pdfauthor

Making Refactoring Tools Part of the Programming Workflow

Emerson Murphy-Hill and Andrew P. Black
Department of Computer Science
Portland State University
{emerson,black}@cs.pdx.edu

To adapt software to changing requirements, many authors have advocated interspersing code modifications with refactorings to "keep the code clean". Thus, it is important that refactoring tools are fast, correct, and integrate well with programmers' primary workflow. We present data that suggest that programmers underuse refactoring tools and instead refactor by hand; this is a problem because manual refactoring is slow and error-prone.

Our data show that programmers underuse refactoring tools because the tools have poor usability: instead of fitting into programmers' workflow, the tools get in the programmers' way. We propose guidelines for improving the usability of refactoring tools, and then apply these guidelines to the design of two independent user interfaces for tool activation — interfaces that are designed to make the tool part of the programming workflow. A controlled experiment, an analytic comparison of task models, and interviews with experienced programmers combine to suggest that our new interfaces are faster, less error-prone, and more memorable than existing interfaces. We expect that the application of our guidelines to other refactoring tools will improve programmer productivity by helping programmers to focus on the refactoring that they want to accomplish, rather than on the tool that they might use to accomplish it.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments

General Terms: Design, Human factors

Additional Key Words and Phrases: environments, refactoring, tools, usability

1. INTRODUCTION

Refactoring is critical to software development because it allows programmers to adapt their software to the changing demands of customers, markets, and managers. Fowler [1999, p. xvi] characterizes refactoring as the process of changing the structure of existing program code while preserving its functionality. Examples of refactorings include renaming variables, extracting a sequence of statements and turning them into a method, moving methods between classes, and encapsulating instance variables. The term *refactoring* captures what programmers have done for a long time: ever since programs have been structured, programs have been re-structured.

Because of the importance of refactoring, many integrated development environments now include tools that semi-automate the process. For instance, in many mature environments, a programmer can tell a refactoring tool to change the name of a class, and the tool will update all references to that class throughout the whole program. Without a refactoring tool, a programmer would have either to update ref-

erences by hand, or to use a language-ignorant find-and-replace tool. In the decade since Roberts and colleagues built the first refactoring tool for Smalltalk [1997], dozens of similar tools have been deployed for a wide variety of languages.

Refactoring tools promise two benefits. First, refactoring tools promise to preserve functionality. For example, if a programmer renames a method by hand she may inadvertently forget to update some references to that method, whereas a tool can ensure that every reference is consistently renamed. Second, the tools promise to refactor faster than a programmer can refactor by hand. In the same example, a programmer may spend a significant amount of time finding and updating method references, whereas a tool can find and update those references almost instantly. Refactoring tools that deliver on these promises ought to help produce less buggy, more timely software. Unfortunately, the promise of refactoring tools has yet to be fulfilled: in practice, refactoring tools are underused.

1.1 What This Paper is About

This paper explores the underuse of refactoring tools and proposes guidelines and mechanisms that we believe will increase tool use. In Section 2, we demonstrate this underuse and link it to problems with the user interface. In Section 3, we characterize user interface problems during the activation of refactoring tools by observing when and how programmers refactor, and then deduce guidelines for future refactoring tools. In Section 4, we discuss how prior research has helped toolsmiths begin to meet these guidelines. In Section 5, we present pie menus for refactoring and refactoring cues, two improvements to the user interface of refactoring tools. In Section 6, we assess these user interface improvements with three different evaluations. We outline future work in Section 7 and summarize our conclusions in Section 8.

1.2 Contributions

Our major contributions in this paper can be summarized as follows:

- We provide a model of how programmers use conventional refactoring tools (Section 3.1).
- We demonstrate significant usability problems during activation of refactoring tools (Section 2).
- We present a set of guidelines for making refactoring tools that are easier to activate (Sections 3.4–3.6).
- We present a fast and memorable user interface for the initiation of refactoring tools (Section 5.1).
- We present a refactoring tool user interface that (a) eliminates syntax selection errors without shifting focus from the editor, (b) enables selection of multiple program elements for refactoring, and (c) allows the programmer to bypass tool configuration when it is unnecessary (Section 5.2).
- We present a study that explores the relationship between pie menu item placement and memory recall (Section 6.2), a contribution which has implications outside the domain of refactoring tools.

2. PROGRAMMERS UNDERUSE REFACTORING TOOLS

Refactoring with a tool promises to be faster and less error-prone than refactoring by hand, so if refactoring is frequent, then refactoring tools ought be used frequently. Xing and Stroulia [2006] observed that refactoring is indeed frequent, and note that "about 70% of structural changes may be due to refactorings". However, we have found that refactoring tools are *not* used frequently in practice.

2.1 Evidence for Underuse

We have collected data from three sources to demonstrate that programmers underuse refactoring tools.

The first is a study of programers using the Eclipse program development environment [Eclipse 2008] at Portland State University. Over a period of 15 months, Eclipse programmers logged refactoring activity on workstations in our college. Excluding the authors of this paper, a total of 42 people used Eclipse; only 6 used the refactoring tools.

The second is a study of 16 upper-division computer science students at the same university. Of these 16 students, only 2 self-reported that they used refactoring tools. When a tool is available for the refactoring that they want to perform, those 2 students reported using tools only 20% and 60% of the time, versus refactoring by hand [Murphy-Hill and Black 2008].

The third source is a survey that we conducted at Agile Open Northwest 2007 [Murphy-Hill and Black 2007b], a regional conference for enthusiasts of Agile programming. In contrast to the first two studies, this third study involved industrial programmers. Moreover, we can assume that they were predominantly agile programmers, and thus likely to be advocates of refactoring. During the survey, we asked 112 people, of whom 83 were programmers, to relate their experiences with refactoring tools. When refactoring tools were available, programmers said that, on average, they used them 68% of the time. Optimally, we would expect programmers to always use a refactoring tool to perform a refactoring whenever such a tool is available. This suggests that programmers are avoiding tools and instead refactoring by hand for nearly one-third of their refactorings. We suspect that programmers who are less enthusiastic about refactoring would rate their tool usage even lower.

2.2 Why Underuse is a Problem

Although these surveys provide us with no more than an estimate of refactoring tool usage, they indicate *underuse*: programmers are not using refactoring tools as much as they could. This is missed opportunity. Because a large portion of program changes are refactorings, as Xing and Stroulia have found [2006], and because only a small portion of those refactorings are performed with tools, as our data indicate, it follows that a large number of refactorings are being performed by hand, unnecessarily slowing down software development and increasing the potential for new bugs.

2.3 The User Interface: a Significant Cause of Underuse

We believe that problems with the user interface are a significant cause of the underuse of refactoring tools. Evidence comes from two sources:

- In our Agile survey, we asked programmers why they chose not to use refactoring tools when they were available. The most popular of the multi-choice responses suggested user interface problems: *the tools are not flexible* ($n = 44$), *I do not understand how to use them* ($n = 26$), and *I can refactor faster by hand than with a tool* ($n = 24$). The least popular responses indicated technical problems: *the tool might not be correct* ($n = 13$), *the tool will probably mutilate my code* ($n = 7$), and *the tool takes too long to run on a large code base* ($n = 2$).
- When we observed programmers using the EXTRACT METHOD refactoring tool in Eclipse, usability problems occurred frequently, while there were few technical problems [Murphy-Hill and Black 2008]. For instance, programmers often had difficulty understanding why the refactoring tool could not perform a refactoring whose precondition was not satisfied — a user interface problem.

The success of a refactoring tool depends on the quality of its interface more than does the success of other software tools, such as compilers. Even though a compiler may have terrible error messages, the programmer is dependent on the compiler and will develop strategies to cope with the poor interface. Not so with refactoring tools. The programmer is not dependent on a refactoring tool, so if the user interface does not help her do her job efficiently, she may reject the tool and simply refactor by hand.

3. TOOLBUILDING GUIDELINES BASED ON HOW PROGRAMMERS REFACTOR

To better understand *why* current user interfaces are problematic, we looked closely at how programmers refactor.

3.1 A Model of How Programmers Use Refactoring Tools

Figure 1 shows our model of how programmers use conventional refactoring tools. We started by examining Mealy and colleagues' 4-step model [2007], Kataoka and colleagues' 3-step model [2002], Fowler's description of small refactorings [1999], and Lippert's description of large refactorings [2004]. We expanded these simpler models into this new model by adding finer-grained steps, and the possibility of a recursive workflow, based our own observations of programmers refactoring [Murphy-Hill and Black 2008]. We have found this model useful for reasoning about how programmers use refactoring tools and improving the basic usability of those tools. However, while this model is meant to cover the most common refactoring tools, new tools are not compelled to follow it; indeed, as we will show in Section 5, reordering or eliminating some of the steps can provide benefits.

