Refactoring Tools: Fitness for Purpose

Emerson Murphy-Hill and Andrew P. Black, *Portland State University*

Programmers don't use refactoring tools as much as they could because the tools don't always align with "floss refactoring," a recommended refactoring tactic.

Refactoring is the process of changing software's structure while preserving its external behavior. William Opdyke and Ralph Johnson introduced the term in 1990,[1] and Martin Fowler popularized it nine years later in his book,[2] but programmers have practiced refactoring for as long as they've been writing programs. Fowler's book is largely a catalog of structural changes that programmers have observed repeatedly in various languages and application domains. Research confirms that refactoring has become mainstream. For example, Zhenchang Xing and Eleni Stroulia recently studied the Eclipse code base's version history and found that refactoring was "a frequent practice, and it involves a variety of restructuring types, ranging from simple element renamings and moves to substantial reorganizations of the containment and inheritance hierarchies."[3] Gail Murphy, Mik Kersten, and Leah Findlater studied 41 programmers using the Eclipse environment; they found that every programmer used at least one refactoring tool.[4]

Indeed, refactoring tools can improve the speed and accuracy with which developers create and maintain software—but only if they're used. In practice, tools seem not to be used as much as they could be. This appears to be because they don't align with the way most programmers refactor. On the basis of our research, we've developed five principles to help programmers choose appropriate refactoring tools and toolsmiths design products that fit programmers' purposes.

A refactoring example

Some refactorings make localized changes to a program, while others make more global changes. As an example of a localized change, when you perform Fowler's Inline Temp refactoring, you replace each occurrence of a temporary variable with its value. Taking a method from `java.lang.Long`,

```
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= –128 && l <= 127) { // will cache
      return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

we might apply the Inline Temp refactoring to the variable `offset`. Here is the result:

```
public static Long valueOf(long l) {
    if (l >= –128 && l <= 127) { // will cache
      return LongCache.cache[(int)l + 128];
    }
    return new Long(l);
}
```

The inverse operation, in which we take the second of these methods and introduce a new temporary variable to represent 128, is also a refactoring, which Fowler calls Introduce Explaining Variable. Whether the code is better with or without the temporary variable depends on the context. The first version would be better if you were about to change the code so that `offset` appeared a second time; the second version might be better if you pre-

fer more concise code. So, whether a refactoring improves your code depends on the context: you must still exercise good judgment.

Refactoring is an important technique because it helps you make semantic changes to your program. Let's look at a larger, more global refactoring as an example. Suppose that you want to read and write to a video stream using java.io. Figure 1 shows the relevant existing classes in darker boxes at the top. Unfortunately, this class hierarchy confounds two concerns: the direction of the stream (input or output) and the kind of storage it runs over (file or byte array). It would be difficult to add video streaming to the original java.io because you would have to add two new classes, VideoInputStream and VideoOutputStream, shown in the grayed boxes at the top of Figure 1. You would probably be forced to duplicate code between these two classes because their functionality would be similar.

