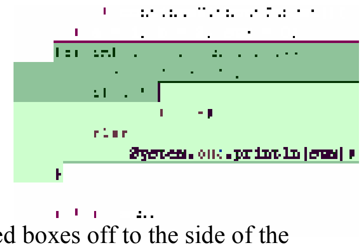
Helpfulness

1. I found the **mouse/keyboard alone** a helpful mechanism for selecting *if* blocks.

0	Strongly Disagree
9	Disagree
1	Neutral
5	Agree
0	Strongly Agree

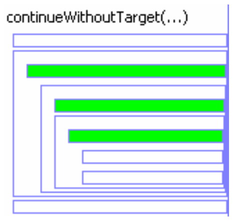
2. I found the **selection assist tool** (green statement highlighter) a helpful mechanism for selecting *if* blocks.

0	Strongly Disagree
1	Disagree
0	Neutral
5	Agree
10	Strongly Agree



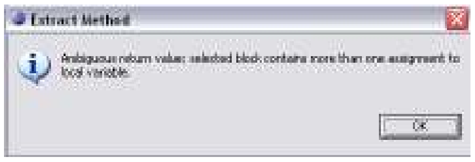
3. I found the **statement box view** (nested boxes off to the side of the code) a helpful mechanism for selecting *if* blocks.

0	Strongly Disagree
0	Disagree
3	Neutral
7	Agree
6	Strongly Agree



4. I found the **extract method wizard** (popups that contained an error message) a helpful indication of what went wrong when I tried to extract a method.

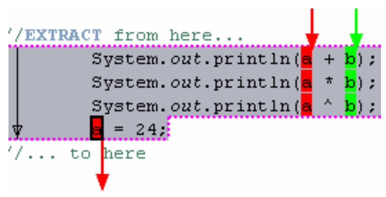
0	Strongly Disagree
3	Disagree
3	Neutral
6	Agree
3	Strongly Agree



Ambiguous return value: selected block contains more than one assignment to local variable.

5. I found the **extract method annotations** (colors/lines on top of code) a helpful indication of what went wrong when I tried to extract a method

0	Strongly Disagree
0	Disagree
0	Neutral
3	Agree
13	Strongly Agree

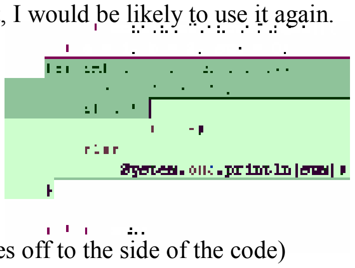


```
//EXTRACT from here...
        System.out.println(a + b);
        System.out.println(a * b);
        System.out.println(a ^ b);
        a = 24;
//... to here
```

<u>Likely to Use Again</u>

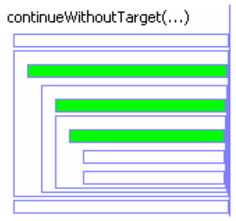
6. If the **selection assist tool** (green statement highlighter) were available for my usual development environment, I would be likely to use it again.

0	Strongly Disagree
0	Disagree
1	Neutral
3	Agree
12	Strongly Agree



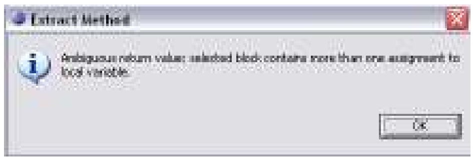
7. If the **statement box view** (nested boxes off to the side of the code) were available for my usual development environment, I would be likely to use it again.

0	Strongly Disagree
3	Disagree
3	Neutral
6	Agree
4	Strongly Agree



8. If the **extract method wizard** (popups that contained an error message) were available for my usual development environment, I would be likely to use them again during the extract method refactoring.

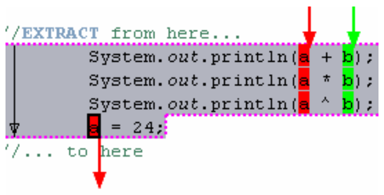
1	Strongly Disagree
1	Disagree
4	Neutral
7	Agree
3	Strongly Agree



Ambiguous return value: selected block contains more than one assignment to local variable.

9. If the **extract method annotations** (colors/lines on top of code) were available for my usual development environment, I would be likely to use them again during the extract method refactoring.

0	Strongly Disagree
0	Disagree
0	Neutral
1	Agree
15	Strongly Agree



```
//EXTRACT from here...
        System.out.println(a + b);
        System.out.println(a * b);
        System.out.println(a ^ b);
        a = 24;
//... to here
```