

Restructuring Software with Gestures

Emerson Murphy-Hill
The University of British Columbia
Vancouver, British Columbia, Canada
emhill@cs.ubc.ca

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

Abstract

Refactoring is the process of changing the structure of code without changing its meaning, and is a frequent practice among developers. Once the need for a refactoring has been recognized, refactoring tools are available that automate the restructuring, yet programmers use these tools infrequently. This is a problem because refactoring without a tool increases the risk of introducing bugs. We argue that the need to recall the name of a refactoring before the appropriate tool can be invoked makes it unnecessarily hard to initiate a refactoring with a tool. Conventional ways of initiating a tool also make it hard to transition from novice tool user to expert tool user.

The contribution of this paper is a memorable mapping from gestures to refactorings, and an implementation of that mapping in the form of marking menus. In the first reported experiment to explore the effect of the position of items in marking menus on people's ability to infer the location of those items, we asked 16 programmers to complete a paper-based evaluation of our mapping. The results suggest that programmers can infer the gesture that will invoke the appropriate refactoring tool, even if they do not know the name of the refactoring.

1. Introduction

Fowler characterizes *refactoring* as the process of changing the structure of existing code while preserving its functionality [3, p. xvi]. While the term “refactoring” is fairly new, it captures what programmers have done for a long time: ever since programs have been structured, programs have been re-structured. Refactoring is critical to software development: it allows programmers to adapt software to the changing demands of customers, markets, and managers. In industry, refactoring is practiced frequently, regularly, and in support of higher-level programming goals such as fixing bugs and adding features [11, 14]. Because of the importance of refactoring, many integrated

development environments now include tools that semi-automate the process, such as Eclipse (<http://eclipse.org>), Visual Studio (<http://microsoft.com/visualstudio>), and Xcode (<http://developer.apple.com/tools/xcode>). An example of a refactoring tool in such an environment is the PULL UP METHOD tool, which moves a method from a class to its superclass and makes sure that all references to that method will continue to be valid after the method is moved.

Although previous research has shown that programmers refactor frequently, programmers do not use refactoring *tools* frequently [11]. This underuse signals a missed opportunity: because refactoring is frequent, it follows that a large number of refactorings are being performed by hand, unnecessarily slowing-down software development and increasing the chance that a bug might be introduced.

Much research has been performed to promote refactoring tool use by improving the user interface of the tools, including improving the programmers' ability to identify opportunities for refactoring [13], to select code as input to a refactoring tool [9], to refactor code spread across a code base [12], and to understand the errors that refactoring tools sometimes produce [9].

To use a refactoring tool, a programmer must communicate to the programming environment which refactoring it should perform. We call this the *initiation* step in the process of refactoring. Initiation is an under-explored area of research in the user interface of refactoring tools. Typically, initiation takes the form of invoking a refactoring tool by using a linear menu or by pressing a hotkey; Figure 1 shows the refactoring menu in Eclipse. As we argue in the next section, neither of these mechanisms is ideal.

The contribution of this paper is a novel mapping from gestures to refactorings: we discuss why such a mapping it is needed, describe our implementation, and report the results of an evaluation.

2. Problems with Initiation

We argue that the traditional initiation mechanisms—context menus, application menus, and hotkeys—have two