Fortunately, we can separate these concerns by applying Fowler's Tease Apart Inheritance refactoring to produce the two hierarchies shown in darker boxes at the bottom of Figure 1. It's easier to add video streaming in the refactored version. All you must do is add a class VideoStorage as a subclass of Storage, as shown in the grayed box at the bottom of Figure 1. Because it enables software change, "Refactoring helps you develop code more quickly" (p. 57).[2]

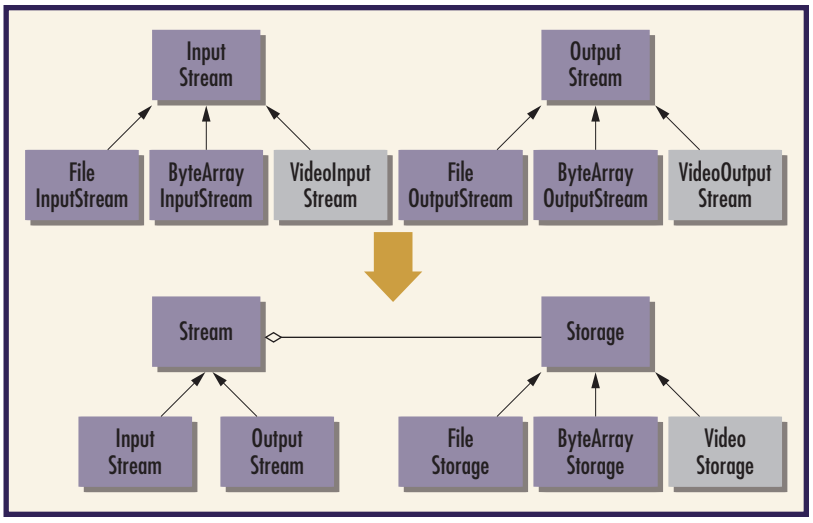


Figure 1. A Stream class hierarchy in java.io (top, darker boxes) and a refactored version of the same hierarchy (bottom, darker boxes). The grayed boxes show an equivalent change in each version.

Refactoring tools

Refactoring tools automate refactorings that you would otherwise perform with an editor. Many popular development environments for a variety of languages now include refactoring tools—for example, Eclipse (http://eclipse.org), Microsoft Visual Studio (http://msdn.microsoft.com/vstudio), Xcode (http://developer.apple.com/tools/xcode), and Squeak (http://squeak.org). You can find a more extensive list at http://refactoring.com/tools.html.

Let's see how we might use the Eclipse refactoring tools to refactor some code from java.lang.Float. First, we choose the code we want refactored, typically by selecting it in an editor. In this example, we'll choose the conditional expression in an if statement, shown in Figure 2, that checks to make sure that f is in subnormal form. Suppose we want to put this condition into its own method so that we can give it an intention-revealing name and reuse it elsewhere in the Float class. After selecting the expression, we choose the desired refactoring from a menu, which in this case is labeled "Extract Method . . ." (see Figure 2).

The menu selection starts the refactoring tool, which brings up a dialog, shown in Figure 3, asking us to supply configuration options. We must provide a name for the new method: we'll call it isSubnormal. We can also select some other options. We then have the choice of clicking either OK, which would perform the refactoring immediately, or Preview. The Preview page in Figure 4 shows the differences between the original code and the refactored version. If we like what we see, we can click OK to have the tool apply the transformation.

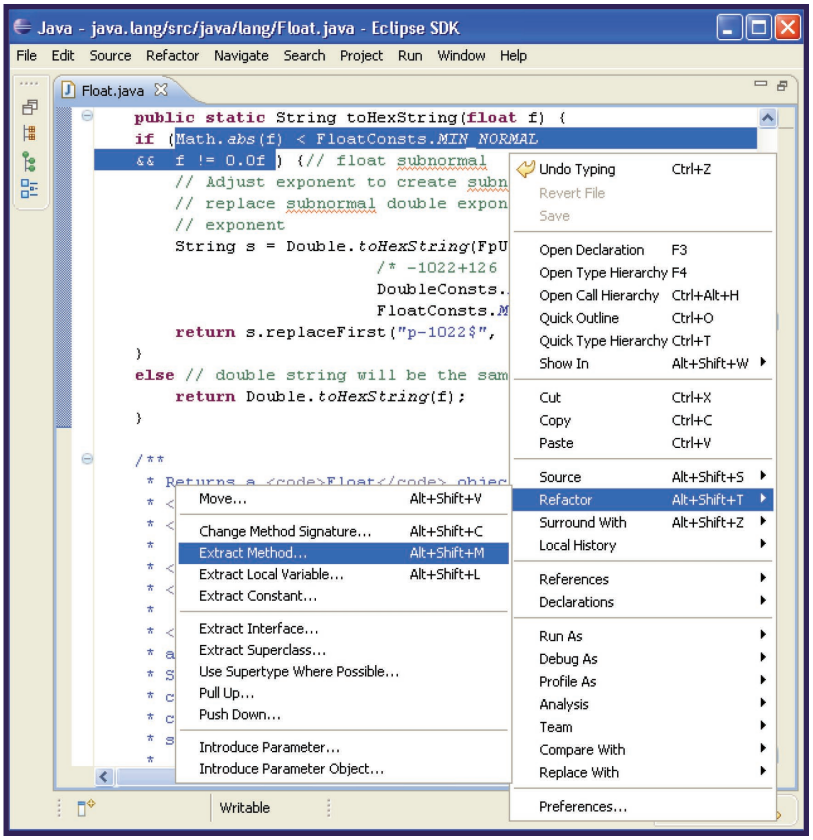


Figure 2. Selected code to be refactored in Eclipse, along with a context menu. The next step is to select "Extract Method . . ." from the Refactor pulldown menu.

