

A Model of Refactoring Tool Use

Emerson Murphy-Hill

The University of British Columbia

emhill@cs.ubc.ca

Abstract

For the most part, refactoring tools have changed little since the Smalltalk Refactoring Browser. By continuing to mimic the Refactoring Browser’s user interface, the community of tool builders may be not find new user interfaces that help programmers do their job more effectively. In this position paper, I put forward a general model of how programmers use Refactoring Browser-like tools. I argue that the model is useful for breaking away from the design of traditional refactoring tools and thinking about the design of new ones.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Object-oriented design methods

General Terms Design

Keywords refactoring, tools, model, process

1. Origins of the Model

In 2008, Andrew Black and I published a paper at the International Conference on Software Engineering that discussed two problems with existing refactoring tools [8]. By doing a formative study where we observed several programmers in the lab performing EXTRACT METHOD refactorings, we showed that programmers have difficulty selecting code for refactoring and understanding error messages. We then attempted to implement solutions to both problems; two tools to help programmers select code, and one that displays error messages graphically. Our evaluation provided evidence that our techniques were more usable.

But how well do our observations about problems – or claims of usability of the improvements – generalize to other brands of refactoring tools, other than Eclipse? Also, how does it generalize to different types of refactorings, other than EXTRACT METHOD? At the time, we simply claimed that Eclipse’s EXTRACT METHOD had a user interface that is typical of most refactoring tools. We had some confidence

in this statement because we had painstakingly looked at every refactoring tool, at least a dozen at a time. In subsequent research, we found this process tedious and the argument somewhat unconvincing. Thus, we have found it useful to create a generalized model of how most refactoring tools work, then made the argument for generalizability based on congruence with the model. Moreover, we have found that the model is also useful for creating new, innovative user interfaces to refactoring tools.

In the next section, I discuss the model in detail (Section 2), and compare it to existing models (Section 3). I then discuss how the model can be used effectively (Section 4). My main contribution is the model itself, a model that I have found useful and believe other researchers and toolsmiths will find useful as well.

2. The Model

Figure 1 shows a model of how programmers use conventional refactoring tools, in the style of the Refactoring Browser [11]. Let me explain each step in turn, then explain the transitions between steps.

2.1 Steps in the Model

Finding code to be refactored is the Identify step, often called finding code smells [3]. This is often a step that programmers perform manually.

Typically you Select code in an editor, but it could also be done in derivative code representation (such as Eclipse’s Outline View, Quick Fixes, or O’Connor and colleagues’ graph representation [9]). Code can also be selected semi-automatically, such as by specifying queries in iXj [1].

To Initiate a refactoring tools is to choose the desired refactoring. Initiation can be done by choosing the desired refactoring name from a menu or with a hotkey.

To Configure a refactoring, you choose some options for the desired transformation, such as when you move a method, whether you want to delete the original method. This is often achieved with a dialog box.

To Execute a refactoring is to indicate to the refactoring tool that you are ready for the transformation to be applied, often by clicking an “OK” button.

To Interpret Results, you determine whether the refactoring that was applied was the refactoring that you desired.

sponds to the Identify, Interpret Results, and Execute steps of our refactoring model, respectively.

4. Using the Model

It is easy to argue that the model does not capture how every refactoring tool is used. But to make such an argument is to miss the point of the model in the first place; it is precisely the ways that it can be changed that makes it useful. Indeed, as Petre has observed, innovation does not simply spring from the minds of insightful individuals; instead, it happens as deliberate methods are employed by designers, such as the systematic variation of constraints [10]. It is my hope that the model, combined with the following methods, will help toolsmiths systematically explore the design space of refactoring tools.

Here I identify 6 different methods for modifying the model, methods that afford several advantages. For each method, I sketch how the method is used, an existing example of what the method entails (when possible), and a new example of what the method entails.

4.1 Enrichment

One way to use the model is to enrich steps by analyzing how one step in the model might be improved by adding something to it.

For example, you might notice that there is little support for identifying opportunities for refactoring in most development environments (the identify step). Using the enrichment method on the model could entail the creation of a “smell detector,” such as JCosmo [2]. The advantage of such a tool would be to automate a process that is sometimes difficult for programmers to perform manually [6].

Taking another example, we might see how we could enrich the recursive refactor and clean up steps. I know of no current support for these steps. However, you can imagine a tool that maintains a stack of refactorings for you. Suppose a refactoring tool invocation (again, say EXTRACT METHOD) does not work out as you intended because you need to perform EXTRACT LOCAL VARIABLE first. If so, you can push that *refactoring object* onto a stack without applying it, perform your EXTRACT LOCAL VARIABLE refactoring, and finally, automatically pop the refactoring object off the stack and apply it to your code. The advantage is that you would not have to remember what or how you were refactoring in the first place; the tool would remember for you.

4.2 Depletion

The opposite of enrichment is depletion, meaning removing something from a step, making it optional, or removing a step entirely.

