Improving Usability of Refactoring Tools

Emerson Murphy-Hill, Portland State University, 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 tools to assist in the Extract Method refactoring, and describe a user study that compares these new tools to existing tools. The results of the study show that these new tools increase both the speed and accuracy of refactoring. Based on the new tools and my observation of programmers, I present several guidelines to help build future refactoring tools.

Problem and Motivation

Refactoring, the process of changing the structure of code without changing the way a program behaves, is a potentially useful technique for building and maintaining code [Fowler99]. For example, using the Extract Method refactoring, a programmer may remove duplicated code by putting it into a new method and calling that new method instead. Semi-automated refactoring tools in most commercial programming environments relieve the programmer from having to make tedious and error-prone changes by hand. For example, a tool can encapsulate a field using getter and setter methods, and the tool will automatically replace all direct references to the field with references to the new methods. In this way, a refactoring tool offers the potential for tremendous increases in refactoring productivity.

In a small exploratory study, I set out to determine how productivity is effected using refactoring tools. I used the Extract Method refactoring tool in the Eclipse environment, because Extract Method is a common yet complex refactoring [Fowler99,Murphy06] and because Eclipse contains a mature Extract Method tool. In this exploration, I observed 11 programmers perform several Extract Method refactorings in Java for about half an hour per programmer. Six of the programmers were graduate students, two were professors, and three were commercial software developers. During each session, the programmer was allowed to extract methods anywhere in a large corpus of open-source code. Each programmer was able to successfully extract between 2 and 16 methods. Nine out of the eleven programmers triggered at least one refactoring error message during the study. I made two particularly interesting observations:

- **Observation 1.** Programmers sometimes have difficulty selecting program statements to be extracted into a new method. This was the most common cause of error messages. Selection problems were typically caused by poor or inconsistent formatting, and very long statements.
- **Observation 2.** Programmers sometimes have difficulty understanding the meaning of error messages related to violations of refactoring preconditions. Refactoring preconditions are properties of the original program that must hold for a refactoring to be behavior-preserving. For example, one error message in Eclipse is "Ambiguous return value: selected block contains more than one assignment to local variable." Programmers sometimes were confused or discouraged in the face of such error messages.

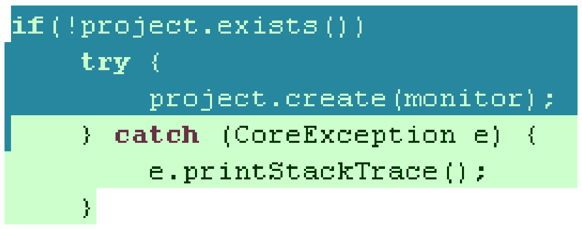
In general, I regard both of these problems as significant hindrances to refactoring for several reasons. First, all refactorings require the programmer to select program elements and to understand violated preconditions, so the problems are not specific to Extract Method. Second, these problems caused programmers to use refactoring tools less often, because they misunderstood how the tool works. Third, while the frequency with which these problems appeared varied among programmers, the problems were sometimes severe. For example, in one session error messages resulted during more than 2/3 of the Extract Method attempts. While this study does not say with certainty that errors are the most important problem facing refactoring tools, the study does provide a glimpse into the usability problems with existing tools. Further details about this exploratory study can be found in my technical report [Hill06].

To address the problems exposed in the exploratory study, I created two tools to help the programmer select code and one tool to help the programmer understand violations of refactoring preconditions. These tools were written in Eclipse; a short movie and downloads are available at http://www.multiview.cs.pdx.edu/refactoring.

The two tools to assist the programmer in selecting statements suitable for input to Extract Method tools are called Selection Assist and Box View. Selection Assist (Figure 1) overlays program text with a light green color, in order to provide a visual cue to the programmer of the extent of a program statement. Box View (Figure 2) resides to the left of the program text, and represents each statement as a rectangle. When a box is selected, the corresponding program text is selected, and vice versa.

