2008

Refactoring Tools: Fitness for Purpose

Emerson Murphy-Hill
Portland State University

Andrew P. Black
Portland State University, black@cs.pdx.edu

Refactoring Tools: Fitness for Purpose

Emerson Murphy-Hill and Andrew P. Black
Department of Computer Science
Portland State University
Portland, Oregon

May 7, 2008

Introduction

Refactoring tools can improve the speed and accuracy with which we create and maintain software — but only if they are used. In practice, tools are not used as much as they could be; this seems to be because sometimes they do not align with the refactoring tactic preferred by the majority of programmers, a tactic we call *floss refactoring*. We propose five principles that characterize successful floss refactoring tools — principles that can help programmers to choose the most appropriate refactoring tools and also help toolsmiths to design tools that fit the programmer's purpose.

What is Refactoring?

Refactoring is the process of changing the structure of software while preserving its external behavior. The term was introduced by Opdyke and Johnson in 1990 [1], and popularized by Martin Fowler's book [2], but refactoring has been practiced for as long as programmers have been writing programs. Fowler's book is largely a catalog of refactorings; each refactoring captures a structural change that has been observed repeatedly in various languages and application domains.

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);
}
```

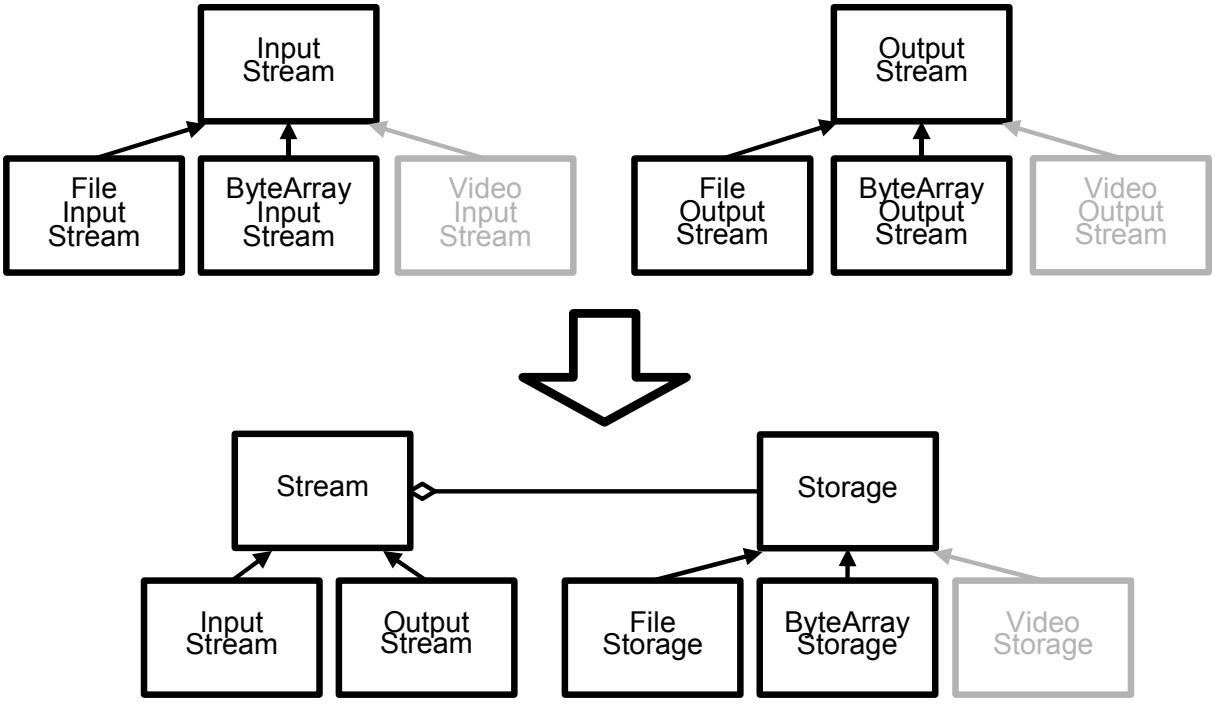

Figure 1: A stream class hierarchy in `java.io` (top, black) and a refactored version of the same hierarchy (bottom, black). In grey, an equivalent change is made in each version.

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 version of the code with or without the temporary variable is better 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 prefer more concise code. So, whether a refactoring improves your code depends on the context: you must still exercise good judgement.