Let us explain the model by applying it to a simple refactoring. The programmer begins by finding code that should be refactored (the Identify step). Then, the programmer tells the tool which program element to refactor (Select), often by selecting code in an editor. The programmer initiates the refactoring tool (Initiate), often by choosing the desired refactoring from a menu. The programmer then gives the tool some configuration information (Configure), such as by typing a new name

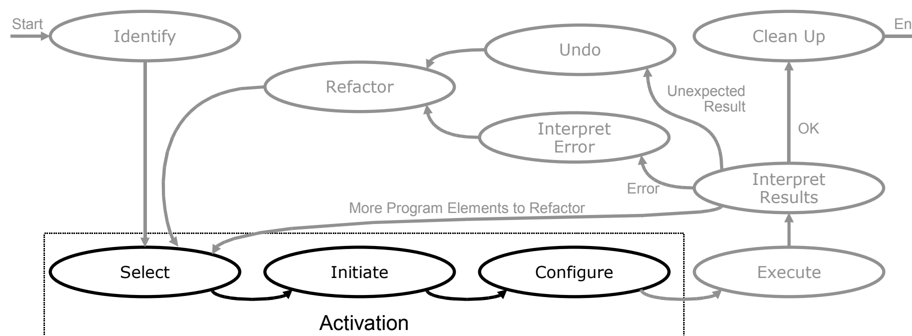


Fig. 1. A model of how programmers use conventional refactoring tools. The 3 steps in the "Activation" box are the focus of this paper.

into a dialog box. The programmer signals the tool to actually transform the program (Execute), often by clicking an "OK" button in the dialog. The programmer interprets the results of the transformation to make sure that the tool performed the desired refactoring. Finally, the programmer may choose to perform some clean-up refactorings.

The model also captures the more complicated situation in which the programmer is presented with a violated precondition, encounters a transformation that she did not expect, or decides to refactor several other program elements. When a precondition is violated, the programmer typically must interpret an error message and choose an appropriate course of action (Interpret Error). When an unexpected result is encountered, the programmer will revert the program to its original state (Undo). The programmer may recursively perform a sub-refactoring (Refactor) in order to make the desired refactoring successful. When the programmer wants to refactor several program elements at once, such as renaming several related variables, she must repeat the Select, Initiate, Configure, and Execute steps.

This model is a generalization: it describes how refactoring tools are typically used, but particular programmers and specific tools may diverge from it in three ways. First, different tools provide different levels of support at each step. For instance, only a few tools help identify candidates for refactoring. Second, although the model defines a recursive refactoring strategy, a linear refactoring strategy is certainly possible. In a linear refactoring strategy, a programmer performs sub-refactorings first and avoids errors before they occur. We don't favor a strictly linear refactoring strategy because it requires more foresight regarding what the tool *will* do, which we consider an unnecessary burden on the programmer. We have also observed that the need for such foresight can sometimes lead programmers to avoid refactoring altogether [Murphy-Hill and Black 2008]. Third, some steps can be reordered or skipped entirely; for example, some tools provide a refactoring preview so that the programmer may interpret the results of a refactoring before it is executed.

In this paper, we focus on the activation stage, which is composed of three steps: selection (Section 3.4), initiation (Section 3.5), and configuration (Section 3.6). The other steps in the refactoring process are interesting as well, such as how a tool can help to determine that code should be refactored [Simon et al. 2001], how a tool communicates a violated refactoring precondition [Murphy-Hill and Black 2008], and how a tool can support executing refactorings when developers collaborate [Balaban et al. 2005]. Nevertheless, we will restrict ourselves to activation for three reasons. First, activation is the core activity with modern refactoring tools: it is the only piece of the tool-supported refactoring process that is common to each and every refactoring. Second, little attention has been paid to activation in previous research. Third, activation of modern refactoring tools poses significant usability problems, as we will demonstrate in Sections 3.4 through 3.6.

In the remainder of this section, we summarize observations made by other researchers and present new ones; we then derive some guidelines based on these observations. At each step during activation, we analyze how a programmer interacts with a typical refactoring tool, and concurrently analyze how the tool hinders the programmer. Along the way, we use our analyses to suggest guidelines for future refactoring tools.

3.2 The Value of Usability Guidelines Specific to Refactoring

The guidelines in Sections 3.4 through 3.6 are a refined version of broader usability guidelines, where the refinement criteria are derived from observations about refactoring tool usage. For instance, Shneiderman [1987, p. 72] proposes the guideline "Minimal input actions by user," which is a more general form of our guideline "Allow the programmer to bypass viewing or entering unnecessary configuration information" (Section 3.6). General guidelines are just that, general, so interface designers and toolsmiths may find it difficult to apply them to specific user interfaces. Thus, Schneiderman states, general guidelines "must be interpreted, refined, and extended for each environment" [Shneiderman 1987, p. 62], and so in this paper we interpret, refine, and extend them for refactoring tools.

3.3 A Running Example

We use a small program restructuring as a running example. Suppose that we start with this code in a video game application:

```
characterCount = 4 + 1;
```

We realize that we want to say explicitly that `4` is the number of ghosts in our game. So we would like to extract `4` into a local variable with a meaningful name, to produce this code:

```
int ghostCount = 4;
...
characterCount = ghostCount + 1;
```

Now let's examine how this small refactoring would be performed using a refactoring tool for Java in Eclipse [2008], a popular program development environment with mature refactoring tools.

3.4 The Selection Step

We first tell the refactoring tool *what* it should transform by selecting a program element. In our example, we select 4, typically using the keyboard or mouse in an editor:

```
characterCount = 4 + 1;
```

Although this step appears quite simple, it can be problematic in the context of a refactoring tool for a number of reasons.

We have previously demonstrated [Murphy-Hill and Black 2008] that selection can be error-prone. For example, programmers sometimes mis-select complex program elements, such as **if** statements that span many lines. In the same paper we made three recommendations to guide the development of selection tools:

◇ Programmers can normally select code quickly: do not slow them down by adding overhead to the common case.

◇ Help the programmer to overcome unfamiliar or unusual code formatting.

◇ Allow the programmer to select code in a manner specific to the task she is performing.

In retrospect, we realize that a fourth, unstated recommendation guided the development of a tool called Box View [Murphy-Hill and Black 2008]. Box View, a tool for selecting program statements, almost completely eliminated selection errors by forgoing character-by-character selection, and instead presenting the program as a nested structure of selectable boxes corresponding to program elements. The unstated guideline is as follows:

◇ Eliminate as many selection errors as possible by separating character-based selection from program-element selection: the only possible selections should be those that are valid inputs to the refactoring tool.

This guideline raises the issue that the programmer may not know what selections are valid inputs to the tool, especially if she has not previously used that tool. For instance, if a programmer wants to move a method up into a superclass (the PULL UP refactoring), it may not be clear whether she should select the whole method declaration, the method heading, or the method name. To make it clear what to select to perform a refactoring, we propose that the interface

◇ Make explicit what the refactoring tool expects as input before selection begins.

Finally, selecting multiple program elements for refactoring is usually impossible with present-day refactoring tools. For instance, we know of no existing tool that can extract both 4 and 1 into variables in a single step. However, it seems that programmers would benefit from this ability. Murphy and colleagues [2006] conducted a large-scale study of 41 professional developers; according to data from that study, nearly half of all refactorings occur within 1 minute of another refactoring of the same kind (for example, two back-to-back EXTRACT LOCAL VARIABLE refactorings). Based on these observations, we propose the following guideline:

◇ Allow the programmer to select several program elements at once.

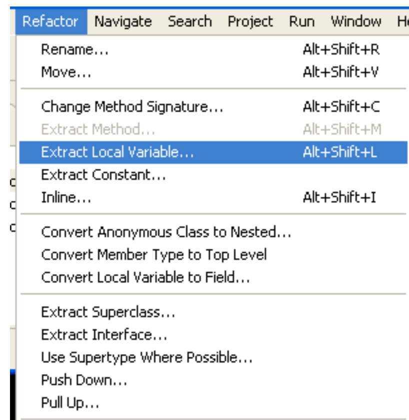


Fig. 2. Initializing a refactoring from a system menu in Eclipse, with hotkeys displayed for some refactorings.

3.5 The Initiation Step

Once a program element has been selected, the programmer must indicate which refactoring tool to initiate, usually with a linear menu or a hotkey (Fig. 2). A linear menu is a system or context menu where items appear in a linear list, possibly with submenus. Hotkeys are invoked by a combination of keypresses on a keyboard.

System menus can be slow, because the user must take significant time to navigate *to* them, then *through* them, and finally *point* at the desired menu item [Callahan et al. 1988]. In the case of refactoring, the problem is worsened as many development environments offer a very long list of possible refactorings from which the programmer must choose. As one programmer complained in our Agile survey, the refactoring "menu is too big sometimes, so searching [for] the refactoring takes too long." Context menus, menus that contain different items depending on the context under the cursor, alleviate the problem of navigating *to* the menu, but worsen the problem of navigating *through* the menu because they must be shared with a large number of non-refactoring items.

The slowness of linear menus is a problem during refactoring because using a refactoring tool *must* be fast. Programmers usually refactor frequently to maintain healthy code, a process we call "floss refactoring"[1] [Murphy-Hill and Black 2007b]. The process of adding new features and maintaining code is interleaved with floss refactoring. Floss refactoring is always performed to enable a larger programming goal and never for its own sake. The frequent switching between refactoring and other programming activities during floss refactoring dictates that the time to switch between the activities must be as small as possible. Thus, the speed at which a refactoring can be initiated is critical, inspiring the following guideline:

◇ Be fast enough to avoid distracting the programmer from the primary programming task, to which refactoring is subordinate.