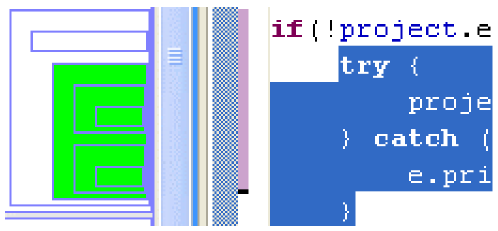
The tool that helps the programmer understand violations of refactoring preconditions is called Refactoring Annotations. Refactoring Annotations eliminate the need for several error messages � instead, a description of the error is displayed graphically. The code that is about to be extracted into a new method is first selected by the programmer, then Refactoring Annotations are activated before the refactoring takes place. Each variable involved in the refactoring is assigned a distinct color. Lines are drawn in the top of the selection to indicate parameters that must be passed in to the extracted method. Lines are drawn through the bottom of the selection identifying the value that will be returned. Other lines are drawn to indicate control flow. When a refactoring precondition is violated, an X is drawn on top of the appropriate line, indicating the location(s) of the offending code. Figure 3 shows an example of Refactoring Annotations where two values would be returned from the extracted method � a violation of a refactoring precondition.

At this point, most research projects halt: a problem has been identified, a tool has been created, and the problem is considered solved. However, this is just the beginning of my research, because my objectives included showing measurable usability improvements in real refactoring situations and producing guidelines to help build other kinds of highly usable refactoring tools.



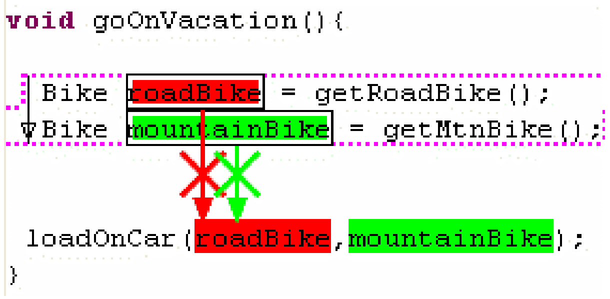
```
if(!project.exists())
    try {
        project.create(monitor);
    } catch (CoreException e) {
        e.printStackTrace();
    }
```

Figure 1. Selection Assist.



```
if(!project.e
    try {
        proje
    } catch (
        e.pri
    }
```

Figure 2. Box View.



```
void goOnVacation(){

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


  loadOnCar(roadBike,mountainBike);
}
```

Figure 3. Refactoring Annotations.

Background & Related Work

The three tools I have created provide only a limited technical contribution, as they can be viewed as simply another application of existing user interface techniques. DrScheme has an tool similar to Selection Assist [Findler02] and Box View's appearance is similar to a web page authoring tool in Adobe GoLive [Adobe05]. Refactoring Annotations are similar to Control Structure Diagrams [Hendrix00] and variable arrows drawn in DrScheme [Findler02]. Nevertheless, my tools represent a novel application of existing techniques.

Some tools avoid having to present precondition violations by silently resolving them. For instance, when you try to extract an invalid selection in Code Guide, the environment expands the selection to a valid list of statements [Omni05]. You may then end up extracting more than you intended. With X-Refactory, if you try to use Extract Method on code that would return more than one value, the tool generates a new tuple class [XRef07]. Again, this may or may not be what you intended, and is not the only solution to the violation.

When considering how to improve the human interface to refactoring tools, it is worthwhile to examine existing usability guidelines. For example, if we try to apply Smith and Mosier�s five objectives for data display [Smith86] to refactoring precondition violations, we find that the error messages presented by refactoring tools meet the objectives quite well. However, as we saw in the last section, there are still problems with these error messages. Generally, I have found that high-level guidelines are not specific enough to guide the development of the user interface to refactoring tools. Furthermore, if we try to apply lower level guidelines, the advice can imply an overly-restrictive user interface. For example, if we apply Shneiderman's principles for good error messages [Shn82], we are compelled to use a natural language notation to explain errors. But as I will show later in this paper, a graphical notation can be employed more effectively.

While the literature is rife with new refactoring tools and techniques, very little work exists on what the interface to refactoring tools should look like. Mealy and Strooper comparatively evaluated several refactoring tools, concluding that "usability of refactoring tools requires further research/consideration" [Mealy06]. My research provides a step in that direction.

Uniqueness of the Approach

This research is unique in two respects. First, the three tools I have built are based on empirical observation and have been validated in a controlled human-subjects experiment. Second, based on the observations of programmers, the new tools, and the old tools, I expose a set of guidelines that I anticipate will be useful for future refactoring tools. The experiment and derived guidelines are discussed in the next section.