Refactoring is an important technique because it helps you to make semantic changes to your program. Let's look at a larger, more global refactoring as an example. Suppose that you want the ability to read and write to a video stream using `java.io`. The relevant existing classes are shown in black at the top of Figure 1. Unfortunately, this class hierarchy confounds two concerns: the direction of the stream (input or output) and the kind of storage that the stream works 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`, as shown by the grey 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 black at the bottom of Figure 1. It's easier to add video streaming in the refactored version: all that you need do is add a class `VideoStorage` as a subclass of `Storage`, as shown by the grey box at the bottom of Figure 1. Because it enables software change, "Refactoring helps you develop code more quickly" [2, p. 57].

Not only has refactoring been prescribed by Fowler, but research suggests that many professional programmers refactor regularly. Xing and Stroulia recently studied the version history of the Eclipse code base, and "discovered that indeed refactoring is 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]. Murphy and colleagues studied 41 programmers using the Eclipse environment [4]: they found that every programmer used at least one *refactoring tool*.

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: Eclipse (http://eclipse.org), Microsoft Visual Studio (http://msdn.microsoft.com/vstudio), Xcode (http://developer.apple.com/tools/xcode), and Squeak (http://www.squeak.org) are among them. You can find a more extensive list at http://refactoring.com/tools.html.

Let's see how a the refactoring tools in Eclipse might be used 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 (Fig. 2) that checks to make sure that `f` is in subnormal form. Let's suppose that we want to put this condition into its own method so that we can give it an intention-revealing name and so that we can reuse it elsewhere in the `Float` class. After selecting the expression, we choose the desired refactoring from a menu. The refactoring that we want is labeled Extract Method (Fig. 2).