[1] We contrast this with "root canal refactoring," where programs are refactored after they have become unhealthy, during a dedicated and protracted refactoring period.

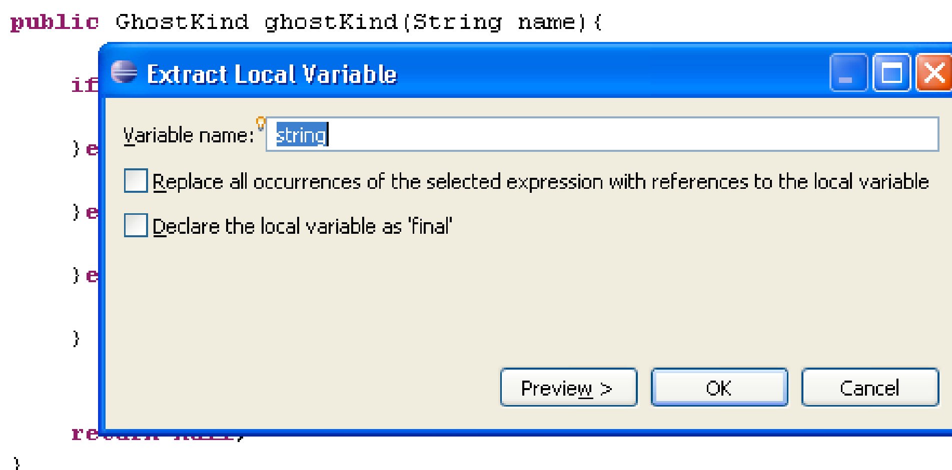


Fig. 3. Configuration gets in the way: an Eclipse configuration wizard obscures program code.

Using a hotkey might seem to be an ideal way of speeding up the initiation of refactoring tools. However, hotkeys are difficult to remember [Grossman et al. 2007], especially for refactorings. One reason is that the mapping from a code transformation to a hotkey is often indirect. A programmer must take the structural transformation that she wants to perform (such as "take this expression and assign it to a temporary variable"), recall the *name* of that refactoring (Extract Local Variable), and finally map that name to a contrived hotkey (remembering that "Alt+Shift" means refactor, and that "L" is for "Local"). The task is especially difficult because the names of the refactorings are themselves somewhat capricious[2], and because the refactorings must compete with the hundreds of other hotkey commands in the development environment. In Murphy and colleagues' study [2006], the median number of hotkeys programmers used for refactoring was just 2; the maximum number of hotkeys used by any programmer was 5. This suggests that programmers do not often use hotkeys for initiating refactoring tools. Thus:

◇ Do not rely on the names of refactorings for fast tool initiation, but rather use an initiation mechanism that more closely matches how programmers think about refactoring, such as structurally or spatially.

3.6 The Configuration Step

Once a tool has been initiated, programmers typically have to configure it. In our example, we want to choose a name for the new local variable that we are introducing. Most refactoring tools use a modal dialog box or multi-page "wizard" for configuration. As an illustration, the Eclipse Extract Local Variable configuration interface is shown in Figure 3.

A modal dialog box compels programmers to configure the refactoring by not letting them do anything else until the configuration is finished. However, it appears

[2]For example, Eclipse'sExtract Local Variable is called Introduce Explaining Variable in Fowler's book [1999].

that programmers frequently perform very little configuration. A small survey that we performed at a Portland Java Users' Group meeting (September 19, 2007) lends some support to this claim; of the 6 programmers who use refactoring tools, the median estimated percentage of refactorings that would require configuration, beyond providing a new program element name, was just 25%. In Figure 3, this means that 75% of the time, the programmer needs to interact only with the "Variable name" text box. In other refactorings tools, such as Eclipse's EXTRACT METHOD tool, the un-utilized user interface space is even larger. Thus, users should have the ability to bypass configuration, giving us this guideline:

◇ Allow the programmer to bypass viewing or entering unnecessary configuration information.

A modal configuration dialog also forces the refactoring tool to become the programmer's primary focus. This is undesirable because refactoring is a subsidiary task in floss refactoring, as we explained in Section 3.5. Moreover, disrupting a programmer's focus may cause visual disorientation, a problematic issue in modern development environments [de Alwis and Murphy 2006]. Thus:

◇ Avoid interrupting the programmer's primary programming task with configuration information. In most cases, this task is editing the program.

Furthermore, the configuration dialog may obstruct the view of the source code, which may hinder the programmer's ability to configure the refactoring. In our example, it may be difficult to choose a contextually meaningful variable name without viewing the context, which may require viewing the source code in the editor, or accessing other programming tools. Thus:

◇ Configuring a refactoring should not obstruct a programmer's access to other tools.

4. PRIOR WORK ON REFACTORING TOOL USER INTERFACES

Prior research has yielded several notable approaches to improving the user interfaces of refactoring tools, specifically for selection, initiation, and configuration, and also for refactoring tools in general.

4.1 Improvements to Code Selection

Several tools have been introduced to improve how code can be selected for refactoring. One such improvement is an outline, such as the Eclipse Outline View [Eclipse 2008] that displays class members as a hierarchy of icons in a view adjacent to program text. Thus, to select a class member, a programmer need only select its representative icon, an operation that is more error resistant than code selection in an editor. Icon selection also allows several members to be selected for repeated refactoring. However, this technique has two chief disadvantages. First, it does not apply to program elements that don't have icons, such as statements for EXTRACT METHOD or expressions for EXTRACT LOCAL VARIABLE. Second, it requires the programmer to change contexts from the editor to a seperate view, which may be slower than selecting a member name in the editor.

In iXj [Boshernitsan et al. 2007], a programmer can select several program elements at once using a program transformation language based on examples. Code

is selected by first mouse-selecting an example in the editor and then generalizing that example using the mouse in a separate view, an interaction that may not be sufficiently fast for floss refactoring [Murphy-Hill and Black 2007b]. Furthermore, while iXj can assist the programmer in selecting some kinds of program elements, such as expressions, it does not help select every kind of program element that a programmer might want to restructure, such as a list of statements.

In a previous study, we introduced two tools to help programmers select code appropriate for refactoring: Selection Assist and Box View [Murphy-Hill and Black 2008]. However, neither tool follows every guideline in Section 3.4. Furthermore, while programmers using these tools demonstrated significant improvements in both speed and accuracy of selection, the tools were limited to just one refactoring.

In summary, while previous approaches have improved the accuracy of selection and added the ability to select several program elements at once, they cannot be generalized to the selection of every kind of program element.

4.2 Improvements to Refactoring Tool Initiation

Researchers and toolmakers have demonstrated some effective alternatives to hotkeys and menus in the domain of development environments. Burnett and Gottfried [1998] have shown that people using gestures to issue programming commands can program faster and with fewer errors. In Eclipse and other environments, a programmer can sometimes drag an icon (perhaps representing a class) to some other program icon (representing a package) to invoke a refactoring, but only for Move refactorings such as MOVE CLASS. Development environments such as Netbeans [NetBeans 2008] and IntelliJ IDEA [Smartdec 2008] provide plugins for gesture recognition, but we have not seen any research into how such plugins might be used as a memorable and effective mechanism for initiating refactorings.

While we have claimed that linear menus are slow, Callahan and colleagues [1988] have shown that pie menus are a significantly faster alternative. Pie menus are similar to context menus except that items appear around a circle rather than in a vertical list (Figure 4). Pie menus are faster than linear menus because items are selected only by direction, rather than by direction and distance. Even faster are marking menus, an extension of pie menus that allow the user to "mouse ahead" by gesturing in the direction of the desired menu item, even if that item has not yet been drawn on the screen. After frequent use of a menu item, users utilize "muscle memory" to remember the direction in menu items are located [Kurtenbach and Buxton 1993]. We know of one development environment, Fabrik [Ludolph et al. 1988], that used a radial menu to initiate common programming commands, such as connecting and disconnecting visual program components. However, radial menus do not appear to have made inroads into modern integrated development environments. In Section 6.1, we discuss two empirical studies evaluating pie menus, marking menus, and linear menus.

4.3 Improvements to Refactoring Configuration

To avoid the distraction of configuring a refactoring, some development environments try to avoid configuration altogether. In the X-develop environment [Omnicore 2008], the programmer supplies the name of a newly-created program element *after* the refactoring has been performed, using a linked in-line RENAME. For ex-

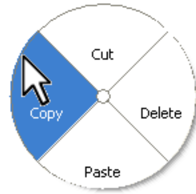


Fig. 4. A simple pie menu.

```
characterCount = 4 + 1;
```

```
int v = 4;
characterCount = v + 1;
```

```
int ghostC = 4;
characterCount = ghostC + 1;
```

Fig. 5. The user begins refactoring by selecting the 4 in X-develop, as usual (top). After initating Extract Local Variable (middle), the user types "ghostC" (bottom), using a linked in-line Rename refactoring tool.

ample, when an Extract Local Variable refactoring is performed, the tool extracts an expression into a new variable, which is given a default name (Fig. 5, middle). The programmer may change the name by typing (Fig. 5, middle and bottom), which causes all references to that variable to be updated in real time by the refactoring tool. We feel that this is a natural way to bypass configuration, but is not applicable to all refactorings. Eclipse [Eclipse 2008] and IntelliJ IDEA [JetBrains 2008] also use this technique for some refactorings.