Results and Contributions

I performed a controlled experiment in order to determine if and when the new refactoring tools allow programmers to perform better than with existing refactoring tools. The experiment has two parts. In the first part, programmers used the standard mouse and keyboard, Selection Assist, and Box View to select program statements. In the second part, programmers used the standard Eclipse Extract Method Wizard (with error messages) and Refactoring Annotations to identify problems in a selection that violated Extract Method preconditions. In both parts, I evaluated their answers for speed and correctness. Experiments were conducted with each subject individually, lasting between 0.5 and 1.5 hours each.

I drew subjects from an object-oriented programming class containing 18 students, 16 of whom elected to participate. Of these 16 participants, most had around 5 years of programming experience, but three students had around 20 years. Half of the students had used integrated development environments, but only two students had used refactoring tools.

Space constraints prohibit a full description of the experiment, explanation of the results, discussion of the threats to validity, and the contents of a post-test questionnaire; these can be found in my technical report [Hill06].

Experiment 1: Code Selection

In this experiment, I compared how fast people select statements in open-source Java code using three tools. Using a randomized blocked experiment design, I randomly assigned each participant to one of five groups. Within each group, each subject was told to use a keyboard or mouse (subject's choice), Selection Assist, or Box View to select every `if` statement in several predefined methods. Between groups, tool usage and code selection order was randomized. Essentially, each programmer was told to select about 20 `if` statements with each tool. Each subject was trained for a few minutes on how to use each tool and was allowed to practice using each tool on some example code. I recorded whether each programmer selected the statement correctly (ignoring whitespace), and how long it took to select each statement. A summary of the results across all participants is shown below:

Tool	Total Mis-selected If Statements	Total Correctly Selected If Statements	Mean Selection Time	Selection Time as Percentage of Mouse/Keyboard Selection Time
Keyboard/Mouse	37	303	10.2 seconds	100%
Selection Assist	6	355	5.5 seconds	54%
Box View	2	357	7.8 seconds	71%

The data in this table shows that Selection Assist allowed the programmer to select statements fastest, but Box View allows the most accurate selection. Both Selection Assist and Box View were faster and more accurate than the keyboard or mouse.

A post-experiment questionnaire was administered to gauge users' subjective feelings about the tools. 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. Selection Assist scored the highest marks of the selection tools, with 15 of 16 users reporting it was helpful and they were likely to use it again.

Experiment 2: Error Comprehension

In this experiment, I compared how well error messages and Refactoring Annotations helped programmers understand the causes of violated refactoring preconditions. I randomly assigned each participant to one of two groups. Both groups first used the standard Eclipse Extract Method Wizard (with error messages) on 4 Extract Method candidates, then used Refactoring Annotations on 4 different Extract Method candidates. Subject group 1 attempted Extract Method candidate set A, then candidate set B, whereas subject group 2 attempted set B and then set A. I pre-selected the Extract Method candidates from open-source Java code to vary in length and anticipated difficulty. However, set A and set B were chosen to contain candidates of approximately equal size and to contain the same number and kind of violated preconditions. Participants were instructed to use Eclipse error messages to determine the location(s) of code causing errors in 4 Extract Method candidates, then repeat the process for different code using Refactoring Annotations. For example, suppose the subject were given several statements to extract into a new method, but those statements contained assignments to two different variables whose values were used in the following code. I would then ask the subject to diagnose the problem using the assigned tool and then allowed them to indicate the two offending variables. For each Extract Method candidate, the time to complete the task and the correctness of the response was recorded. Below is a summary of the results across all participants:

Tool	Missed Violation	Irrelevant Code	Mean Identification Time
Eclipse Error Messages	11	28	164 seconds
Refactoring Annotations	1	6	46 seconds

In the table, �Missed Violation� means that a subject failed to recognize that one or more preconditions were being violated.

�Irrelevant Code� means that a subject indicated some piece of code that was irrelevant to the violated precondition, such as indicating a `break` statement when the problem was multiple return values. The data in the table shows that, using Refactoring Annotations, programmers were several times less error prone and about three times faster.