The menu selection starts the refactoring tool, which brings up a dialog asking us to supply configuration options (Fig. 3). We have to 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 OK, which would perform the refactoring immediately, or Preview >.

The preview page (Fig. 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. 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: we could have used the editor to make a new method called `isSubnormal`, 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.

1. The tool is less likely to make a mistake than is a programmer refactoring by hand. In our example, the tool correctly inferred the necessary argument and return types for the newly created method, as well as deducing that the method should be static. When refactoring by hand, you can easily make a mistake on such details.

2. 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 be quite time-consuming to perform manually. They can be accomplished almost instantly by a refactoring tool.

In short, refactoring tools allow us to program faster and with fewer mistakes — but only if we choose to use them. Unfortunately, refactoring tools are not being used as much as they could be (Sidebar 1). Our goal is to make tools that programmers will choose to use more often.

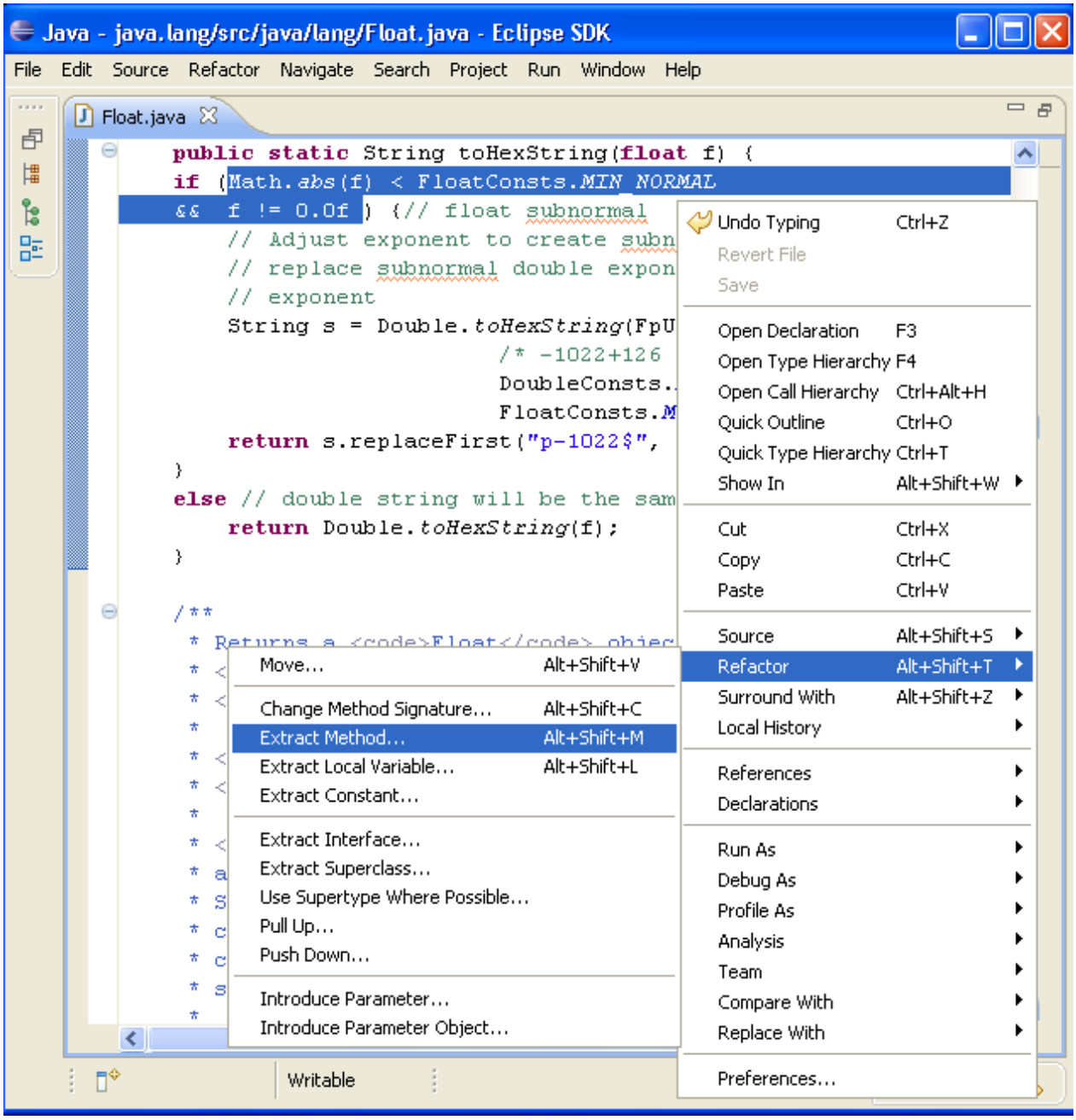

Figure 2: Selected code to be refactored in Eclipse, and a context menu. The next step is to select Extract Method... in the menu.

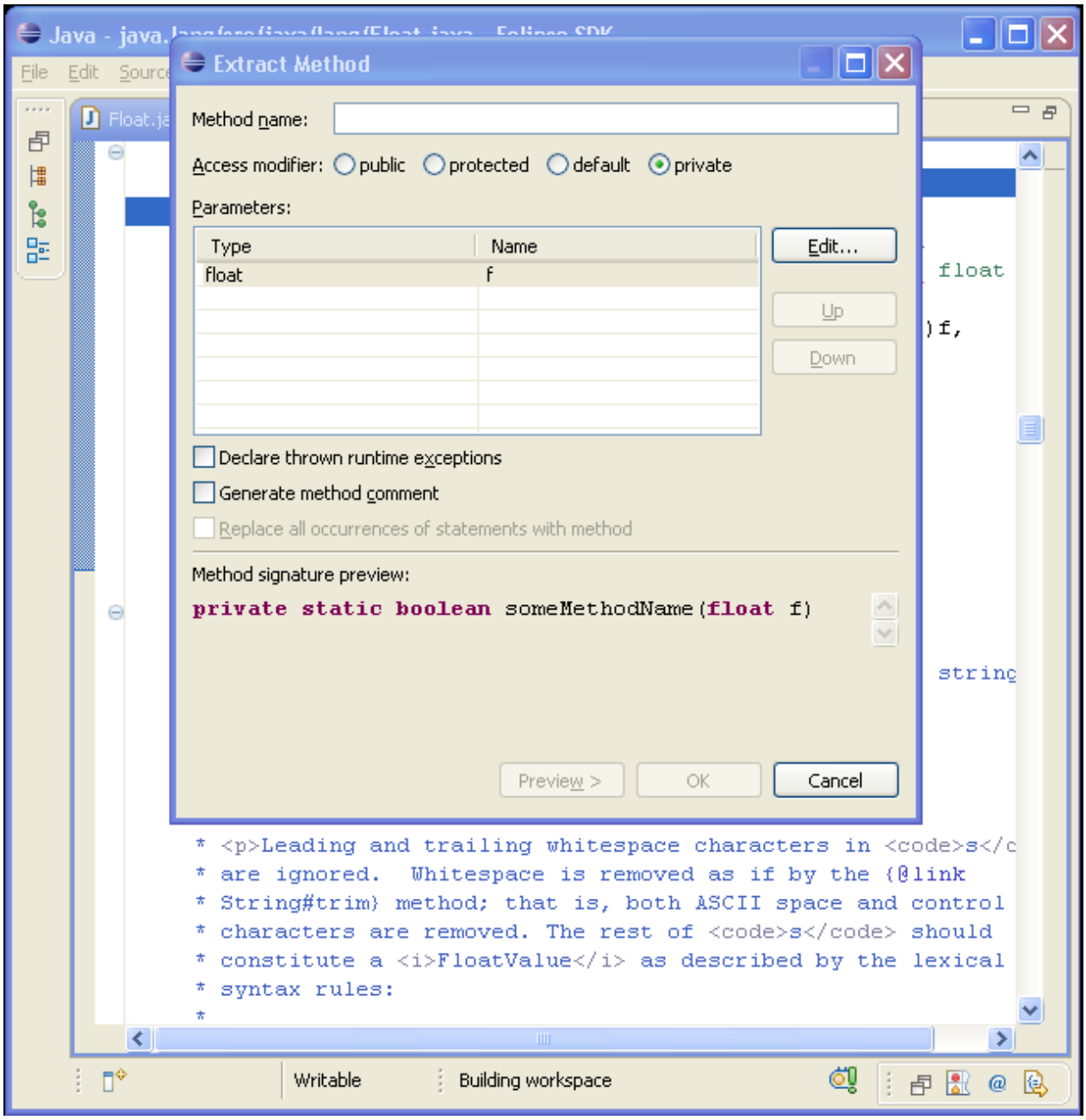

Figure 3: A configuration dialog asks us to enter information. The next step is to type "isSubnormal" into the Method name text box, after which the Preview > and OK buttons will become active.

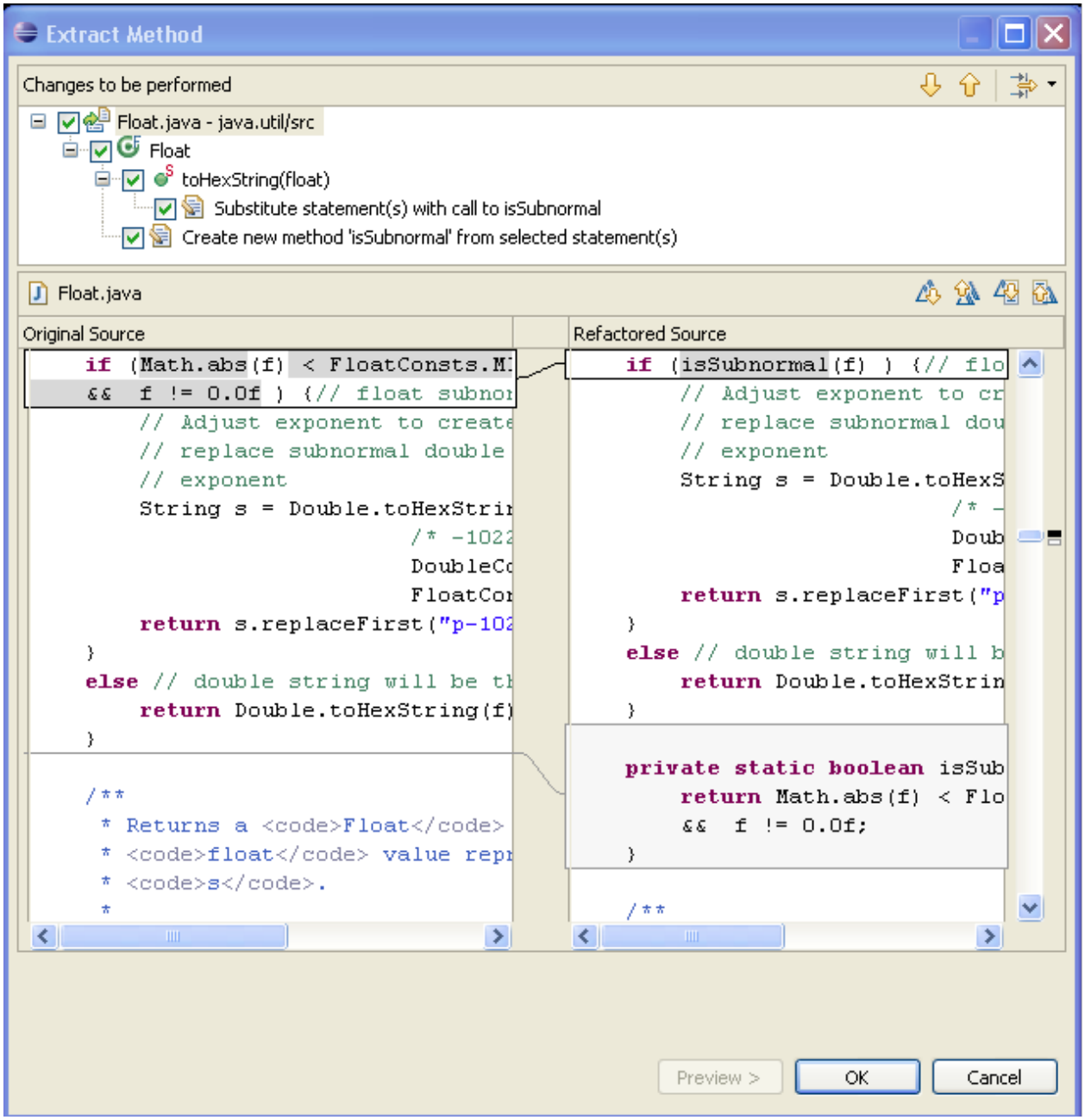

Figure 4: A preview of the changes that will be made to the code. At the top, you can see a summary of the changes. The original code is on the left, and the refactored code on the right. You press OK to have the changes applied.

Sidebar 1: The Underuse of Refactoring Tools

While there are many potential benefits to using refactoring tools, it appears that programmers don't use refactoring tools as much as they could [5].