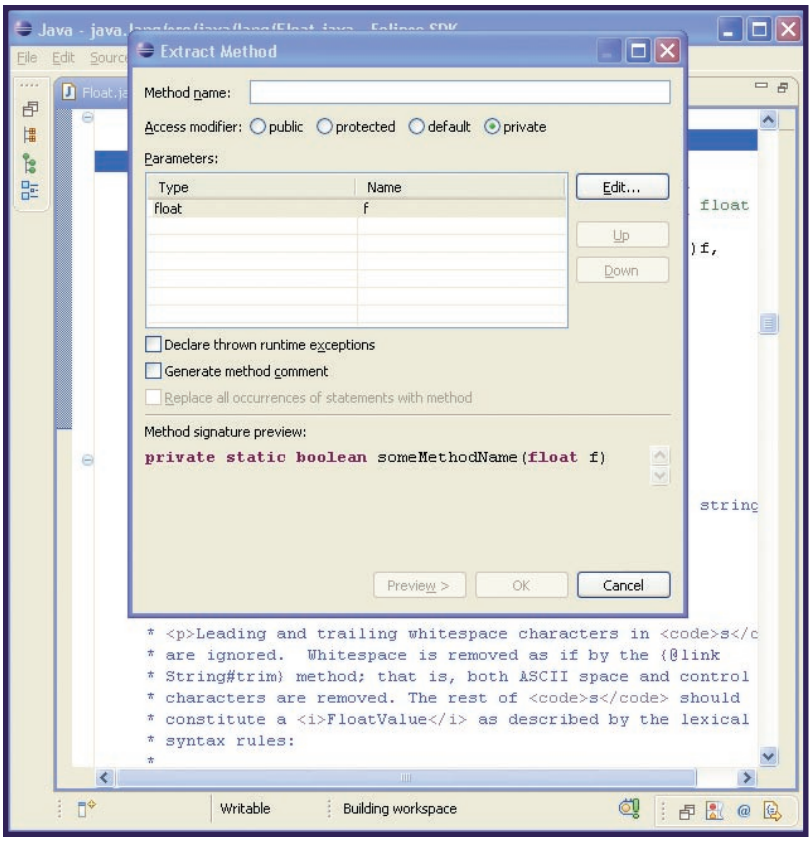
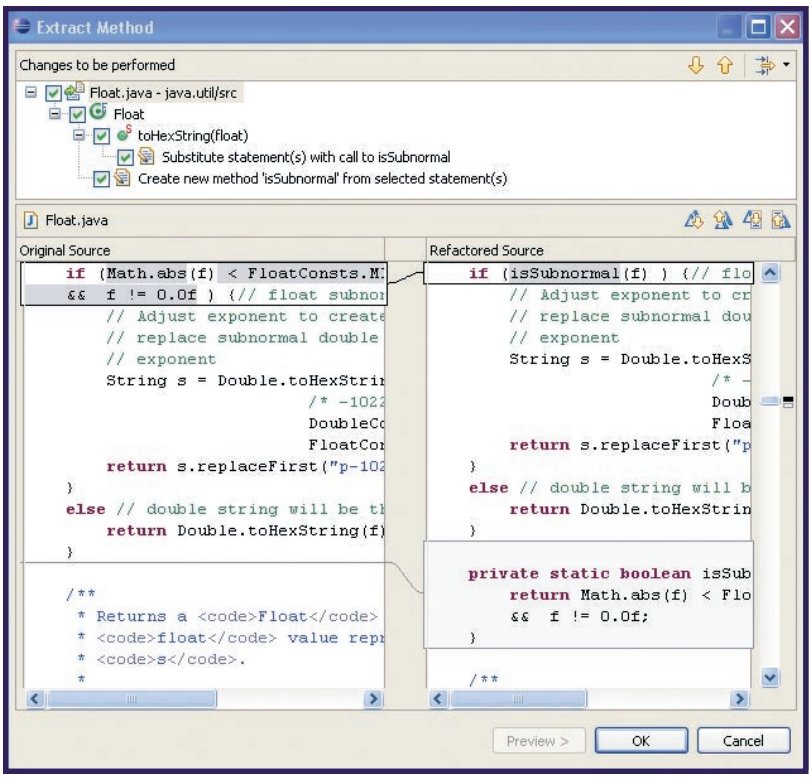


Figure 3. The Extract Method configuration dialog box. The dialog asks us to enter information. The next step is to type isSubnormal into the "Method name" text box, after which the Preview and OK buttons become active.



The tool then returns us to the editor, where we can resume our previous task.

Of course, we could have performed the same refactoring by hand, using the editor to make a new method called isSubnormal, then cutting and pasting the desired expression into the new method, and editing the if statement so that it uses the new method name. However, using a refactoring tool can have two advantages:

- The tool is less likely to make a mistake than a programmer is when refactoring by hand. In our example, the tool correctly inferred the necessary argument and return types for the newly created method; it also deduced that the method should be static. When refactoring by hand, you can easily make a mistake on such details.
- The tool is faster than refactoring by hand. Doing it by hand, we would have to take time to make sure that we got the details right, whereas a tool can make the transformation almost instantly. Furthermore, refactorings that affect many locations throughout the source code, such as renaming a class, can take a long time to perform manually and happen almost instantaneously with a tool.

In short, refactoring tools let us program faster and with fewer mistakes—but only if we choose to use them. Unfortunately, refactoring tools aren't being used as much as they could be (see the sidebar, "The Underuse of Refactoring Tools"). Our goal is to make tools that programmers will choose to use more often.

