Designing Refactoring Tools for Developers

Dustin Campbell
Microsoft Corporation
dustinca@microsoft.com

Mark Miller
Developer Express Inc.
markm@devexpress.com

Abstract

Because manual refactoring is both tedious and prone to error, automatic refactoring tools have become increasingly important to a programmer's workflow. Unfortunately, many refactoring tools suffer from deep discoverability and usability problems that make them less useful for general development. In this paper, we present three primary issues that plague refactoring tools and present our approach to solving these issues in a commercial add-in for Microsoft Visual Studio.

1. Introduction

With the help of a good tool, refactoring can be a natural part of any programmer's general development process. Tools should make the application of refactorings trivial at any time during the development of program code. For instance, a programmer might write a complex expression and immediately refactor, breaking it into well-named variables using the Introduce Explaining Variable [1] refactoring. Unfortunately, many refactoring tools inadvertently place barriers between programmers and this natural style of refactoring [2]. We will focus on three of the most common barriers.[1]

- Discoverability. Many refactoring tools are difficult to learn to use—especially if the programmer is not already comfortable with refactoring [3].
- Lack of trust or lack of familiarity. Often, programmers will not apply a refactoring because they are not sure how it will transform their code.
- Productivity. Many programmers do not use refactoring tools because they feel that they can apply refactorings more efficiently by hand. While this might not be true *per se*, the *perception* translates into disuse of tools.

In this paper, we explore each of these barriers in turn and describe our solutions for them. Each solution is implemented in a commercial add-in for Microsoft Visual Studio, Refactor! Pro[2].

2. Discoverablity as a Barrier

Refactoring tools often assume that a programmer already knows how to refactor and is familiar with the catalog of refactorings [1]. A programmer, however, might intuitively refactor her code without knowing the names of any of the refactorings she is applying [3]. This programmer would need guidance in identifying how refactorings might be useful in order to take full advantage of a tool.

To remove this barrier, we introduced a contextual availability-checking system for refactorings. When a refactoring can be applied in the current editor context (based on caret position, selection and language model), the refactoring appears in a menu, along with any other available refactorings. In addition, the programmer is notified via a smart tag if any refactorings can be applied in the current context. To enhance discoverability further, we added a background code analysis and highlight mechanism to highlight code smells where powerful but perhaps less well-known refactorings are available. These approaches greatly improve the discoverability of when and how refactorings can be applied.

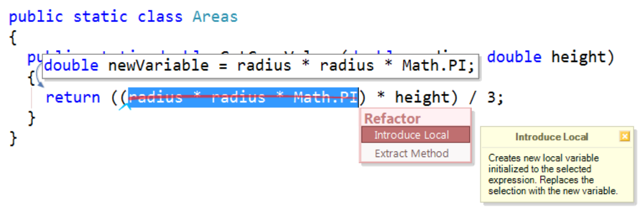
3. Trust or Familiarity as a Barrier

Some programmers fear that an automated tool might mangle their program code. With quality tools, this seldom happens. However, distrust of a tool or a lack of familiarity can prevent the programmer from experimenting with new refactorings. Therefore, it is important to indicate *what* a refactoring will do *before* the programmer decides whether to apply it.

We chose to address this problem with a preview hinting system that provides the programmer with a visualization of the operations that a refactoring will perform, without

[1] The three barriers presented were derived from customer feedback gathered during the authors' development of refactoring tools.

[2] http://devexpress.com/refactor



```
public static class Areas
{
  pu double newVariable = radius * radius * Math.PI; double height)
  {
    return ((radius * radius * Math.PI) * height) / 3;
  }
}
```

Creates new local variable initialized to the selected expression. Replaces the selection with the new variable.

Figure 1. Preview hinting.