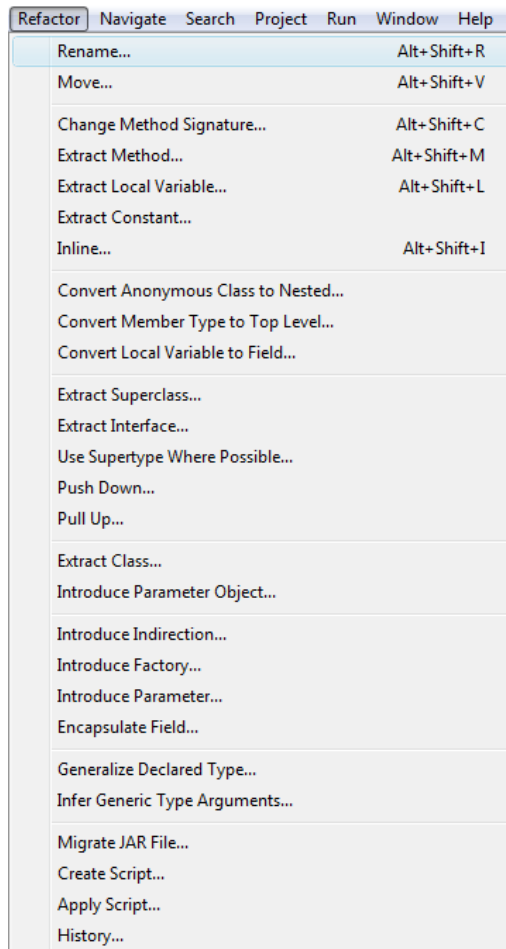


Figure 1. A refactoring menu in Eclipse.

problems that make them less than ideal.

Problem 1: Naming. Suppose that a programmer wants to make a code change that is a refactoring. How does she know which refactoring tool to use to perform that change? Typically she recalls the *name* of the refactoring that she wants to perform, such as RENAME, EXTRACT METHOD, or INTRODUCE FACTORY, and then finds that name in a menu. There are two problems with this approach. If the programmer is sure of the name of the refactoring, she must still find it in what may be a long menu (see Figure 1). We have some evidence of this problem from a survey of programmers; when asked why they choose not to use refactoring tools, one respondent said that the “menu is too big sometimes, so searching [for] the refactoring takes too long” [10]. However, we hypothesize that the more serious problem is that the names of refactorings are somewhat capricious and vary from one environment to another. For example, Fowler’s INTRODUCE EXPLAINING VARIABLE [3] is called EXTRACT LOCAL VARIABLE in Eclipse.

We observe that the *names* of the refactoring tools are really a distraction to the programmer, who needs to realize, for example, only that her code would be improved if a long sub-expression is assigned to a meaningfully-named local variable. In a follow-up interview to our recent research on programmers’ refactoring behavior [11], one programmer complained about this problem explicitly: “I may know what I’m doing is refactoring, but may not know what this particular refactoring action is called within the tool”. Thus, by forcing the programmer to learn (or guess) the name that the environment gives to that refactoring before she can use the refactoring tool, the environment makes tool initiation unnecessarily burdensome.

Problem 2: Transitioning from Novice to Expert. Assuming that the programmer knows the name of the refactoring that she wants, she can select that refactoring from a context or application menu. But such linear menus can be slow [2] because the user must take significant time to navigate *to* the menu, then *through* the menu, and finally *to point at* the desired menu item. Thus, to initiate a tool faster, the programmer may instead want to use a “hotkey”, that is, a combination of keypresses on the keyboard. However, transitioning from using the menu (novice behavior) to using a hotkey (expert behavior) is known to be a slow process [4].

In the case of refactoring tools, transitioning from menus to hotkeys is made difficult by the same problem that we identified above: the need to remember the *name* that the environment gives to the refactoring. The mapping from code transformation to hotkey is indirect: the programmer must start with the *idea* of the transformation that she intends to perform, such as “take this expression and assign it to a temporary variable”, recall the *name* of the corresponding refactoring (EXTRACT LOCAL VARIABLE), and only then map that name to a contrived hotkey (remembering that Shift+Alt means refactor, and that L stands for “Local”).

3. A Mapping from Gestures to Refactorings

To solve the naming problem, we have created a mapping from gestures to refactorings. This mapping forms a consistent foundation for initiating refactoring tools. It is based on the observation that the structural nature of many refactorings can be associated with a gesture using a consistent directional metaphor. Our mapping derives from three rules that determine the gestural direction of a refactoring.

- Directional refactorings are assigned their respective direction, so, for example, PULL UP FIELD is assigned the up gesture. This rule capitalizes on the convention that class hierarchies are shown with superclasses above subclasses.