Designing better refactoring tools

Better, more usable refactoring tools must fit into the way programmers refactor. Experts have recommended refactoring in small steps, interleaving refactoring and writing code. For instance, Fowler states:

> *In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts (p. 58).*[2]

Figure 4. The Preview page showing changes that will be made to the code. At the top, you can see a summary of the changes. The original code is on the left, and the refactored code is on the right. You press OK to apply the changes.

The Underuse of Refactoring Tools

Although refactoring tools offer many potential benefits, programmers appear not to use them as much as they could.[1]

Our own observations in academic settings support this conclusion. In a questionnaire administered in March 2006, we asked students in an object-oriented programming class about their use of refactoring tools.[2] Of the 16 students who participated, only two reported having used refactoring tools and, even then, only 20 and 60 percent of the time. Furthermore, between September 2006 and December 2007, of the 42 people who used Eclipse on networked college computers, only six had tried Eclipse's refactoring tools.

Professional programmers also appear not to use refactoring tools as much as they could. We surveyed 112 people at the Agile Open Northwest 2007 conference. We found that, on average, when a refactoring tool is available for a refactoring that programmers want to perform, they choose to use the tool 68 percent of the time; the rest of the time they refactor by hand. Because agile programmers are often enthusiastic about refactoring, tool use by conventional—that is, nonagile—programmers is likely to be lower.