In the post-experiment questionnaire, 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 (with error messages). 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 messages the Eclipse tool produced, one subject quipped, �who reads alert boxes?�

Interpretation

While the results show the new tools as promising alternatives to existing tools, the results are open to interpretation. Due to limitations of the experiments and variability in programming experience and context, I cannot claim that any tool is strictly "better." Furthermore, because the human subjects were students from one particular class and therefore do not represent a random sample of programmers, it would be inappropriate to apply common statistical techniques, such as analysis of variance. Instead, I encourage you to take the data at face value and build your own interpretation. What follows is my interpretation of the results.

Programmers can use either Box View or 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 desired 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 for showing precondition violations during the Extract Method refactoring. The results of this study indicate that Refactoring Annotations communicate the location of precondition violations effectively. When a programmer has a good understanding of refactoring problems, I believe the programmer is likely to be able to correct the problems and successfully perform the refactoring.

Derived Guidelines

Based on the experiments, observations, and tools described in this paper, I have derived a set of guidelines for building future tools that help with refactoring. Tools that help the programmers with selection 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. For example, while bracket matching can be helpful, bracketed statements are not the only meaningful program construct that a programmer needs to select.

Tools that communicate violated refactoring preconditions should:

- Be lightweight. The time it takes a programmer to complete a tool-assisted refactoring should not take longer than the time it takes the same programmer performing the refactoring manually.
- Indicate the location(s) of precondition 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 [Ander05].
- Show every violated precondition. This helps the programmer in accessing the overall severity of the violations.
- Help programmers distinguish precondition violations (showstoppers) from warnings and advisories. Programmers should not be left wondering whether or not 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 the relationship between dispersed 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 be able to confuse one error message for another and waste time tracking down and trying to fix a violation that does not exist.

While these guidelines may seem obvious, prior to this research I could identify only about half of them, and even then had little idea how they might be realized in a practical refactoring tool or whether they were truly important to supporting the programmer when refactoring. In current research, I am working towards expanding these guidelines to all phases of the refactoring process and for several refactorings.

Contributions

This research makes three contributions: three new tools designed to address observed usability problems with existing tools, a human-subjects experiment showing concrete usability improvements using these new tools, and guidelines for future refactoring tools based on my observations.

Conclusion

Refactoring is an important part of software development and refactoring tools are critical to making refactoring fast and behavior preserving. In this paper, I have presented three new tools that help programmers avoid selection errors and understand violations of refactoring preconditions. Through a user study, I have demonstrated that these tools exhibit several qualities that improve the experience of refactoring, help programmers correctly identify problems with a proposed refactoring, and increase speed of the refactoring process. I hope that these qualities will be adopted by new refactoring tools, make tools more usable and thus more used, and eventually contribute to the production of more reliable, on-time software.

References

[Adobe05] Adobe Systems Incorporated, Adobe GoLive, http://www.adobe.com/products/golive, 2005.

[Ander05] T.R. Andersen, �Extract Method: Error Message Should Indicate Offending Variables.� https://bugs.eclipse.org/bugs/show_bug.cgi?id=89942, 2005.

[Findler02] R. Findler, J. Clements, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, �DrScheme: A Programming Environment for Scheme,� Journal of Functional Programming, vol. 12, pp. 159-182, 2002.

[Fowler99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code: Addison-Wesley Professional, 1999.

[Hendrix00] D. Hendrix, J. Cross, S. Maghsoodloo, and M. McKinney, "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.

[Hill06] E. Murphy-Hill. 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, 2006.

[Mealy06] E. Mealy and P. Strooper, "Evaluating software refactoring tool support," presented at Australian Software Engineering Conference, 2006.

[Murphy06] G. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," IEEE Software, 2006.

[Omni05] Omnicore Software, CodeGuide, http://www.omnicore.com, 2005.

[Robert99] D. Roberts, �Practical Analysis for Refactoring,� PhD Thesis. University of Illinois at Urbana-Champaign, 1999.

[Shn82] B. Shneiderman, A. Badre, and B. Shneiderman, "System Message Design: Guidelines and Experimental Results," in Directions in Human/Computer Interaction: Ablex, 1982, pp. 55-78.

[Smith86] S. L. Smith and J. N. Mosier. Design Guidelines for Designing User Interface Software. Technical Report MTR-10090 (August), The MITRE Corporation, Bedford, MA 01730, USA, 1986.

[XRef07] X-Refactory. Xref-Tech. 2007.