From our own observations, it appears that few programmers in an academic setting use refactoring tools. In a questionnaire administered in March 2006, we asked students in an object-oriented programming class about their use of refactoring tools [6]. Of the 16 students who participated, only 2 reported having used refactoring tools, and even then only 20% and 60% of the time. Furthermore, between September 2006 and December 2007, of the 42 people who used Eclipse on networked college computers, only 6 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% of the time; the rest of the time they refactor by hand. Because agile programmers are often enthusiastic about refactoring, tool use by conventional (i.e., non-agile) 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, Mäntylä and Lassenius [7] showed that programmers wanted to perform Extract Method more urgently, and several-fold more often, than Rename. However, Murphy and colleagues' study [4] 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 (Fig. 5). Comparing these two studies, we infer that some refactoring tools — the Extract Method tool in this case — may be underused.

Sidebar 2: Floss Refactoring vs. Root Canal Refactoring

When we talk about refactoring tactics, we are referring to the choices that you make about how to mix refactoring with your other programing tasks, and how frequently you choose to refactor. To describe the tactics, we use a dental metaphor. We call one tactic "floss refactoring"; this is characterized by frequent refactoring, intermingled with other kinds of program changes. In contrast, we call the other tactic "root canal refactoring". This is 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 one's teeth every day is a practice they know that they should follow, but which they sometimes put off. Neglecting to floss can lead to tooth decay, which can be corrected with a painful and expensive trip to the dentist for a root canal procedure. Likewise, a program that is refactored frequently and dutifully is likely to be healthier and less expensive in the long run than a program whose refactoring is deferred until the most recent bug can't be fixed or the next feature can't be added. Like delaying dental flossing, delaying refactoring is a decision developers may make to save time, but one that may eventually have painful consequences.

