# Programmer-Friendly Refactoring Errors

Emerson Murphy-Hill, and Andrew P. Black

**Abstract**—Refactoring tools, common to many integrated development environments, can help programmers to restructure their code. These tools sometimes refuse to restructure the programmer's code, instead giving the programmer a textual error message that she must decode if she wishes to understand the reason for the tool's refusal, and what corrective action to take. This article describes a graphical alternative to textual error messages called *Refactoring Annotations*. It reports on two experiments, one using an integrated development environment and the other using paper mockups, that show that programmers can use Refactoring Annotations to quickly and accurately understand the cause of refactoring errors.

**Index Terms**—refactoring, refactoring errors, usability, programmers, tools

## 1 Introduction

Refactoring is the process of changing the structure of code without changing the way that it behaves [6]. Refactoring is considered a best-practice when creating and maintaining software, and indeed, research suggests that programmers practice it regularly [16], [22]. Examples of refactoring include renaming a variable, moving a method from a superclass to its subclasses, and taking a few statements and extracting them into a new method. Each kind of refactoring has a name: these examples are called RENAME, PUSH DOWN METHOD, and EXTRACT METHOD [6].

Refactoring tools, such as those provided in the Smalltalk [20] and Eclipse [22] programming environments, automate much of what the programmer would normally do manually. Take the example of EXTRACT METHOD. Suppose that I am a programmer working on the method shown in Figure 1, and I am concerned about the similarity of the three `for` loops between lines 18 and 46. If I wanted to factor out the duplicated code, the first step I might take is to put one of the loops into a new method; I could then abstract that method so that it contains the code that is common to the three loops. If I were using the Eclipse environment (eclipse.org), I might do this using the built-in EXTRACT METHOD tool by, for example, selecting the last loop (lines 38–46) with my cursor and invoking the refactoring tool using a hotkey.

When the tool performs the refactoring, it first analyzes the code and learns that three values must be passed in to the new method (`iLastNumber`, `pos`, and `pieceNumbers`). It also learns that the value of `pos` must be returned, because the variable `pos` is assigned to in the extracted code, and is read on line 52. To maintain the original semantics, the refactored code would instead assign the return value of the extracted method to `pos`. The result of the refactoring is shown in Figure 2. This example illustrates the advantage of using a refactoring tool over refactoring by hand: the tool can automatically perform a static analysis that will enable it to "do the right thing," whereas if I refactor by hand, I have to analyze the flow of data through the program myself. In short, refactoring tools save the programmer from doing error-prone work. Thus, because programmers already refactor regularly, we can see that if they used tools for that refactoring, their productivity could be improved.

Suppose that I instead select the first `for` loop (lines 18–26) and try to use Eclipse's refactoring tool. In that case, the tool would refuse to refactor my code, instead displaying the error message of Figure 3. This error message is an indication that a precondition of the refactoring was violated in the original code. In this case, the tool's data flow analyzer determined that several variables were being assigned in the loop, and that at least two of them are used later-on in the code. This is a problem because, in Java, a method can return only a single value, so the calling method could not update multiple variables based on values returned from the new method. Thus, the tool is unable to perform the EXTRACT METHOD refactoring. (In a language that supported multiple results or out parameters, this would not be a problem.)

The topic of this article is making refactoring error messages easier to understand. While significant research has been done to ensure the correctness of refactorings (for example [2], [7], [11], [18]), to our knowledge, no other researchers have investigated how to present errors when behavior preservation cannot be assured. Using a small formative study, we show that programmers encounter errors fairly frequently when using Eclipse's EXTRACT METHOD refactoring tool. Moreover, when errors occur, programmers typically misunderstand the error messages or simply ignore them (Section 2). We then describe a graphical alternative to EXTRACT METHOD error messages, called Refactoring Annotations, which we implemented as an Eclipse plugin (Section 3). We show how programmers can then use Refactoring Annotations to understand the causes of refactoring errors significantly faster and more