4.4 Research Towards Improved Usability in Refactoring Tools

After creating one of the first refactoring tools, Roberts [1999] noted that the tool "was rarely used, even by ourselves." In response, the tool was improved by following three usability guidelines: speed in program transformation, support for undoing refactorings, and tight integration with the development environment. It appears that most refactoring tools since then have heeded these guidelines, yet new usability issues have sprung up, as we have discovered (Section 3). In this paper we will seek to build on Roberts' guidelines to improve usability of refactoring tools.

More recently, Mealy and colleagues [2007] distilled general usability guidelines into specific requirements for refactoring tools. Mealy and colleagues' work is top-down (that is, concerned with producing a complete set of refactoring tool guidelines gleaned from general guidelines), while our research is bottom-up (that is, concerned with producing guidelines derived from specific bottlenecks in the refactoring process). However, both kinds of guideline can be difficult to interpret. Usability guidelines have been shown to be too vague to be useful to a user-interface designer; instead, guidelines should be "developed primarily to *complement* toolkits and interactive examples" [Tetzlaff and Schwartz 1991]. Thus, as Mealy and

colleagues develop a proof-of-concept refactoring tool, we expect that their requirements will solidify. Likewise, in the next section, we will pair our guidelines with two new refactoring tool interfaces.

5. TWO NEW REFACTORING TOOLS CONFORMING TO OUR GUIDELINES

Thus far, we have demonstrated that programmers underuse refactoring tools, linked that underuse to user interface problems with those tools during activation, and presented guidelines for building more usable refactoring tools. Now, to complement our guidelines and to make them more concrete, we present two tool improvements that follow the guidelines. A positive evaluation of these tools indicates that the guidelines may be valid (Section 6.4).

The two tools were designed to implement different guidelines for different steps during activation. Both tools were written as plugins to the Eclipse development environment [Eclipse 2008] and provide new user interfaces to the existing refactoring engines. We encourage the reader to watch our screencasts (http://multiview.cs.pdx.edu/refactoring/activation), which provide a short but illuminating look at how the tools work in practice. An in-depth technical discussion of the tools, and a discussion of how the two tools could work together effectively, can be found in a paper presented at the Eclipse Technology Exchange [Murphy-Hill and Black 2007a].

5.1 Pie Menus for Refactoring

The first tool uses pie menus[3] to address the initiation step of the refactoring process, discussed in Section 3. Our implementation of pie menus is based on the SATIN implementation [Hong and Landay 2006]. Pie menus are not new; our contribution is the application of pie menus to refactoring and the rules for placing refactorings in a particular direction.

We use pie menus for refactoring so that programmers can initiate refactorings quickly; once initiated, the transformation is executed immediately, rather than displaying a configuration dialog. Pie menus are invoked in the same way as standard context menus, with a dedicated mouse button or hotkey. Pie menus are context sensitive, so different menus correspond to different types of program elements (such as statements, method names, or variable names). For example, after a programmer selects a method name and invokes a pie menu, she gets the menu shown in Figure 6.

The advantage of pie menus for refactoring is that the structural nature of many refactorings can be mapped onto the placement of the labels around the pie menu. We use three rules to determine the placement of a refactoring. First, directional refactorings are placed in their respective direction, so, for example, PULL UP is on top. Second, refactorings that are inverses are placed on opposite sides of the menu, so, for example, INLINE METHOD is opposite EXTRACT METHOD. Third, refactorings that are conceptually similar are placed in the same location in different contexts, so, for example, INLINE METHOD and INLINE LOCAL VARIABLE

[3] Technically, we have implemented the *marking menu* variant of pie menus: in addition to selecting the refactoring from the circular menu, marking menus allow the user to gesture to initiate a refactoring. However, for convenience we will use the name pie menus.

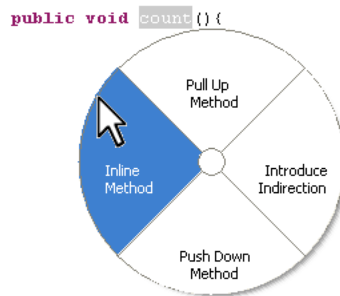


Fig. 6. A pie menu, showing the refactorings that can be applied to the selected method.

occupy the same quadrant in different menus. Refactoring appears to be one of the applications of pie menus that Callahan and colleagues say "seem to fit well into the mold of the pie menu design," for which the authors showed that pie menus have a distinct advantage over linear menus in terms of initiation speed [Callahan et al. 1988].

Moreover, we hypothesize that our placement rules helps programmers *recall* or *infer* the location of refactoring menu items, even before the programmer builds muscle memory. This is important for refactorings, because, as we have noted, most refactoring tools are currently used infrequently. This means that muscle memory would not be an effective recall mechanism for those refactorings. We validate this hypothesis with an experiment described in Section 6.

Our pie menus are restricted to four items with no submenus. The restriction increases the speed at which pie menus can be used [Kurtenbach and Buxton 1993] and reduces the complexity of user interaction. Furthermore, because some programmers prefer to use the keyboard, the restriction allows programmers to use the menus with a simple hotkey scheme (one button to bring-up the menu; a press of the up, down, left, or right arrow to invoke an item). In fact, our placement rules could be used without pie menus, in a pure hotkey scheme that might reduce the cognitive overhead of name-based refactoring hotkeys. Unfortunately, the restriction also means that certain refactorings do not appear on our pie menus, especially refactorings that we believe have no intuitive direction, such as Rename. However, this incompleteness should not be a serious problem, because programmers currently use several mechanisms for initiating different refactorings [Murphy et al. 2006]. Table I shows how each refactoring tool in Eclipse can be activated.

Pie menus for refactoring were designed to meet each of the requirements outlined in Section 3.5. In short, using pie menus to initiate refactorings gives programmers the speed of hotkeys with the low-entry barrier of linear menus. Pie menus that incorporate our placement rules were created to facilitate the transition from beginner (a programmer who uses refactoring tools infrequently) to expert (a programmer who always uses the tool when refactoring) in a way that is not possible with name-based hotkeys, linear menus, or pie menus without our placement rules.

Refactoring	Linear Menus	Hotkeys	Pie Menus
Rename	Yes	Alt+Shift+R	
Move	Yes	Alt+Shift+V	
Change Method Signature	Yes	Alt+Shift+C	
Extract Method	Yes	Alt+Shift+M	Right
Extract Local Variable	Yes	Alt+Shift+L	Right
Extract Constant	Yes		
Inline	Yes	Alt+Shift+I	Left
Convert Anonymous Class to Nested	Yes		Right
Convert Member Type to Top Level	Yes		Right
Convert Local Variable to Field	Yes		Right
Extract Superclass	Yes		
Extract Interface	Yes		
Use Supertype Where Possible	Yes		
Push Down	Yes		Bottom
Pull Up	Yes		Top
Introduce Indirection	Yes		Right
Introduce Factory	Yes		Right
Introduce Parameter Object	Yes		
Introduce Parameter	Yes		
Encapsulate Field	Yes		Right
Generalize Declared Type	Yes		
Infer Generic Type Arguments	Yes		
Convert Nested to Anonymous	No		Left
Increase Visibility	No		Right
Decrease Visibility	No		Left

Table I. How refactorings can be initiated using Eclipse 3.3 and our current implementation of pie menus, in the order in which each refactoring appears on the system menu (Fig. 2); for pie menus, the direction in which the menu item appears is shown in the third column. We implemented the last three refactorings specifically for pie menus.

5.2 Refactoring Cues

The second tool, refactoring cues, address the selection and configuration steps of the refactoring process (Section 3) by taking an "action-first" approach. This means that choosing the refactoring (the action) precedes choosing what to refactor.

We begin by initiating a refactoring by clicking on a refactoring from the adjacent, non-modal *palette* (Fig. 7, right), which displays the available refactorings. After a refactoring is initiated, two things happen simultaneously. (1) The palette item expands to reveal configuration options (Fig. 8, right) and (2) *cues* (highlighted program elements with thin borders) are drawn over program text to indicate all program elements to which the refactoring can be applied (Fig. 8, left).

We can then choose specific program elements to be refactored. To choose a program element, we can mouse click anywhere inside the overlaid cue, a technique we call *targeting*. Cues that are not targeted are colored green while those that are targeted (that is, chosen for refactoring) are colored pink. A cue with nested children can be targeted by clicking on the cue directly, or by clicking repeatedly on one of the cue's children. For example, Table II illustrates what happens when the user clicks on `4` several times.

In our example, we want to extract the number `4`, so we click on `4` once. We

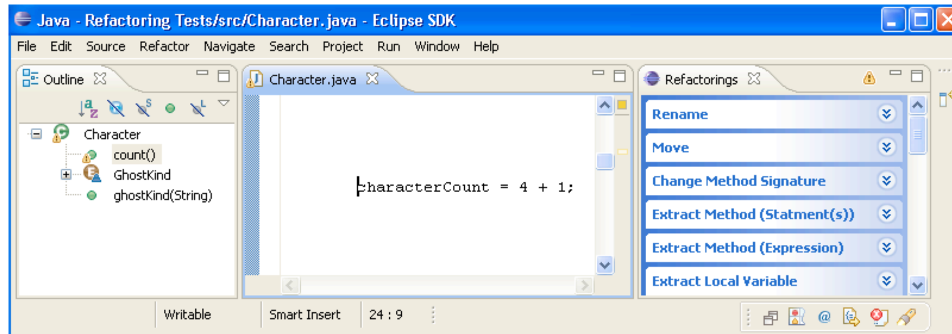


Fig. 7. The Eclipse environment. The standard Java program editor is in the center; the refactoring cues palette is on the right.