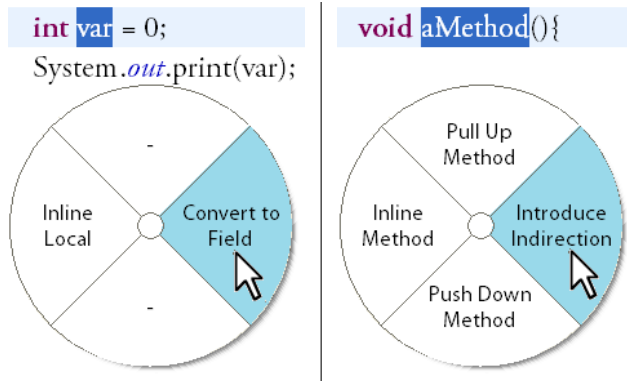


Figure 2. Two marking menus for refactoring, showing applicable refactorings for a local variable and a method.

- Refactorings that are inverses are assigned opposite directions, so, for example, `INLINE METHOD` is opposite `EXTRACT METHOD`. This rule makes use of the observation that the inverse of many refactorings is another refactoring.
- Refactorings that are conceptually similar are assigned the same gesture, so, for example, the same gesture is used for `INLINE METHOD` and `INLINE LOCAL VARIABLE`. This rule capitalizes on similarities between refactorings.

We hypothesize that programmers can *infer* our mapping after being exposed to only a small number of examples, such as by invoking a refactoring tool a few times using gestures. This would assist in solving the naming problem, because it would allow the programmer to infer how to initiate a refactoring without first learning its name. We validated this hypothesis with an experiment, described in Section 5.

This mapping from gestures to refactorings can be used to initiate refactoring tools in a number of ways. One way is with hotkeys: the programmer selects a program element to be refactored, presses a key to bring up the menu, and then presses the up-, down-, left-, or right-arrow key to “gesture” towards a refactoring. Another way is to allow the programmer to gesture using the mouse. Further possibilities include gesturing with the eye, hand, or stylus. In the next section, we describe how we combined mouse and hotkey gestures using *marking menus*.

4. Marking Menus for Refactoring

To demonstrate our mapping, we have added marking menus as a new user-interface element for the existing Eclipse refactoring engine; our implementation is based on SATIN [5]. A marking menu is a special kind of pie menu;

it is similar to a context menu except that items appear around a circle instead of in a linear list. Two examples are shown in Figure 2. Our marking menus are invoked like standard context menus, using a dedicated mouse button or hotkey; we presently use the middle mouse button. Because marking menus are context sensitive, the menu displayed depends on the kind of program element that has been selected. So as one example, as Figure 2 shows, one menu will appear when the selection is a variable name, and a different menu when the selection is a method name. Moreover, the user may transition to expert behavior by “mousing ahead” and gesturing in the direction of the desired menu item, even if that item has not yet been drawn on the screen. In this way, marking menus allow programmers to smoothly transition from beginner behavior (point-and-click) to expert behavior (mouse-ahead).

We encourage the reader to watch our screencast or try marking menus for refactoring at <http://multiview.cs.pdx.edu/refactoring/activation>.

In addition to allowing the programmer to smoothly transition from beginner to expert, implementing our mapping in the form of marking menus has additional advantages. Marking menus are significantly faster than linear menus because items are selected by direction alone, rather than by direction and distance [2]. Moreover, after frequent use of a menu item, muscle memory helps users remember to gesture in the direction in which menu items are located [7]. Gestures are efficient during programming as well; Burnett and Gottfried have shown that people using gestures to issue programming commands can program faster and with fewer errors [1]. Because of the mapping, refactoring appears to be one of the applications of marking menus that Callahan and colleagues say “seem to fit well into the mold of the pie menu design,” and for which they showed that pie menus have a distinct speed advantage [2].

The main limitation of using our mapping with marking menus for refactoring is that our marking menus are restricted to four items per context, with no submenus. Of 22 refactorings that can be initiated via a linear menu in Eclipse, 11 can be initiated with our marking menus; our menus also support 3 additional refactorings not currently in Eclipse, as shown in Table 1. The restriction to four items per menu means that certain refactorings do not appear on our marking menus, especially refactorings that have no spatial mapping, such as `RENAME`.

There are several reasons that we do not view our incomplete coverage of the space of refactorings as a significant problem. First, we argue that completeness at the expense of simplicity is not always desirable when it comes to user interfaces. Second, programmers currently use several mechanisms for initiating different kinds of refactoring tools [8], and thus programmers may be willing to use some other initiation mechanism in conjunction with mark-

ing menus. Third, the incompleteness is not fundamental to our approach; it is possible to extend marking menus for refactoring to provide for the initiation of all refactorings, as we discuss in Section 6.