When we compare predicted usage rates of two refactorings against the usage rates of the corresponding refactoring tools observed in the field, we find a surprising discrepancy. In a small experiment, Mika Mäntylä and Casper Lassenius showed that programmers wanted to perform Extract Method more urgently, and several-fold more often, than Rename.[3] However, Gail Murphy, Mik Kersten, and Leah Findlater's study of 41 professional software developers shows that Eclipse's Extract Method tool is used significantly less often and by fewer programmers than its Rename tool (see Figure A).[4] Comparing these two studies, we infer that some refactoring tools—the Extract Method tool in this case—might be underused.

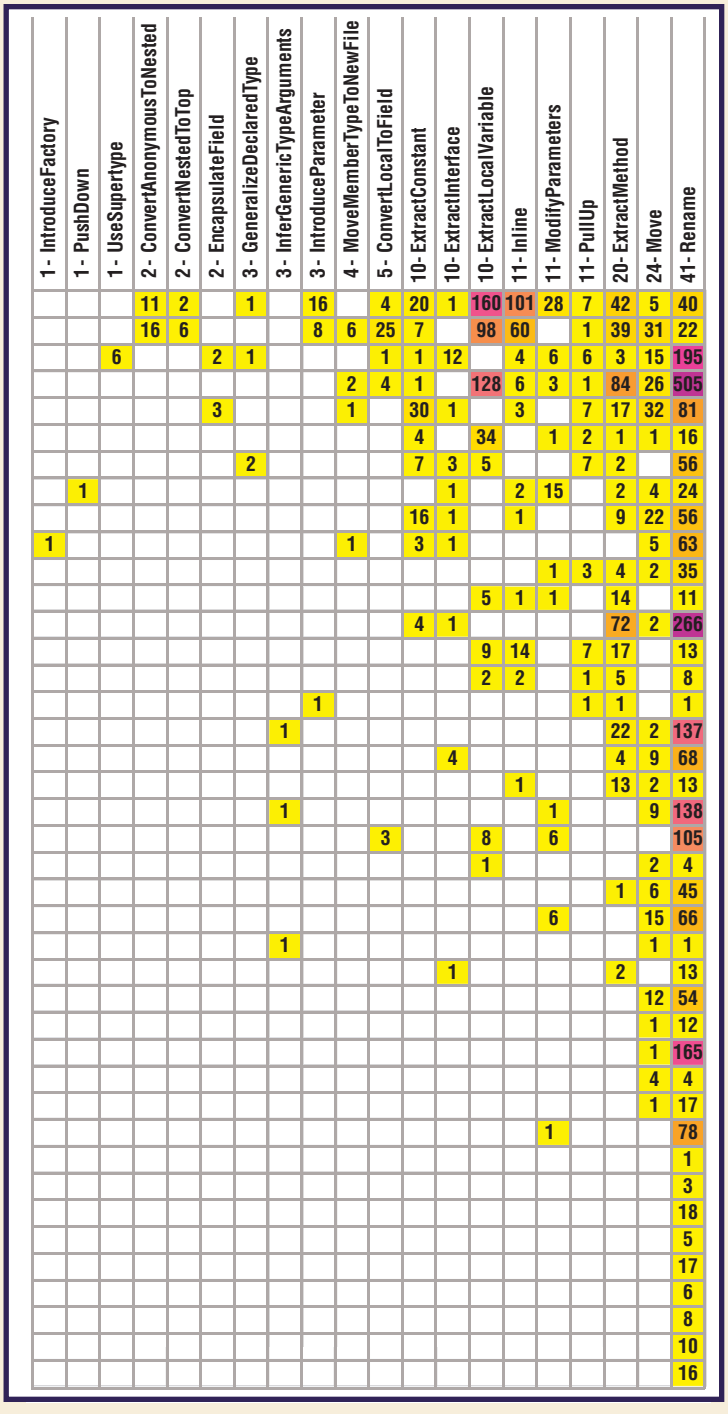


Figure A. Uses of Eclipse refactoring tools by 41 developers. Each column is labeled with the name of a tool-initiated refactoring in Eclipse and the number of programmers that used each tool. Each row represents an individual programmer. Each box is labeled by how many times that programmer used the refactoring tool. As the programmer's use of a tool increases, the box's color deepens from yellow to magenta. Data provided courtesy of Gail Murphy, Mik Kersten, and Leah Findlater.[4]

References

1. E. Murphy-Hill and A.P. Black, "Why Don't People Use Refactoring Tools?" *Proc. 1st ECOOP Workshop Refactoring Tools*, tech. report, Technical Univ. of Berlin, 2007, pp. 61–62.