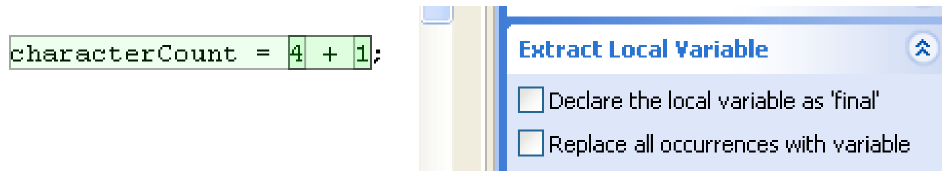


Fig. 8. The expanded refactoring cues palette, right, and program editor with overlaid cues, left.

Mouse Click	**Cue Coloration**	**Program element to be refactored**
First	`characterCount = 4 + 1;`	`4`
Second	`characterCount = 4 + 1;`	`4 + 1`
Third	`characterCount = 4 + 1;`	`characterCount = 4 + 1`
Fourth	`characterCount = 4 + 1;`	(none)

Table II. The results of clicking multiple times on the 4 in refactoring cues. A targeted cue (pink, center column) indicates the expression to be refactored (right column). If a cue is not targeted (green), a mouse click targets it (first row). If a cue or one of its ancestors is targeted, the targeted cue becomes un-targeted and its parent becomes targeted (second and third rows). Clicking within an already-targeted outermost cue un-targets all children (last row).

then press a hotkey or button to perform the transformation. At that point, the refactoring engine modifies the code and gives any newly created program elements default names. Visually, at the same time, the configuration options are hidden in the palette and the cues are removed from the editor. The result is shown in Figure 9. Finally, the cursor is positioned to allow us to change the newly created, default-named program elements with a standard Eclipse linked in-line RENAME, as shown in Figure 10.

Although we have just described one interaction with refactoring cues, the user interface allows some flexibility in the way that the programmer interacts with the tool.

- Because the palette takes up valuable screen real estate, it does not have to

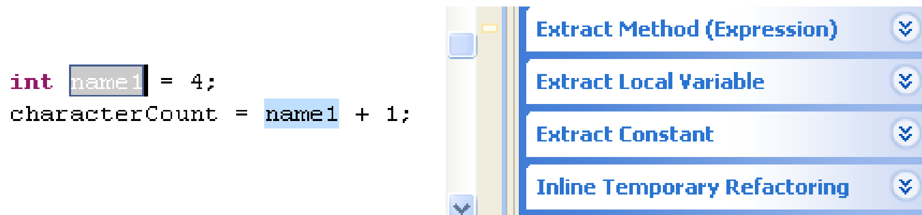


Fig. 9. Just after the EXTRACT LOCAL VARIABLE refactoring, the refactoring cues palette is collapsed (right) and the transformed code is shown (left). A linked in-line RENAME is ready to be performed, as shown by the boxes over the code, with the text selection highlighting the first occurance of `name1`.

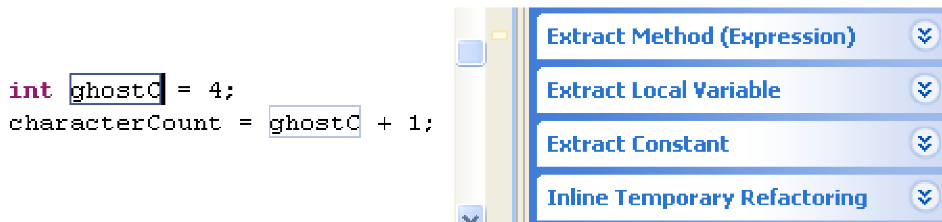


Fig. 10. The refactoring cues palette (right) and program editor with linked in-line RENAME (left).

be displayed by default. Instead, programmers can designate it as an Eclipse "Fast View," bringing it up only when they wish to perform configuration.

- In addition to using the palette as an initiation mechanism, programmers have the option of using any other mechanism, such as hotkeys, linear menus, or even pie menus.
- Because initiation happens before selection, multiple program elements are targeted in the same manner as a single program element. In our example, this means that, during the selection step, the programmer may click the `4` and then the `1`, indicating that *both* should be refactored in one transformation.
- As an alternative to clicking to target a refactoring cue, a programmer can use an ordinary text selection to target a cue or several cues (Figure 11). This technique not only reduces the effort required to target several program elements at once, it also provides backward compatibility with existing refactoring tools: any selection valid as input to a standard refactoring tool is also a valid selection for refactoring cues. Selection using the keyboard is permitted as well.

Refactoring cues were designed to meet all of the guidelines proposed in Section 3.4 and 3.6. They were designed to reduce selection errors by making all cues acceptable input to the refactoring tool. The technique of targeting program elements for refactoring using visual cues can be applied to all refactorings. Refactoring cues were also designed to be faster than standard refactoring tools because multiple elements can be refactored at once and because configuration is optional. They may reduce programmer guesswork by making explicit what is refactorable, and also by allowing the programmer to remain focused on the text of the program.

```
if(name.equals("blinky")){
    return GhostKind.BLINKY;
}else if(name.equals("pinky")){
    return GhostKind.PINKY;
}else if(name.equals("inky")){
    return GhostKind.INKY;
}else if(name.equals("clyde")){
    return GhostKind.CLYDE;
}
```

Fig. 11. Targeting several cues at once using a single selection; the programmer's selection is shown by the grey overlay.

6. EVALUATION

Here we present an overview of two previous studies, as well as three of our own studies, that suggest that our user interfaces to refactoring tools are an improvement over existing user interfaces to refactoring tools. We define a refactoring user-interface improvement as an interface that allows programmers to refactor faster, with fewer errors, or with a greater degree of programmer satisfaction.

First, we discuss previous researchers' comparisons of the speed and error rates of linear and pie menus (Section 6.1); we also describe an experiment designed to evaluate our rules for the placement of refactorings on pie menus (Section 6.2). Next, we analytically compare refactoring cues to conventional refactoring tools with respect to speed and error rate (Section 6.3). Then, we describe a survey used to gauge programmers' preferences regarding our refactoring tool improvements (Section 6.4). We provide a summary of the results of our evaluations in Section 6.5.

6.1 Previous Studies: Pie Menus vs. Linear Menus

Here we compare pie menus for refactoring to the state-of-the-practice refactoring tool initiation mechanism, which is linear menus. We do not compare pie menus against hotkeys, both because refactoring hotkeys are user-mappable and thus can be as fast as pressing a single key, and because users of hotkeys can continue to use hotkeys with our pie menus.

Two previous studies suggest that pie menus like ours are faster and less error-prone than linear context menus. Callahan and colleagues [1988] showed that the speed improvement of pie menus over linear context menus was statistically significant. Furthermore, when Callahan and colleagues' measurements are compared with Kurtenbach and Buxton's measurements [1993], marking menus are more than 3 times faster than pie menus. Callahan and colleagues also observed that pie menus are less error-prone than linear menus, but the statistical significance was marginal. These results strongly suggest that programmers should be able to use our pie menus[4] faster and with fewer errors than a linear menu of the same size.

[4]Recall that our pie menus are actually marking menus, so for our implementation, Kurtenbach and Buxton's results are better predictors than Callahan and colleagues' results.

6.2 Memorability Study: Pie Menus with and without Placement Rules

In Section 5, we noted that users of pie menus for refactoring may not become experts (that is, know *a priori* the gesture of the desired refactoring) because becoming an expert requires repeated use of a refactoring tool, and only a few refactoring tools are used frequently. We hypothesized that our placement rules help programmers recall the direction of items on a pie menu. In this experiment, we test this hypothesis in a memory recall experiment, the first experiment to explore the effect of pie menu item placement on memory recall.

6.2.1 *Methodology.* In this experiment, programmers were asked to memorize the direction (left, right, top, or bottom) of a refactoring on a pie menu. In the training phase, programmers were given a paper packet containing 9 pages. Each page contained an initial piece of code, a refactored piece of code, and a pie menu for refactoring with the associated refactoring highlighted on one of the four menu quadrants (Fig. 12, top). We told programmers to try to associate the before-and-after code with the direction on the pie menu.

In the testing phase immediately following the training phase, the programmers were given the same 9 pages, but in a different order and with the labels on the pie menus removed. We told programmers to try and recall where the refactoring appeared on the pie menu.

During the training phase, programmers were given 30 seconds to understand the refactoring from the before-and-after code, and also to remember the direction of each refactoring. We gave subjects such a short period because we wanted them to spend most of the time reading and understanding the code transformation, and only a small amount of time remembering the direction of a refactoring item. This was because few programmers in the wild are going to spend significant time memorizing the direction of items on a pie menu. Interviewing some dry-run experiment participants informally confirmed that little time was spent memorizing.