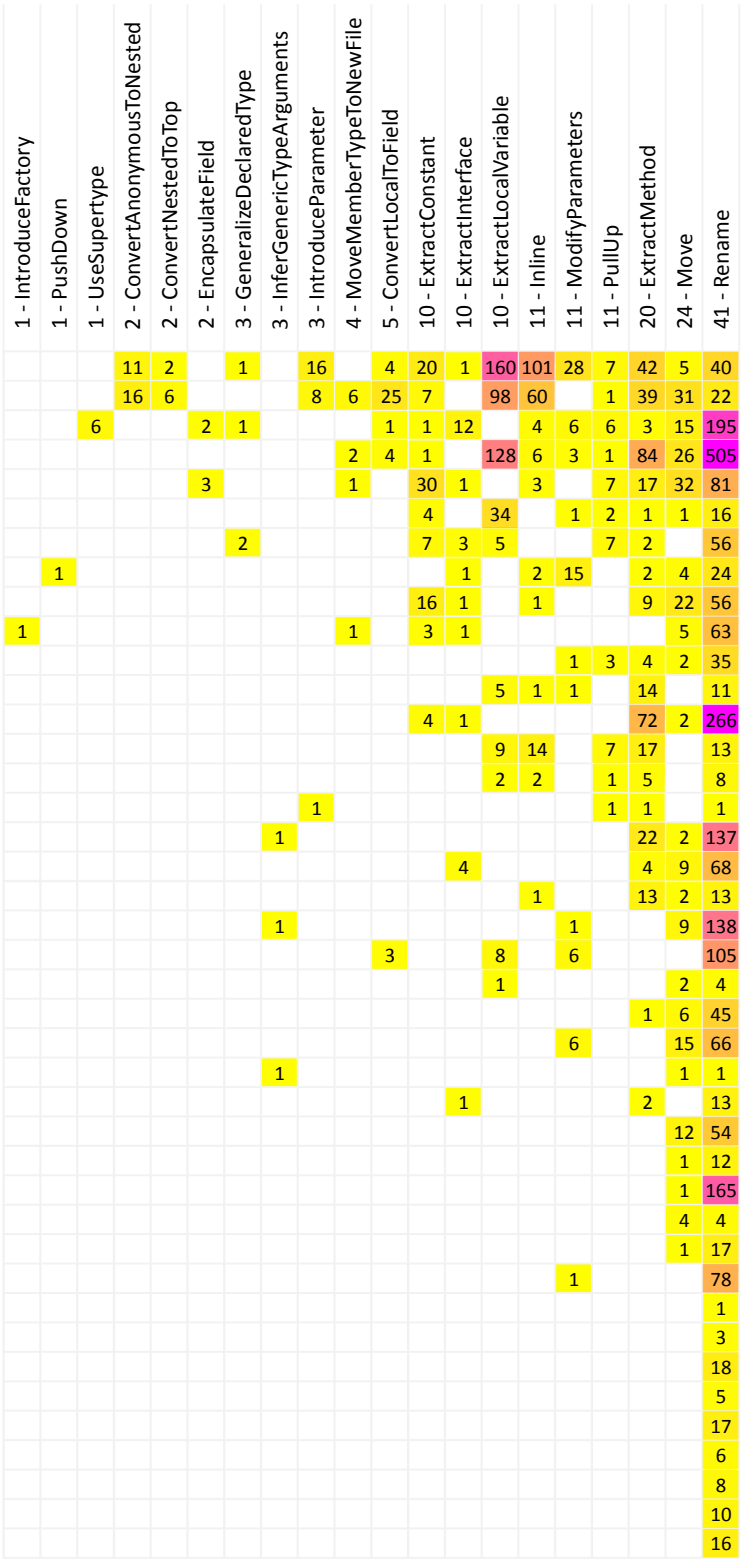

Figure 5: Uses of Eclipse refactoring tools by 41 developers. Each column is labeled with the name of a tool-initiated refactoring in Eclipse, and the number of programmers that used that tool. Each row represents an individual programmer. Each box is labeled by how many times that programmer used the refactoring tool. The darker pink the interior of a box, the more times the programmer used that tool. Data provided courtesy of Murphy and colleagues [4].

Sidebar 3: Programmers Avoid Root Canals

Two studies suggest that programmers do floss refactoring rather than root canal refactoring. Weißgerber and Diehl measured the proportion of daily code changes due to refactorings in three large open-source projects [8]. Their methodology detected simple refactorings like Rename, but not more complex ones like Extract Method. They were surprised to find that although refactorings were applied frequently, at no point did the programmers dedicate a day or more to refactoring alone, suggesting that programmers were performing little or no root canal refactoring. Likwise, in Murphy and colleagues' data [4], in no more than 10 out of 2671 programming episodes did programmers use a refactoring tool without also manually editing their program during the same episode. In other words, in less than 4% of programming episodes could we observe even the *possibility* of root canal refactoring using refactoring tools. However, it is possible that the manual edits were actually refactorings. Thus, although the data are limited and neither study is conclusive, both studies provide evidence that root canal refactoring is not practiced often.

Designing Better Refactoring Tools

Better, more usable, refactoring tools must fit into the way that 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. [2, p. 58]

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. [9]