5. Experiment

This section describes an experiment to test the hypothesis that programmers can infer our mapping. Our main result — that programmers can indeed infer the mapping — implies that programmers will be able to use that mapping to initiate refactoring tools, even if they do not know the name of the refactoring tool that they want.

In addition to serving as a mechanism to test our hypothesis, we chose to perform this particular experiment for two reasons. First, to our knowledge, it is the first experiment of its kind: no other reported experiment has evaluated the effect of item placement on people’s ability to infer where items appear. Second, by performing a paper-based laboratory experiment, rather than a field study, we are able to isolate the cause of improved programmer performance. If we had chosen to do a field study where we, for example, compare marking menus using our mapping to linear menus and hotkeys, if we could indeed show increased performance or adoption of refactoring tools, we would be unable to ascertain the source of the improvement. For instance, an improvement could be due to the mapping, the menus, or a simple novelty effect, where programmers exhibit better performance because of an increased interest.

5.1. Methodology

In this experiment, programmers were asked to memorize the direction (left, right, top, or bottom) of a refactoring on a marking 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 marking menu for refactoring: the associated refactoring was highlighted on one of the four menu quadrants (Figure 3, top). We told programmers to try to associate the before-and-after code with a direction. We gave programmers 30 seconds to make this association to ensure that they were spending most of their time understanding the refactoring, and that little time was available for memorization; interviewing pilot experiment participants informally confirmed this.

We used before-and-after code to identify the refactoring, rather than using only its name. There were three reasons for this:

1. programmers who had no knowledge of refactoring terminology could still participate in the experiment;

2. our choice of refactoring names would not confuse programmers who had experience with refactoring, but who used different terminology; and
3. if programmers think of a refactoring as a code transformation and not as a name in a book, then the experiment more closely matches how programmers refactor in the wild.

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 marking menus removed (Figure 3, bottom). Within a total of 5 minutes, we told programmers to choose where the refactoring appeared on the marking menu. Additionally, programmers were asked to infer 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 mapping, should appear in the same direction as the `CONVERT LOCAL TO FIELD`, `ENCAPSULATE FIELD`, and `INTRODUCE INDIRECTION`, 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>.

5.2. 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 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. Two subjects were excused from the programming class set of subjects because one was not familiar with Java code and one did not follow directions during the experiment; this left a total of 16 participants.

5.3. Theory

The key to this experiment is how we test our theory that programmers can infer our mapping. We assume that the ability to infer our mapping arises from programmers’ ability to create and use a mental model that explains the directions of refactorings. Our experiment is similar to an experiment conducted by Kieras and Bovair [6], in which they showed that users could learn, retain, and execute operations faster on a simple physical device when the users were taught to use a mental model of the device. This group of users, called the “model group,” performed significantly

Refactoring Tool	<i>can be initiated by...</i>	Linear Menus	Hotkeys	Marking 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 1. How refactorings can be initiated in our current implementation of marking menus.

better than the “rote group,” who were taught by rote memorization.

Our experiment and underlying theory is similar to Kieras and Bovair’s. In our experiment, within each set of subjects (students, research assistants, industrial programmers), each programmer was randomly assigned to one of two groups. In the model group, programmers were trained on marking menus whose items were placed according to our mapping. In the rote group, programmers were trained on marking menus whose items were placed in a manner that violated each rule in our mapping. In this way, programmers in the rote group would be forced to rely exclusively on memory recall, whereas programmers in the model group will have, according to our hypothesis, the use of inference using their mental model. Thus, the difference between the two groups is attributable to programmers’ ability to infer our mapping.

5.4. Results

Overall, subjects in the rote group correctly chose a median of 3 refactoring directions, while subjects in the model group correctly chose a median of 7.5 refactoring directions. The difference is statistically significant ($p = .011$,

$df = 1$, $z = 2.553$ using a Wilcoxon rank-sum test). Furthermore, six out of the eight subjects in the model group correctly intuit the direction of the refactoring that they had not seen during training; in comparison, we would expect that if subjects had simply guessed the direction, only two out of eight would have chosen correctly¹. The results of the experiment are shown in Figure 4.