2. E. Murphy-Hill and A.P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), IEEE CS Press, 2008, pp. 421–430.
3. M.V. Mäntylä and C. Lassenius, "Drivers for Software Refactoring Decisions," *Proc. 2006 ACM/IEEE Int'l Symp. Empirical Software Eng.* (ISESE 06), ACM Press, 2006, pp. 297–306.
4. G.C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, 2006, pp. 76–83.

Floss Refactoring vs. Root-Canal Refactoring

When we talk about refactoring tactics, we're referring to the choices you make about how to mix refactoring with your other programming tasks and how frequently you choose to refactor. To describe the tactics, we use a dental metaphor. We call one tactic *floss refactoring*. It's characterized by frequent refactoring, intermingled with other kinds of program changes. In contrast, we call the other tactic *root-canal refactoring*. It's characterized by infrequent, protracted periods of refactoring, during which programmers perform few if any other kinds of program changes. You perform floss refactoring to maintain healthy code, and you perform root-canal refactoring to correct unhealthy code.

We use the metaphor because, for many people, flossing teeth every day is a practice they know they should follow, but they sometimes put off. Neglecting to floss can lead to tooth decay, which a painful and expensive trip to the dentist for a root-canal procedure can correct. Likewise, a program that's refactored frequently and dutifully is likely to be healthier and less expensive in the long run than a program whose refactoring is deferred until a new bug can't be fixed or the next feature can't be added. Like delaying dental flossing, delaying refactoring is a decision developers might make to save time, but one that could eventually have painful consequences.

(a)

```
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == ((Long)obj).longValue();
    }
    return false;
}
```

(b)

```
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == m(obj);
    }
    return false;
}

private long m(Object obj){
    return ((Long)obj).longValue();
}
```

Figure 5. Refactoring with an X-develop editor: (a) a method in java.lang.Long, and (b) the code immediately after completion of the Extract Method refactoring. The name of the new method is m, but the cursor is positioned to facilitate an immediate Rename refactoring.

Agile consultant Jim Shore has given similar advice:

> *Avoid the temptation to stop work and refactor for several weeks. Even the most disciplined team inadvertently takes on design debt, so eliminating debt needs to be an ongoing activity. Have your team get used to refactoring as part of their daily work.*[5]

We call this tactic *floss refactoring*, because the intent is to maintain healthy software. In contrast, *root-canal refactoring* is characterized by long, protracted periods of fixing unhealthy code (see the sidebar, "Floss Refactoring vs. Root-Canal Refactoring"). To be suitable for floss refactoring, a tool must support frequent bursts of refactoring interleaved with other programming activities. We propose five principles to characterize such support. The tool should let the programmer

1. choose the desired refactoring quickly,
2. switch seamlessly between program editing and refactoring,
3. view and navigate the program code while using the tool,
4. avoid providing explicit configuration information, and
5. access all the other tools normally available in the development environment while using the refactoring tool.

Unfortunately, refactoring tools don't always align with these principles; as a result, such tools can make floss refactoring cumbersome. Let's revisit our refactoring tool example in Figures 2–4 to see how these principles apply to a typical refactoring tool.