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 unhealty code (see Sidebar 2). Not only is floss refactoring the recommended tactic, it is also more commonly practiced than root canal refactoring (see Sidebar 3).

If a tool is be suitable for floss refactoring, the tool must support frequent bursts of refactoring interleaved with other programming activities. We propose five principles to characterize such support. To enable floss refactoring, a tool should allow the programmer to:

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, floss refactoring with tools can be cumbersome. Let's revisit our refactoring tool example (Figs. 2–4) to see how these principles apply to a typical refactoring tool.

After selecting the code to be refactored, we needed to choose which refactoring to perform, which we did using a menu (Fig. 2). Menus containing refactorings can be quite long and difficult to navigate; this problem

gets worse as more refactorings are added to development environments. As one respondent complained in our Agile 2007 survey, the "menu's too big sometimes, so searching [for] the refactoring takes too long." Choosing the *name* that most closely matches the transformation that you have in your head is also a distraction: the mapping from the code change to the name is not always obvious. Thus, using a menu as the mechanism to initiate a refactoring tool violates Principle 1.

Next, most refactoring tools require configuration (Fig. 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 like that shown in Figure 3 obscures your view of the context. Furthermore, you cannot proceed unless you provide the name of the new method, even if you don't care what the name is. Thus, such configuration dialogs violate Principles 2, 3, and 4.

Before deciding whether to apply the refactoring, we were given the opportunity to preview the changes in a difference viewer (Fig. 4). While it is useful to compare your code before and after, 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 cannot hover over a method reference to see its documentation — something that can be done in the normal editing view. Thus, a separate, modal view for a refactoring preview violates Principle 5.