5.5. Threats to Validity

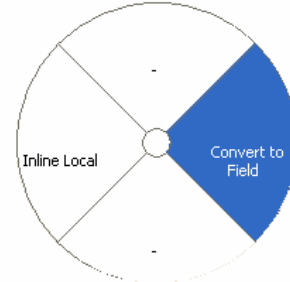
There are three limitations of this experiment. First, subjects were asked to memorize the direction of 9 refactorings, which represent only about one-tenth of the refactorings cataloged by Fowler [3]. The effect of trying to recall the full catalog of refactorings is unclear. Second, we cannot easily explain the outliers in the data set (one overperforming rote group subject and two underperforming model group subjects, Figure 4). It may be the case that some programmers can easily recall the direction of a refactoring, regardless of their placement, and that some programmers

¹We do not report the number of rote group subjects who correctly guessed because the definition of a “correct guess” is ambiguous, since refactorings for rote group subjects were essentially placed randomly.

```

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;
    }
}

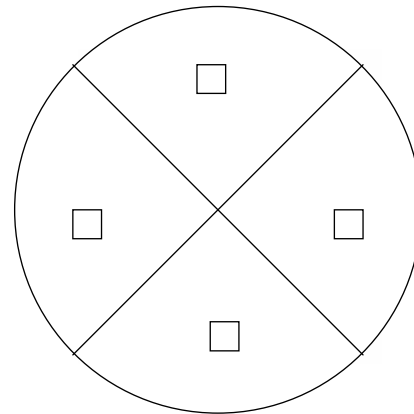
```