Using before-and-after code to recall the direction of a refactoring, rather than using only the name of a refactoring, gave the experiment several advantages:

- Programmers who had no knowledge of refactoring terminology could still participate in the experiment.
- Our choice of refactoring names would not confuse programmers who had experience with refactoring, but whose knowledge of terminology differed from what we chose in the experiment (see Section 3.6).
- If programmers think of a refactoring as a transformation of code and not as a name in a book or in a development environment, then the experiment more closely matches how programmers refactor in the wild.

During the testing phase, programmers were given a total of 5 minutes to recall the direction of all 9 refactorings. The bottom of Figure 12 shows an example of a recall page. Additionally, during that time programmers were asked to guess the direction of a refactoring that they had *not* seen during the training period. The refactoring to be guessed was EXTRACT LOCAL VARIABLE, which, according to our placement rules, should appear in the same direction as the CONVERT LOCAL TO FIELD, ENCAPSULATE FIELD, and INTRODUCE INDIRECTION (see Table I), three refactorings that the subjects had seen during training.

More information about this experiment, including the test materials and the resulting data set can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

6.2.2 *Subjects.* We recruited 18 programmers to participate in this experiment. In an attempt to sample a diverse population, these programmers were recruited from three sources. Seven were students from a graduate programming class on Java design patterns, six were computer-science research assistants, and five were programmers from industry. To expose the programming class to the concept of refactoring, the first author of this paper gave students a 20-minute presentation on refactoring two weeks prior to the experiment.

The only prerequisite for participating in the experiment was the ability to read Java code. Two subjects were excused from the programming class set because one did not meet this prerequisite and one did not follow directions during the experiment, leaving a total of 16 participants. With the exception of offering refreshments during the experiment, we did not compensate the participants for their time.

Within each set, each programmer was randomly assigned to one of two groups. In the experimental group, programmers were trained on pie menus that contained refactoring items placed according to our rules. In the control group, programmers were trained on pie menus that contained refactoring items placed in opposition to each of our rules.

6.2.3 *Results.* Overall, subjects in the control group could recall a median of 3 refactoring directions, while subjects in the experimental group could recall a median of 7.5 refactoring directions. The difference is statistically significant ($p = .011$, $df = 1$, $z = 2.553$ using a Kruskal-Wallis one-way analysis of variance). The results of the experiment are shown in Figure 13.

Six out of the eight subjects in the experimental group correctly guessed the direction of the refactoring that they had not seen during training. This suggests that this group of programmers were not simply recalling what they had learned, but had synthesized a mental model regarding where refactorings "should" appear.

In summary, the results of the experiment confirm the hypothesis that our placement rules help programmers to recall the direction of refactorings. We believe that this will help programmers to initiate a large number of refactorings quickly, while building muscle memory, and without having to resort to rote memorization.

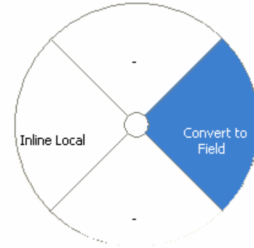
6.2.4 *Limitations.* There are three limitations of this experiment. First, subjects were asked to memorize the direction of 9 refactorings, which represent only about one-tenth of the refactorings suggested by Fowler [1999]. The effect of learning and trying to recall the full catalog of refactorings is unclear. Second, we cannot easily explain the outliers in the data set (one overperforming control group subject and two underperforming experimental group subjects). It may be the case that some programmers can easily recall the direction of a refactoring on a pie menu, regardless of their placement, and, conversely, that some programmers have difficulty recalling the direction of refactorings even when placed using our rules. Third, the experiment was conducted in such a way that it is difficult to discern which of the 3 placement rules (outlined in Section 5.1) the programmers found most intuitive.

```
class Bike extends Vehicle{

      protected double tireWidth = 2.2;//inches

      public double getTireWidthInCentimeters(){

            double cmsPerInch = 2.54;
            return cmsPerInch * tireWidth;
      }
}
```




```
class Bike extends Vehicle{

      protected double tireWidth = 2.2;//inches
      private double cmsPerInch;

      public double getTireWidthInCentimeters(){

            cmsPerInch = 2.54;
            return cmsPerInch * tireWidth;
      }
}
```

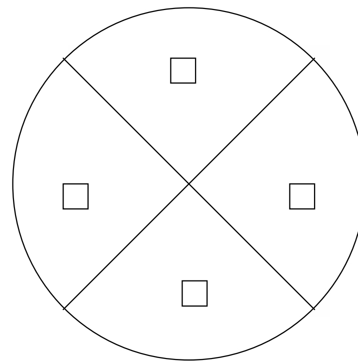
2

```
class Bike extends Vehicle{

      protected double tireWidth = 2.2;//inches

      public double getTireWidthInCentimeters(){

            double cmsPerInch = 2.54;
            return cmsPerInch * tireWidth;
      }
}
```



```
class Bike extends Vehicle{

      protected double tireWidth = 2.2;//inches
      private double cmsPerInch;

      public double getTireWidthInCentimeters(){

            cmsPerInch = 2.54;
            return cmsPerInch * tireWidth;
      }
}
```

Fig. 12. A sample training page (top) and a sample recall page (bottom). The refactorings (left, as program code before-and-after refactoring) are the same on both pages. Subjects were instructed to put a check mark in the appropriate direction on the recall page.

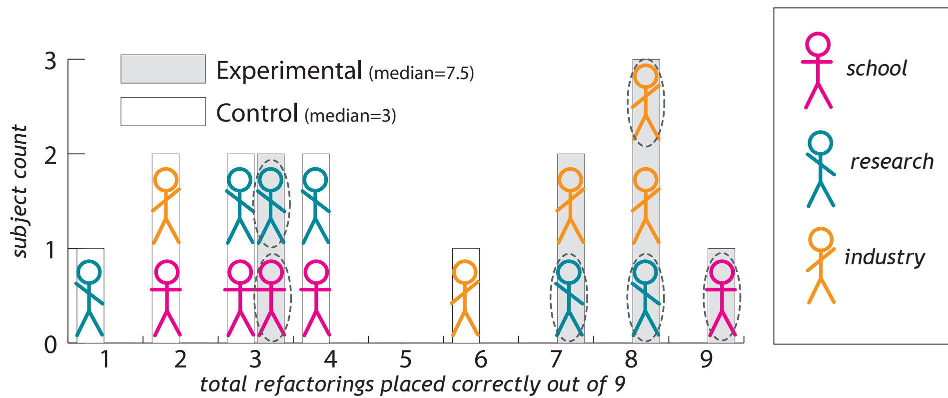


Fig. 13. A histogram of the results of the pie menu experiment. Each subject is overlaid as one stick figure. Subjects from the experimental group who correctly guessed the refactoring that they did not see during training are denoted with a dashed oval.

Method for goal: Refactor with Traditional Tool

Step 1. Select Code **Operator**: Make selection with mouse	S_{CT}
Step 2. Initiate Tool	I_{CT}
Step 3. Type element name	N_{CT}
Step 4. Enter other information in GUI	G_{CT}
Step 5. Execute	E_{CT}
Step 6. **If** More code to refactor **Then** go to **Step 1**	M_{CT}

Return with goal accomplished

Fig. 14. An NGOMSL method for conventional refactoring tools.

6.3 Analytical Study: Refactoring Cues vs. Traditional Tools

We will argue that refactoring cues are faster and less error prone than conventional refactoring tools. By a conventional refactoring tool, we mean one that is used by selecting code, initiating the refactoring, then configuring with a wizard or dialog, as described in Section 3. We will use a step-by-step analysis to compare the speed and likelihood of error using refactoring cues and conventional refactoring tools.

6.3.1 *Analysis by Stepwise Comparison.* Here we argue in a stepwise manner that refactoring cues are at least as fast and error-free as conventional refactoring tools by comparing their NGOMSL methods [John and Kieras 1996]. An NGOMSL method is a sequential list of user interface actions for achieving a goal, and so in Figures 1 and 2, we outline the major steps necessary to use each refactoring tool. To keep the methods general, we omit details for several steps, such as how one enters configuration information in each tool's GUI. To the right of each step, a capital letter and a subscript labels which tool the step belongs to. For example, E_{CT} refers to the Execution step using a Conventional Tool. The NGOMSL method

Method for goal: Refactor with Refactoring Cues

Step 1. Initiate Tool	I_{RC}
Step 2. Select Code **Operator**: Make selection with mouse	S_{RC}
Step 3. Enter other information in GUI	G_{RC}
Step 4. **If** More code to refactor **Then** go to **Step 2**	M_{RC}
Step 5. Execute	E_{RC}
Step 6. Type element name	N_{RC}

Return with goal accomplished

Fig. 15. An NGOMSL method for refactoring cues.

for conventional tools refers to the activation steps of the model shown in Figure 1.

We chose NGOMSL as a way to describe how programmers use refactoring tools because of its standardized structure, convenient and flexible notation, and ability to express high-level goals rather than individual keystrokes. Usually, NGOMSL is used to estimate how long it takes for a human to learn and use a user interface by assigning times to specific user interface operations. Here, however, we use NGOMSL as a notation for comparing two methods of refactoring tool usage.