Suppose that you eliminate the configuration step. Then you would have, for example, Eclipse’s more streamlined refactoring tools, which perform the refactoring immediately. The advantage is that you do not have to wade through a dialog just to perform a simple refactoring.

To take another example, suppose you depleted the interpret error step by building a tool that did not show errors that would cause the code to stop compiling. Thus, a programmer would be allowed to perform a refactoring that introduced compilation errors. The advantage to this is two-fold: first, I would argue that programmers sometimes want to break code for some higher purpose, and second, programmers already know how to fix compilation errors, so having them fix compilation errors should be easier than fixing unfamiliar refactoring tool errors.

4.3 Reordering

Changing the order of steps in the model can be useful as well.

For example, suppose that you switched select and initiate. Black and I did this in our refactoring cues tool [7], so that you chose what refactoring you want before selecting code. The advantage to initiating the refactoring first is that you can select multiple program elements for refactoring at one time, something that is more difficult if you select first, then initiate.

As another example, suppose you executed first, then configured the refactoring. Essentially, you would change your code with default settings, then change how the refactoring was performed. For example, you could EXTRACT METHOD first, then choose where you want the method inserted or what the parameter names would be (both Eclipse and Refactor! can do variants of this). The advantage is that interpreting what the refactoring tool did would be more incremental, and thus, I would argue, easier for the programmer to understand.

4.4 Parallelization

While the refactoring process is generally linear, it may also pay off to think about what would happen if you were to parallelize some steps in the process.

Taking our refactoring cues tool as an example again, we parallelized select and configure; once the refactoring is chosen, the programmer can configure using a non-modal palette next to the editor, or select code to be refactored in the editor itself. The advantage is allowing for implicit flexibility when using the tool.

To take another example, suppose that you made interpret results parallel with configure. Then, as you changed options in your refactoring, you could immediately see the results. The advantage of this immediate feedback is that you would not have to go back and forth between configure and interpret results to see the effect of your changes.

4.5 Splitting

One step may usefully be replaced by several steps via splitting.

For example, suppose that you split the initiation step done with a system menu (the menu at the top of the IDE) into two steps, by first filtering the list of applicable refac-

torings, then initiating one of those refactorings. Thus, you have context menus, which are filtered based on what is selected in the editor.

To take another example, suppose you split the interpret results step into two steps, one before configuration and one after. The advantage would be that you could determine whether you wanted to perform any configuration at all, or if the default options were sufficient.

4.6 Merging

On the other hand, one may take several steps and merge them into one.

For example, you might merge initiate and configure together usefully. Suppose that you gestured to initiate a refactoring, such as gesturing up with the mouse to initiate PULL UP METHOD [7]. Suppose further that the distance that you gestured up indicates how far up the inheritance hierarchy the method should go. So if you gesture up, say, less than one centimetre, the method would go into the direct superclass, and if you gestured further, it would go into other superclasses. The advantage would be that you could specify some configuration parameters while initiating the tool.

5. Conclusion

Refactoring tools have the potential to go far beyond what the original toolsmiths intended, but toolsmiths must consider alternative designs. In this paper, I have presented a model for how people use refactoring tools, a model that I have argued can be employed to generate new designs for refactoring tools. I have found this model useful in my own research; I hope that other researchers will find it useful as well.

Acknowledgments

Thanks to Andrew P. Black, who helped develop the model discussed here. Thanks to the National Science Foundation for partially funding this research under grant CCF-0520346.

References

- [1] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM. doi: doi.acm.org/10.1145/1240624.1240715.
- [2] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society.
- [3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [4] Apple Inc. Xcode refactoring guide, 2009. Technical Document, http://developer.apple.com/DOCUMENTATION/DeveloperTools/Conceptual/XcodeRefactoring/Xcode_Refactoring.pdf.
- [5] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, pages 576–585, Washington, DC, USA, 2002. IEEE Computer Society.
- [6] Mika V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 287–296, November 2005. doi: 10.1109/ISESE.2005.1541837.
- [7] Emerson Murphy-Hill and Andrew P. Black. High velocity refactorings in Eclipse. In *ETX '07: Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange*, pages 1–5, New York, NY, USA, 2007. ACM. doi: doi.acm.org/10.1145/1328279.1328280.
- [8] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, NY, USA, 2008. ACM. doi: doi.acm.org/10.1145/1368088.1368146.
- [9] Alexis O'Connor, Macneil Shonle, and William Griswold. Star diagram with automated refactorings for Eclipse. In *ETX '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, New York, NY, USA, 2005. ACM. doi: doi.acm.org/10.1145/1117696.1117700.
- [10] Marian Petre. How expert engineering teams use disciplines of innovation. *Design Studies*, 25(5):477 – 493, 2004. ISSN 0142-694X.
- [11] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997. ISSN 1074-3227.
- [12] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. *European Conference on Software Maintenance and Reengineering*, 0:91 – 100, 2003. doi: 10.1109/CSMR.2003.1192416.