After selecting the code to refactor, we needed to choose which refactoring to perform, which we did using a menu (see Figure 2). Menus containing refactorings can be quite long and difficult to navigate—a problem that gets worse as more refactorings are added to development environments. As one respondent complained in a refactoring survey we conducted last year at the Agile Open Northwest 2007 conference, the "menu's too big sometimes, so searching [for] the refactoring takes too long." Choosing the name that most closely matches the transformation you have in your head is also a distraction because the mapping from the code change to the refactoring name isn't always obvious. So, using a menu as the mechanism to initiate a refactoring tool violates Principle 1.

Next, most refactoring tools require configuration (see Figure 3). This makes the transition between editing and refactoring particularly rough, as you must change your focus from the code to the refactoring tool. Moreover, it's difficult to choose contextually appropriate configuration information without viewing the context, and a modal configuration dialog, such as the one in Figure 3, obscures your view of the context. Furthermore, you can't proceed unless you provide the new method's name, even if you don't care what the name is. Such con-

figuration dialogs thus violate Principles 2, 3, and 4.

Before deciding whether to apply the refactoring, we had the opportunity to preview the changes in a difference viewer (see Figure 4). Although comparing your code before and after is useful, presenting the code in this way forces you to stay inside the refactoring tool, where no other tools are available. For instance, in the difference view, you can't hover over a method reference to see its documentation—something you can do in the normal editing view. So, a separate modal view for a refactoring preview violates Principle 5.

Although this discussion used the Eclipse Extract Method tool as an example, we've found similar problems with other tools. These problems make the tools less useful for floss refactoring than would otherwise be the case.

Tools for floss refactoring

Fortunately, some tools support floss refactoring well and align with our principles. Let's look at some examples.

In Eclipse, while you initiate most refactorings with a cumbersome hierarchy of menus, you can perform a Move Class refactoring simply by dragging a class icon in the Package Explorer from one package icon to another. The tool will update all references to the moved class to reflect its new location. This simple mechanism lets the refactoring tool stay out of your way. Because the drag gesture implicitly chooses the class and target package, you've already provided all the configuration information required to execute the refactoring. Because this refactoring initiation mechanism is simple and fast, it adheres to Principles 1, 2, and 4.

The X-develop environment (www.omnicore.com/xdevelop.htm) makes a significant effort to avoid modal dialog boxes for configuring its refactoring tools. For instance, X-develop performs the Extract Method refactoring without any configuration at all, as Figure 5 shows. It automatically gives the new method a name. After the refactoring is complete, you can change the name by placing the cursor over the default name and simply typing a new name. This is actually a Rename refactoring, and the tool makes sure that all references are updated appropriately. Because they stay out of your way, X-develop refactoring tools adhere to Principles 2 and 4.

To avoid modal-configuration dialogs and yet retain the flexibility to configure a refactoring, we built a tool in Eclipse called Refactoring Cues that presents configuration options nonmodally.[6] To use Refactoring Cues, you ask Eclipse to display a palette of refactorings adjacent to the program code.

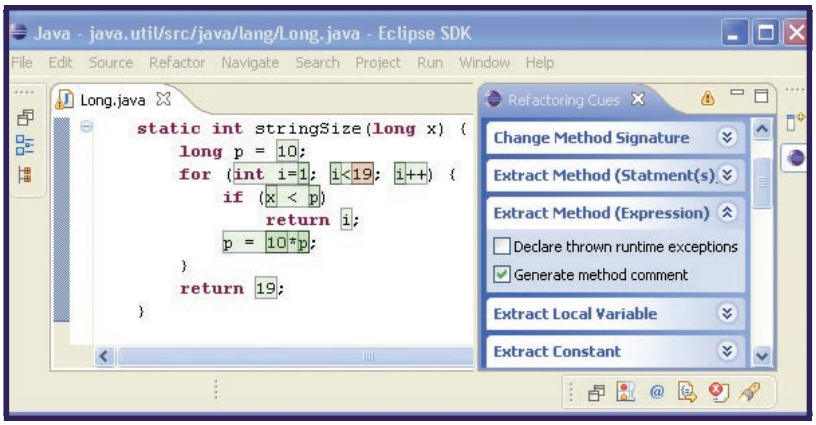


Figure 6. Refactoring Cues' nonmodal view. Users can enter configuration information (right) or select the code fragments they wish to refactor (left).