We will use colors for clarity and introduce some notation for brevity. We have color-coded the steps in each method so that the correspondence between the steps in the two methods is more obvious. We will use = to mean "is equally fast and error-free as," $\succ$ to mean "is faster or more error-resistant than," and $\succeq$ to mean "is at least as fast and error-resistant as." Intuitively, you can think of = to mean "as good as," $\succ$ to mean "is better than," and $\succeq$ to mean "is at least as good as." Step-by-step, we demonstrate that refactoring using refactoring cues is at least as fast and error-resistant as with conventional tools.

$\boldsymbol{S_{RC} \succeq S_{CT}}$. Any valid editor selection in conventional refactoring tools is a valid cue selection with refactoring cues, so in these cases, $S_{RC} = S_{CT}$. Furthermore, refactoring cues are more error-resistant in certain cases, such as when a programmer over-selects by a few parentheses. In other cases, refactoring cues allow program elements that are leaves (that is, do not contain other program elements of the same kind, such as constants) to be selected with a single click. So in a some cases $S_{RC} \succ S_{CT}$, otherwise $S_{RC} = S_{CT}$.

$\boldsymbol{I_{RC} \succeq I_{CT}}$. Whatever initiation mechanism is used for conventional refactoring tools can also be used for refactoring cues, whether it be hotkeys, linear menus, or even pie menus. Thus, the time and accuracy of these two tools is the same. We have also made available a third initiation mechanism, the palette, which has various advantages and disadvantages when compared to hotkeys and menus, but will not be compared here. Therefore, when the palette is adventageous $I_{RC} \succ I_{CT}$, otherwise $I_{RC} \succeq I_{CT}$.

$\boldsymbol{N_{RC} = N_{CT}}$. Names are entered into a text box using conventional tools and directly into the editor with a linked, in-line rename using refactoring cues. However, from the user's perspective, using either tool for entering a new

name is a matter of typing, so $N_{RC} = N_{CT}$.

$\boldsymbol{G_{RC} \succeq G_{CT}}$. Configuration with the GUI in refactoring cues is not different from that with conventional tools, except that the user need not restart the process when she wishes to navigate the underlying source code. In cases when the programmer does not wish to perform any configuration, she can retain focus on the editor with refactoring cues but must shift focus to a dialog with conventional tools. In these cases, $G_{RC} \succ G_{CT}$, otherwise $G_{RC} = G_{CT}$.

$\boldsymbol{E_{RC} = E_{CT}}$. The underlying refactoring engine is the same for both refactoring cues and conventional tools, and both tools are initiated with a keystroke or button press, so the execution step is the identical. Therefore, $E_{RC} = E_{CT}$.

$\boldsymbol{M_{RC} \succeq M_{CT}}$. Using either tool, a programmer may choose to refactor more than one program element. With conventional tools, she must repeat steps 1 through 5, but with refactoring cues, she need repeat only steps 2 and 3. In Section 3.4, we showed that refactoring multiple elements was a significant use case. However, when the programmer wishes to refactor only one element, the user does nothing at this step using either tool. So when multiple program elements need to be refactored, $M_{RC} \succ M_{CT}$, otherwise $M_{RC} = M_{CT}$.

The analysis shows that refactoring cues are at least as fast and error-free as conventional refactoring tools because using both interfaces involves similar steps, just in a different order. When the order of the steps is factored out, both tools can be used in similar manners, and under certain significant use cases, refactoring cues are more error-resistant or faster.

6.3.2 *Limitations.* NGOMSL, and the GOMS family of analysis techniques in general, is limited in that it considers only procedural aspects of usability, not perceptual issues or users' conceptual knowledge of the system [John and Kieras 1996]. Additionally, a limitation of our analysis is that it does not take into account every use case. The most significant omitted cases are as follows:

- The programmer does not have to perform every step; she can bypass entering information into the GUI or typing an element name. These steps can be simply skipped using refactoring cues, but the programmer must at least dismiss a dialog when using conventional tools. Refactoring cues are slightly faster in this situation.
- Some steps can be reordered, such as N_{CT} and G_{CT}. However, this should not significantly change the speed and accuracy with which either tool can be used.
- While the programmer can use the keyboard or mouse for selection, our analysis assumes that the selection step is performed with the mouse. This is because using the keyboard for selecting when using refactoring cues requires one key-press more than when using a conventional tool. We are confident that the cost of this key press is overwhelmed by the speed improvements achieved elsewhere with refactoring cues, but we cannot express this with the step-by-step comparison given here. Therefore, for this analysis, we exclude the use of the keyboard during the selection step.

6.4 Opinion Study: Pie Menus, Refactoring Cues, Hotkeys, and Linear Menus

So far in this section we have talked about how quickly and accurately our tools can be used, but have neglected programmer satisfaction. If a programmer does not *want* to use a tool, it does not matter how fast or accurate it is.

6.4.1 *Methodology.* In an attempt to assess how programmers feel about using our pie menus and refactoring cues, we conducted interviews at the 2007 O'Reilly Open Source Convention in Portland, Oregon. Each interview lasted about 20 minutes. Subjects answered a brief oral questionnaire regarding their programming and refactoring experience. Subjects were then shown four videos, each lasting about 90 seconds, demonstrating a programmer's development environment while performing a refactoring:

- In the first video, the programmer is shown initiating a number of refactorings in series via a linear popup menu.
- In contrast, the second video shows the programmer performing the same refactorings with pie menus (albeit slowly, to allow the viewer to read the menus).
- In the third video, a programmer is shown using a conventional refactoring tool for several refactorings.
- In contrast, the fourth video shows the programmer performing the same refactorings with refactoring cues.

The videos can be viewed at http://multiview.cs.pdx.edu/refactoring/activation. After the second and fourth video, the interviewer compared pie menus for refactoring and refactoring cues with the standard Eclipse tools by verbally enumerating the advantages and disadvantages of each tool. This allowed us to reiterate the qualities of the tools shown in the videos and highlight some disadvantages that were not obvious from the videos. For instance, programmers were told that we place items on pie menus in a way that we believed was memorable, but also that we omit some popular refactorings for which our placement rules provide no guidance, such as Rename. A full list of the advantages and disadvantages presented to each subject is shown in Table III. We believe that telling the programmers about the advantages and disadvantages allowed them to make a more informed estimate of how the tools might effect their refactoring behavior.

We chose to show the subjects videos instead of allowing them to use the tools for two reasons. First, the videos ensured that each subject saw the tool working over the same code in the same manner. Second, because our tool was a prototype when we conducted the evaluation, the videos ensured that the subjects did not encounter bugs in our implementation that might have interfered with the experiment.

More information about this experiment, including the survey and data set, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

6.4.2 *Subjects.* The interviewer (also the first author of this paper) approached conference attendees to determine whether they were appropriate interviewees. The interviewer attempted to approach attendees indiscriminately, and interviewees were chosen if they had recently programmed Java, knew what refactoring was, and did not know the interviewer or his research. Of all the attendees that

Pie Menus for Refactoring
+ Can be activated using hotkeys or the mouse
+ Faster to activate than context menus
+ Easier to remember than hotkeys
+ Placement should be especially memorable
– Some refactorings do not make sense to put on this menu
– Adding new refactorings will disrupt memory

Refactoring Cues
+ Can be activated using hotkeys our mouse
+ Many items can be refactored at once
+ Selection errors are cut down
+ Makes "what is refactorable" explicit
+ Configuration becomes optional
– All refactoring activators must be shown

Table III. Advantages and disadvantages of pie menus and refactoring cues enumerated by the interviewer, labeled with + for advantage and – for disadvantage.

the interviewer approached, about one quarter met these criteria. While the interviewees cannot be considered a random sample of programmers, we believe that the interviewees were unbiased and fairly representative because we sampled from a national conference whose topic was orthogonal to refactoring. In total, 15 attendees were interviewed fully, while one declined to complete the interview due to time constraints.

The 15 interviewees reported a mean of 16 years of programming experience and spent an average of about 27 hours per week programming; 13 of the 15 interviewees were Eclipse users. We characterize the group as experienced programmers because 13 of 15 subjects had been programming for at least 10 years and because 13 of 15 subjects spent at least 20 hours per week programming at the time of the interview.

6.4.3 *Results.* When asked whether having the option of using pie menus, in addition to other initiation mechanisms, would increase their usage of refactoring tools, most interviewees said that they would use the tools the same amount. However, 6 of 15 interviewees estimated that the presence of pie menus would encourage them to refactor more. When we asked the same question about refactoring cues, 8 of 15 reported that they would use refactoring tools more often if refactoring cues were available. Several subjects expressed difficulty in answering this question without using the tools, but the positive responses indicate that both tools may encourage programmers to refactor more often using tools and less often by hand.

We asked one additional question at the end of the last 11 interviews. In that question, interviewees were asked to estimate how often they might use refactoring tools with hotkeys, linear menus, pie menus, or refactoring cues, if all were available in the same development environment. On average, pie menus were rated higher than linear menus, a difference that was statistically significant ($p = .028$, $df = 10$, $z = 2.2$, using a two-tailed Wilcoxon matched-pairs signed-rank test). This result is notable, because subjects reported that linear menus were the most popular way to initiate refactoring tools. No other differences were statistically significant; refactoring cues, pie menus, and hotkeys were all estimated to be used about the same amount.