Although this discussion used the Eclipse Extract Method tool as an example, we have 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. All references to the moved class will be updated to reflect its new location. This simple mechanism allows the refactoring tool to stay out of your way; since the class and target package are implicitly chosen by the drag gesture, you have already provided all the configuration information required to execute the refactoring. Because of the simplicity and speed of this refactoring initiation mechanism, it adheres to Principles 1, 2, and 4.

The X-develop environment (http://www.omnicore.com/xdevelop.htm) makes a significant effort to avoid modal dialog boxes for configuring its refactoring tools. For instance, the Extract Method refactoring is performed without any configuration at all, as shown in Figure 6. Instead, the new method is given an automatic name. After the refactoring is complete, you can change the name by placing the cursor over the default name, and simply type 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 of configuring a refactoring, we have built (in Eclipse) a tool called Refactoring Cues that presents configuration options non-modally [10]. To use Refactoring Cues, you ask Eclipse to display a refactoring view adjacent to the program code. You then select the desired refactoring in this view, and also configure it there, as shown in Figure 7. Because the refactoring view is not 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 with the selection task, which can otherwise be surprisingly difficult [6]. Thus, this tool adheres to Principles 2, 3, and 5.

Rather than displaying a refactoring preview in a separate difference view, Refactor! Pro (http://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 the effect of a refactoring before you commit to it. Because you don't have to leave the editor to see the effect of a refactoring, preview hints adhere to Principles 3 and 5.

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