Then you select the desired refactoring in this palette and configure it there, as shown in Figure 6. Because the refactoring view isn't modal, you can use other development tools at the same time. Moreover, because you select the refactoring before the code to which you wish to apply it, the tool can help you select the code, which can otherwise be surprisingly difficult.[7] So, this tool adheres to Principles 2, 3, and 5.

Rather than displaying a refactoring preview in a separate difference view, Refactor! Pro (www.devexpress.com/Products/NET/Refactor) marks the code that a refactoring will modify with preview hints. Preview hints are editor annotations that let you investigate a refactoring's effect before you commit to it. Because you don't have to leave the editor to see the refactoring's effect, preview hints adhere to Principles 3 and 5.

Refactoring is a mainstream software engineering practice, and Fowler's book remains the best starting point for those who want to know more about it.[2] We hope the distinction we've made here between floss and root-canal refactoring and the principles we've defined for tools to support floss refactoring will serve two purposes. First, we hope the principles can help programmers choose refactoring tools that suit their daily programming tasks. If you're a programmer who usually floss refactors, you should choose tools that adhere to these principles. Second, we hope the principles can help toolsmiths build better interfaces for refactoring tools. Because floss refactoring is the recommended tactic, tools that adhere to our principles should be useful to more programmers.

The larger message for tool designers and tool users is that a good tool doesn't just help programmers do their work—it also aligns itself with the way they work.

About the Authors

Emerson Murphy-Hill is a PhD student in the Department of Computer Science at Portland State University. His research interests include human-computer interaction and software tools. He's a student member of the ACM. Contact him at emerson@cs.pdx.edu; www.cs.pdx.edu/~emerson.

Andrew P. Black is a professor in the Department of Computer Science at Portland State University. His research interests include the design of programming languages and programming environments. In addition to his academic posts, he also worked as an engineer at Digital Equipment Corp. He holds a D.Phil in computation from the University of Oxford and is a member of the ACM. Contact him at black@cs.pdx.edu; www.cs.pdx.edu/~black.

Acknowledgments

We thank Rafael Fernández-Moctezuma, Leah Findlater, Mark Jones, Bart Massey, Gail Murphy, Kal Toth, and the anonymous reviewers for their comments. We also thank our survey respondents and subjects. This material is partially based on work supported by the US National Science Foundation under grant CCF-0520346.

References

1. W.F. Opdyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Proc. 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications*, (SOOPPA 90), ACM Press, 1990.
2. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
3. Z. Xing and E. Stroulia, "Refactoring Practice: How It Is and How It Should Be Supported—An Eclipse Case Study," *Proc. 22nd IEEE Int'l Conf. Software Maintenance* (ICSM 06), IEEE CS Press, 2006, pp. 458–468.
4. G.C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, 2006, pp. 76–83.
5. J. Shore, "Design Debt," *Software Profitability Newsletter*, Feb. 2004; http://jamesshore.com/Articles/Business/Software%20Profitability%20Newsletter/Design%20Debt.html.
6. E. Murphy-Hill and A.P. Black, "High Velocity Refactorings in Eclipse," *Proc. Eclipse Technology Exchange Workshop* (eTX 07) at OOPSLA 2007, ACM Press, 2007, pp. 1–5.
7. E. Murphy-Hill and A.P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," *Proc. Int'l Conf. Software Eng.* (ICSE 08), IEEE CS Press, 2008, pp. 421–430.