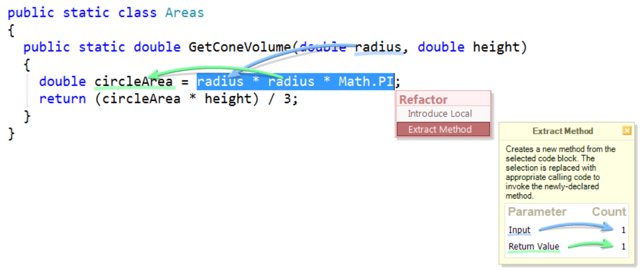
actually performing them. The visualization can take different forms. Some refactorings might provide a visualization which looks very much as if a copy editor has used a red pen to mark several corrections across the code (fig. 1). Other refactorings might take a different approach. For example, the Extract Method [1] visualization uses arrows to represent the dependencies of the selected code in a style reminiscent of how an American football coach might diagram a play (fig. 2).

We have found that preview hinting can play a major role in decreasing the resistance to applying refactorings by helping to build the programmer's trust for a tool. We have also found that preview hinting is more effective than the older convention of modal code preview confirmation windows, which require a commitment on the part of the programmer to first apply the refactoring to see what it will do.

4. Productivity as a Barrier

A common complaint of refactoring tools is that many programmers feel that they can refactor more efficiently by hand [2]. In many cases, this isn't entirely true. A refactoring tool performs enough code transformations that manual code editing could not possibly best it. However, the perception of low productivity is real and valid. We will focus on two primary design choices made by many refactoring tools that influence this perception.

First, many refactoring tools suffer from an explosion of keyboard shortcuts. This usually occurs because each refactoring receives a different keyboard shortcut. Trying to remember every shortcut can be taxing on a programmer's productivity. In contrast, we chose to assign only *one* keyboard shortcut for all refactoring. When pressed, that single keyboard shortcut invokes the contextual availability-checking system to determine which refactorings are currently available. If only one refactoring is available, that refactoring is immediately applied. If more than one refactoring is available, a menu of all available refactorings is displayed.



```
public static class Areas
{
  public static double GetConeVolume(double radius, double height)
  {
    double circleArea = radius * radius * Math.PI;
    return (circleArea * height) / 3;
  }
}
```

Creates a new method from the selected code block. The selection is replaced with appropriate calling code to invoke the newly-declared method.

Parameter	Count
Input	1
Return Value	1

Figure 2. Code dependency arrows.

The second design choice that can detract from programmer productivity is the use of modal dialogs. Often, a refactoring will have many optional behaviors. For example, an implementation of Extract Method might provide several options that affect the signature of the generated method. In order to set options before applying a refactoring, many tools choose to display a dialog. In addition, that dialog is made modal to ensure that the programmer does not modify any program code while setting options for a refactoring to be applied.

Modal dialogs detract from programmer productivity by presenting the programmer with new UI that *must* be dealt with in order to return to writing code. This is an important point: while working with the dialog, the programmer is no longer writing program code. To make matters worse, modal dialogs visually obscure the code below—the very code that the programmer wants to transform. Furthermore, modal dialogs are often littered with buttons, which tend to result in a switch from hands on the keyboard to a reach for the mouse. All of this adds up to a user interface that is much less efficient than it should be.

Instead of the traditional modal dialog design choice, we have gone to great lengths to minimize the questions a programmer must answer when applying a refactoring. We separate interactive phases (where these questions are answered) into two areas. The first area is in the menu displayed before a refactoring is applied and is tightly integrated with the preview hinting discussed above. For example, if a refactoring has several flavors, we might present those flavors in a submenu. Any additional interactive states take place *inside* the code editor, with the appropriate UI weaved directly onto the surface of the editor itself. These are a few of the techniques we use to enhance productivity.

5. Conclusion

Building a refactoring tool is hard. Building a refactoring tool that programmers want to use is harder yet. However, with creative thinking and attention to productivity, the task is not impossible.

References

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

[2] E. Murphy-Hill, and A. Black, "Why Don't People Use Refactoring Tools?," Proceedings of the First Workshop on Refactoring Tools, 2007.

[3] P. Weißgerber, B. Biegel, and S. Diehl, "Making Programmers Aware Of Refactorings," Proceedings of the First Workshop on Refactoring Tools, 2007.