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;
    }
}

```

Figure 3. A training page (top) and associated testing page (bottom); in each, code before-and-after refactoring at left. Code to-be-refactored is highlighted in black, and changes are in yellow.

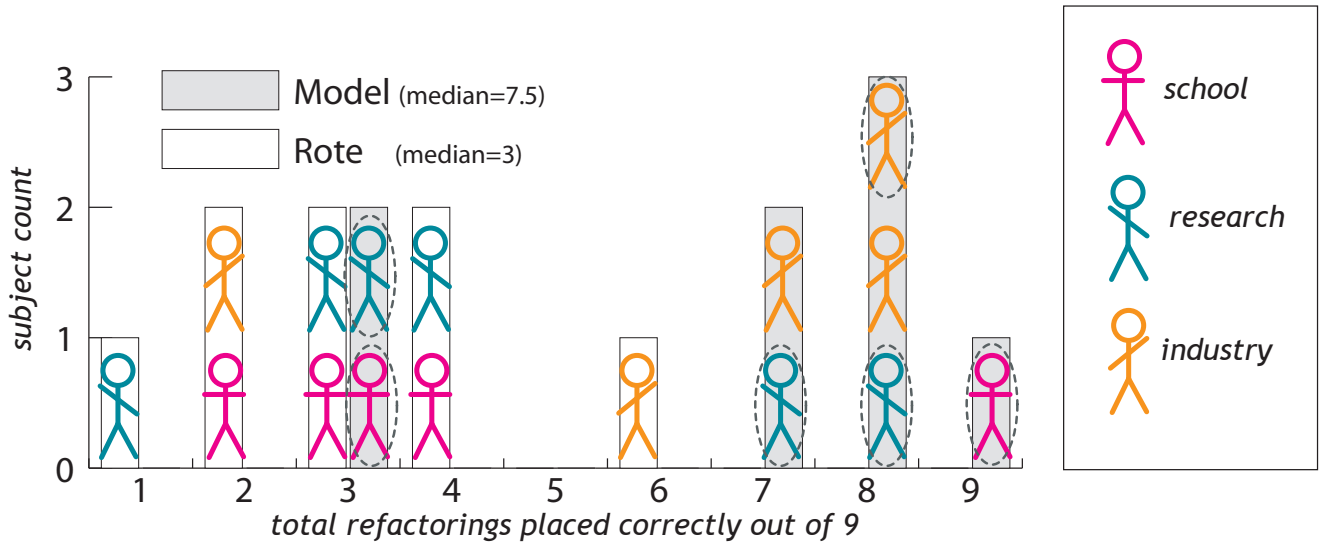


Figure 4. A histogram of the results of the marking menu experiment. Each subject is overlaid as one stick figure. Subjects from the model group who correctly guessed the refactoring that they did not see during training are denoted with a dashed oval.

have difficulty recalling the direction even when placed according to our mapping. Third, the experiment was conducted in such a way that it is difficult to discern which of the three mapping rules was most helpful.

5.6. Discussion

The results confirm our hypothesis that programmers can infer our mapping because programmers in the model group more often chose the correct direction for refactorings than programmers in the rote group. This conclusion is strengthened by the ability of the majority of model group subjects to choose the correct location of a refactoring on which they had not been trained. We postulate that the ability to infer our mapping can help programmers use refactoring tools, and, in the long run, improve their adoption.

6. Further Directions

Previous work showing improved speed and accuracy of marking menus, along with the experiment described in this paper, suggest that programmers should be able to use marking menus to initiate refactoring tools efficiently. With these results in place, further study in the form of a long-term field study is now warranted. Towards that end, we have conducted structured developer interviews at the O’Reilly Open Source Convention in Portland, Oregon. After showing 15 experienced Java developers short demonstrations of marking menus for refactoring, on average they

estimated that they would use marking menus for refactoring and hotkeys equally often, but would use marking menus for refactoring significantly *more* than linear menus for refactoring. This predicts that our tools will be a useful addition to the programming toolbox, but further study is needed to see if this prediction holds up in practice.

We are considering how to alleviate one of the major disadvantages of using our mapping with marking menus, that some refactorings do not appear on the menu. One way to do this would be to allow the user to click on the center of the marking menu to reveal a nested marking menu, or a linear menu, that contains refactorings that do not have a spatial mapping. Another way would be to use traditional hierarchical marking menus [7]. Each of these designs has usability tradeoffs, and we plan on investigating these tradeoffs in the future.

We would also like to investigate whether similar mappings are effective in other domains where system interaction is well-structured. For example, we can envision an analogous mapping for graphical design systems, where the up-down and more-less metaphor is systematic, such as moving objects up and down layers, increasing or decreasing transform intensities (e.g., transparency, blurring, hue), and increasing or decreasing magnitudes (e.g., text size, stroke weight, rotation).

7. Conclusion

This paper introduces a novel mapping from gestures to refactorings and reports results that suggest that programmers can infer that mapping. This result suggests that gestural initiation of refactoring tools will help the right tool come quickly to hand when needed and otherwise fade into the background: our objective is for the code, not the tool, to be the focus of the programmer's attention.

8. 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, Karyn Moffatt, Gail Murphy, David Novick, Susan Palmiter, and our anonymous reviewers. Thanks to the Interaction Design Reading Group at UBC for their insightful comments. We also thank the participants in our experiment, and the National Science Foundation for partially funding this research under grant CCF-0520346.

References

- [1] Margaret M. Burnett and Herkimer J. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5(1):1–33, 1998.
- [2] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 95–100, New York, NY, USA, 1988. ACM.
- [3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [4] Tovi Grossman, Pierre Dragicevic, and Ravin Balakrishnan. Strategies for accelerating on-line learning of hotkeys. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1591–1600, New York, NY, USA, 2007. ACM.
- [5] Jason I. Hong and James A. Landay. SATIN: A toolkit for informal ink-based applications. In *ACM SIGGRAPH Courses*, pages 63–72, 2006.
- [6] David E. Kieras and Susan Bovair. The role of a mental model in learning to operate a device. *Cognitive Science: A Multidisciplinary Journal of Artificial Intelligence, Psychology and Language*, 8(3):255–273, 1984.
- [7] Gordon Kurtenbach and William Buxton. 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*, pages 482–487, New York, NY, USA, 1993. ACM.
- [8] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.
- [9] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, NY, USA, 2008. ACM.
- [10] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), September-October 2008.
- [11] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.
- [12] Alexis O'Connor, Macneil Shonle, and William Griswold. Star diagram with automated refactorings for eclipse. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, New York, NY, USA, 2005. ACM.
- [13] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization*, pages 77–86, New York, NY, USA, 2008. ACM.
- [14] 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.