Tool	Generality	Keyboard or Mouse	Speed	Memorability
Hotkeys	Some	Keyboard	Fast	Low
Linear Menus	All	Both	Slow	*
Pie Menus	Some	Both	Fast	High

Table IV. A comparison of initiation mechanisms for refactorings tools.

Tool	Steps Addressed	Generality	Keyboard or Mouse	Selection Error-Resistance	Stay in Editor
Editor Selection	Selection	All	Both	Low	Yes
Selection Assist	Selection	One	Both	Medium	Yes
Box View	Selection	One	Both	High	No
Wizard/Dialog	Configuration	All	Both	*	No
Linked In-line	Configuration	Some	Keyboard	*	Yes
Refactoring Cues	Selection & Configuration	All	Both	High	Yes

Table V. A comparison of selection and configuration mechanisms for refactoring tools.

We interpret the results of the survey to indicate that programmers are at least willing to try both pie menus and refactoring cues for activating refactoring tools.

6.4.4 *Limitations.* There are two main limitations of this survey. First, the programmers saw videos of how the tools worked, but were not able to try the tools themselves; programmers' actual usage of the tool will vary from their estimated usage. Second, the refactorings in the videos were created to exercise the features of each new refactoring tool. Therefore, the refactorings shown to the programers may not be representative of typical refactorings in the wild.

6.5 Summary: A Comparison of Activation Mechanisms for Refactoring Tools

Tables IV and V compare the features and performance of pie menus and refactoring cues with existing refactoring tool user interfaces. In the tables, by **Generality** we mean the number of refactorings the tool supports. For example, all refactorings can be initiated with linear menus, but typically by default only a subset can be initiated with hotkeys. The **Keyboard or Mouse** column refers to whether a tool can be used with the keyboard, the mouse, or both. The **Speed** and **Memorability** columns are summaries of the results presented in this paper. The **Steps Addressed** column names the steps in the refactoring process that the tool helps the programmer to accomplish (Section 3.1). The **Selection Error-Resistance** column refers to how well the tool helps the programmer avoid and recover from mis-selected program elements, again, in a simplified manner. The **Stay in Editor** column refers to whether the tool allows the programmer to stay focused in the editor while refactoring. An asterisk indicates that a column does not apply to a tool.

7. FUTURE WORK

In the future, we plan on further validating our guidelines by implementing tools to address usability problems in other steps of the refactoring process. For instance, avoiding distracting the programmer from her primary programming task may be

important when the programmer is finding candidates for refactoring. Building tools for other steps of the refactoring process should also help us to expand our guidelines.

Conducting a case-study evaluation of our refactoring tools would be valuable as well. While it has been helpful to conduct interviews and laboratory experiments, we believe that observing long-term tool usage is a better predictor of how programmers will behave in the wild. Such a study should measure how long it takes to perform individual refactorings, the number and variety of refactorings that are performed over the long-term, and whether users are satisfied after using our refactoring tools for several weeks.

We intend to improve the interfaces to our tools to address deficiencies exposed during our programmer interviews at the Open Source Convention. The most common dislike of programmers regarding pie menus was that they were "too ugly" and, ironically, obtrusive. In response, we plan on implementing a labels-only design, allowing a large proportion of the circular menu to be transparent. Regarding refactoring cues, we are concerned that cues may not be readable after many levels of nesting. In response, we are considering alternative cue coloring strategies, such as cushion tree maps [Lommerse et al. 2005], which would give the illusion of cue depth without the need for varied saturation. We will be changing our cue colors as well, as color-blind programmers report difficulty in distinguish pink and green cues. We also plan on addressing a few other minor usability issues with our tools.

We have created refactoring cues to make it faster to refactoring several program elements at once, and two future improvements may make this even faster. First, if multiple refactorings are executed at once, it may be possible to amortize the cost of precondition checking and code transformation. Second, refactoring cues could be modified to refactor-on-select, eliminating a keypress in the refactoring process.

8. CONCLUSION

In this paper we have described two usability improvements to the activation of refactoring tools: pie menus for refactoring and refactoring cues. We have presented data that justify both the need for improvements and the guidelines for their design. We have presented validation that suggests that these improvements are memorable, fast, error-resistant, and appealing to real programmers.

Refactoring tools have the potential to increase significantly the speed at which programs can be developed safely. However, to realize that potential, programmers must have refactoring tools that are appealing, fast, error-resistant, and that fit into their workflow. The ideal tool comes quickly to hand when needed and otherwise fades unobtrusively into the background; it must be the code, not the tool, that is the focus of the programmer's attention. In short, we seek to design refactoring tools that stay out of the programmer's way; the tools described here are a step towards that goal.

ACKNOWLEDGMENTS

For their advice and guidance, we thank Barry Anderson, Sergio Antoy, Margaret Burnett, Rob DeLine, Anthony Hornof, Mark Jones, Chuan-kai Lin, Ralph London, David Novick, Susan Palmiter, and our anonymous reviewers. Thanks to Gail

Murphy, Mik Kersten, and Leah Findlater for generously providing their Eclipse usage data. We also thank the participants in our experiment and interviewees in our surveys, and the National Science Foundation for partially funding this research under grant CCF-0520346.

REFERENCES

Balaban, I., Tip, F., and Fuhrer, R. 2005. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 265–279.

Boshernitsan, M., Graham, S. L., and Hearst, M. A. 2007. 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*. ACM, New York, NY, USA, 567–576.

Burnett, M. M. and Gottfried, H. J. 1998. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computerer-Human Interaction 5,* 1, 1–33.

Callahan, J., Hopkins, D., Weiser, M., and Shneiderman, B. 1988. An empirical comparison of pie vs. linear menus. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 95–100.

de Alwis, B. and Murphy, G. C. 2006. Using visual momentum to explain disorientation in the Eclipse IDE. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*. IEEE Computer Society, Washington, DC, USA, 51–54.

Eclipse. 2008. *Eclipse*. The Eclipse Foundation. Computer Program, http://www.eclipse.org.

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

Grossman, T., Dragicevic, P., and Balakrishnan, R. 2007. Strategies for accelerating online learning of hotkeys. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1591–1600.

Hong, J. I. and Landay, J. A. 2006. SATIN: A toolkit for informal ink-based applications. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*. ACM, New York, NY, USA, 7.

JetBrains. 2008. IntelliJ IDEA. Computer Program, http://www.jetbrains.com/idea.

John, B. E. and Kieras, D. E. 1996. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction 3,* 4, 320–351.

Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 576.

Kurtenbach, G. and Buxton, W. 1993. The limits of expert performance using hierarchic marking menus. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 482–487.

Lippert, M. 2004. Towards a proper integration of large refactorings in agile software development. In *Extreme Programming and Agile Processes in Software Engineering, Proceedings of 5th International Conference (XP 2004)*, J. Eckstein and H. Baumeister, Eds. Lecture Notes in Computer Science, vol. 3092. Springer, Garmisch-Partenkirchen, Germany, 113–122.

Lommerse, G., Nossin, F., Voinea, L., and Telea, A. 2005. The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*. IEEE Computer Society, Washington, DC, USA, 4.

Ludolph, F., Chow, Y., Ingalls, D., Wallace, S., and Doyle, K. 1988. The Fabrik programming environment. In *IEEE Workshop on Visual Languages*. IEEE Computer Society Press, Pittsburgh, PA, USA, 222–230.

Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. 2007. Improving usability of software refactoring tools. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 307–318.

Murphy, G. C., Kersten, M., and Findlater, L. 2006. How are Java software developers using the Eclipse IDE? *IEEE Software 23,* 4, 76–83.

Murphy-Hill, E. and Black, A. P. 2007a. High velocity refactorings in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange*. ACM, New York, NY, USA. In Press.

Murphy-Hill, E. and Black, A. P. 2007b. Why don't people use refactoring tools? In *Proceedings of the 1st ECOOP Workshop on Refactoring Tools*, D. Dig and M. Cebulla, Eds. TU Berlin, Berlin, Germany, 61–62. Technical Report 2007/08, ISSN 1436-9915. http://iv.tu-berlin.de/TechnBerichte/2007/2007-08.pdf.

Murphy-Hill, E. and Black, A. P. 2008. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*. ACM, New York, NY, USA, 421–430.

NetBeans. 2008. Mouse gestures plugin to Netbeans. Computer Program, https://mousegestures.dev.java.net/.

Omnicore. 2008. *X-develop IDE*. Omnicore Software. Computer Program, http://www.omnicore.com/xdevelop.htm.

Roberts, D., Brant, J., and Johnson, R. 1997. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems 3,* 4, 253–263.

Roberts, D. B. 1999. Practical analysis for refactoring. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. Adviser-Ralph Johnson.

Shneiderman, B. 1987. *Designing the User Interface (2nd ed.): Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Simon, F., Steinbrückner, F., and Lewerentz, C. 2001. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, 30.

Smartdec. 2008. IDEA mouse gestures. Computer Program, http://www.smardec.com/products/mouse-gestures.html.

Tetzlaff, L. and Schwartz, D. R. 1991. The use of guidelines in interface design. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 329–333.

Xing, Z. and Stroulia, E. 2006. 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*. IEEE Computer Society, Washington, DC, USA, 458–468.