```
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 6: At the top, a method in `java.lang.Long` in an X-develop editor. At the bottom, the code immediately after the 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.

Conclusion

We have explained what refactoring is and why it's important in modern software development. Refactoring is mainstream; Fowler's book [2] is the best starting point for those who want to know more. Our own contribution is the distinction between floss and root canal refactoring (see Sidebar 2) and our claim that floss refactoring is and should be the dominant tactic. We think that this distinction, and the observation that some tools have been designed in a way that makes them more suitable for root canal refactoring than for floss refactoring, helps to explain why refactoring tools are not used as much as one might expect.

We hope that our principles for building floss refactoring tools will serve two purposes. First, the principles can help programmers to choose a refactoring tool that suits their daily programming tasks. If you are a programmer who usually performs floss refactoring, then you should choose tools that adhere to these principles. Second, the principles can help toolsmiths build better interfaces for refactoring tools. Because floss refactoring is the dominant tactic, tools that adhere to our principles should be useful to more programmers. This is true not just of refactoring tools themselves, but of any tool (such as a smell detector [11]) that is intended to help programmers keep their code clean as they work on it.

The larger message for tool designers and tools users is that a good tool doesn't just help programmers to do their work, but also aligns with the *way* that they work.

Acknowledgements

We thank Rafael Fernndez-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 upon work supported by the National Science Foundation under Grant No. CCF-0520346.

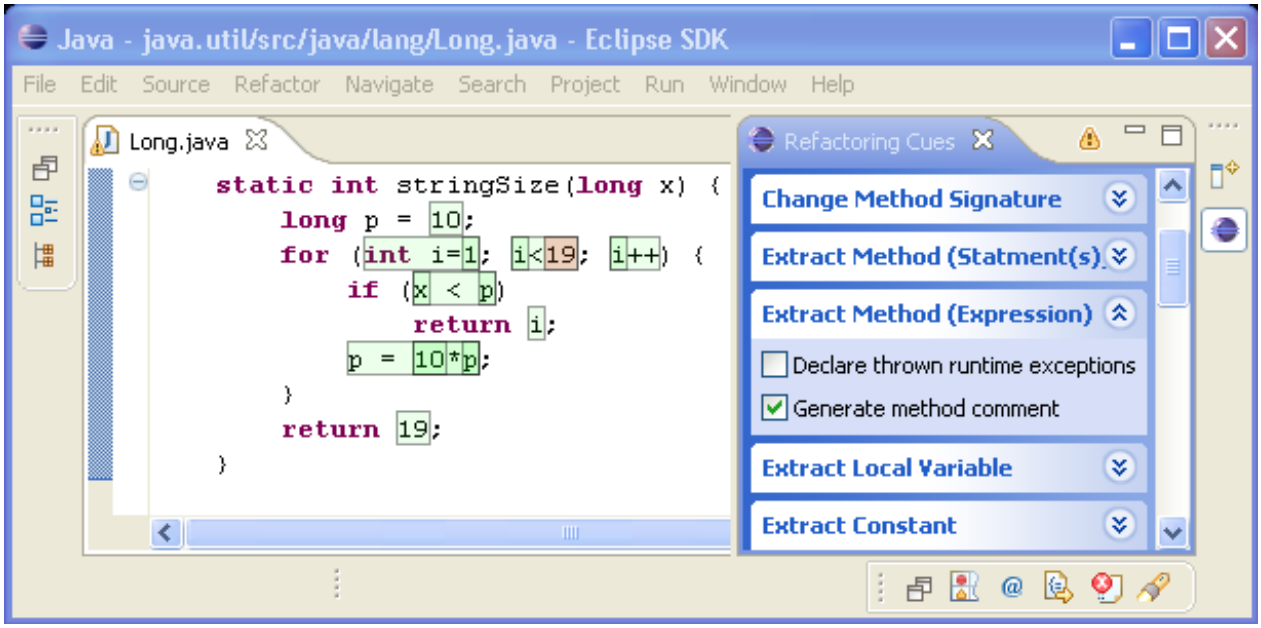

Figure 7: Refactoring Cues' non-modal view. Users can enter configuration information (right), or select the code fragment that they wish to refactor.

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; http://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; http://www.cs.pdx.edu/~black.

References

[1] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA '90: Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported — an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[5] Emerson Murphy-Hill and Andrew P. Black. Why don't people use refactoring tools? In Danny Dig and Michael Cebulla, editors, *Proceedings of the 1st ECOOP Workshop on Refactoring Tools*, pages 61–62, Berlin, Germany, July 2007. TU Berlin Technical Report 2007–08, ISSN 1436-9915. http://iv.tu-berlin.de/TechnBerichte/2007/2007-08.pdf.

[6] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings, International Conference on Software Engineering*, Leipzig, Germany, May 2008. IEEE Computer Society.

[7] Mika V. Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 297–306, New York, NY, USA, 2006. ACM.

[8] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, New York, NY, USA, 2006. ACM.

[9] James Shore. Design debt. *Software Profitability Newsletter*, February 2004. http://jamesshore.com/Articles/Business/Software%20Profitability%20Newsletter/Design%20Debt.html.

[10] Emerson Murphy-Hill and Andrew P. Black. High velocity refactorings in Eclipse. In *Eclipse '07: Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange*, pages 1–5, New York, NY, USA, October 2007. ACM.

[11] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.