- *E. Murphy-Hill is with the Department of Computer Science, North Carolina State University, Raleigh, NC, 27695.*
*E-mail: emerson@csc.ncsu.edu*
- *A. P. Black is with the Department of Computer Science, Portland State University, Portland, OR, 97201.*
*E-mail: black@cs.pdx.edu*

```
1  if( destroyed )  return new int[0];
2
3  /** Cheap hack to reduce (but not remove all)
4          the # of duplicate entries */
5  int iLastNumber = -1;
6  int pos = 0;
7  int[] pieceNumbers;
8
9  try {
10   lock_mon.enter();
11
12   // allocate max size needed
13   // (we'll shrink it later)
14   pieceNumbers = new int[queued_messages.size()
15          +loading_messages.size()
16          +requests.size()];
17
18   for(Iterator iter = queued_messages.keySet().
19          iterator(); iter.hasNext();) {
20     BTPiece msg =
21          (BTPiece) iter.next();
22     if (iLastNumber != msg.getPieceNumber()) {
23       iLastNumber = msg.getPieceNumber();
24       pieceNumbers[pos++] = iLastNumber;
25     }
26   }
27
28   for(Iterator iter=loading_messages.iterator();
29          iter.hasNext();) {
30     DiskManagerReadRequest dmr =
31          (DiskManagerReadRequest) iter.next();
32     if (iLastNumber != dmr.getPieceNumber()) {
33       iLastNumber = dmr.getPieceNumber();
34       pieceNumbers[pos++] = iLastNumber;
35     }
36   }
37
38   for(Iterator iter = requests.iterator();
39          iter.hasNext();) {
40     DiskManagerReadRequest dmr =
41          (DiskManagerReadRequest) iter.next();
42     if (iLastNumber != dmr.getPieceNumber()) {
43       iLastNumber = dmr.getPieceNumber();
44       pieceNumbers[pos++] = iLastNumber;
45     }
46   }
47
48 } finally {
49   lock_mon.exit();
50 }
51
52 int[] trimmed = new int[pos];
53 System.arraycopy(pieceNumbers,0,trimmed,0,pos);
54
55 return trimmed;
```

Fig. 1. The body of a method from the `OutgoingBTPieceMessageHandler` class in the open-source Vuze project (azureus.sourceforge.net).

```
38   pos = extractedLoop(iLastNumber,pieceNumbers,pos);

58 int extractedLoop(int iLastNumber,
59                   int[] pieceNumbers,
60                   int pos){
61   for(Iterator iter = requests.iterator();
62          iter.hasNext();) {
63     DiskManagerReadRequest dmr =
64          (DiskManagerReadRequest) iter.next();
65     if (iLastNumber != dmr.getPieceNumber()) {
66       iLastNumber = dmr.getPieceNumber();
67       pieceNumbers[pos++] = iLastNumber;
68     }
69   }
70  return pos;
71 }
```

Fig. 2. The result of an EXTRACT METHOD performed on the code in Figure 1.

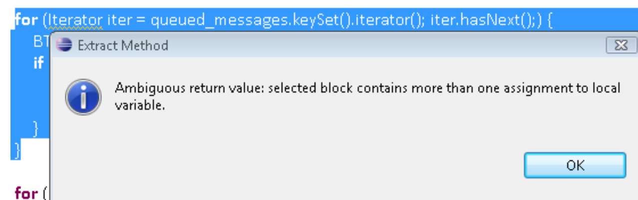


Fig. 3. Eclipse refusing to perform EXTRACT METHOD.

accurately than standard Eclipse error messages (Section 4). Based on what we learned while building and evaluating Refactoring Annotations for EXTRACT METHOD, we distill a set of usability guidelines that, we believe, capture what makes Refactoring Annotations effective. We then map the guidelines onto a taxonomy of refactoring tool error messages, sketching how our guidelines can be applied to all refactoring errors (Section 6). Finally, in a paper-based evaluation, we show that programmers can understand the causes of a variety of refactoring errors faster and with significantly more accuracy when using our graphical Refactoring Annotations, compared to using traditional textual error messages (Section 7).

This article partially overlaps a previous study, presented at ICSE 2008 [15], in which we described Refactoring Annotations for EXTRACT METHOD, discussed here in Sections 2 through 5, as well as two tools for selecting code, which are not discussed in this article. The contributions of the previous study were

- the introduction of Refactoring Annotations, a novel visualization for expressing precondition violations during EXTRACT METHOD;
- an evaluation that suggests that Refactoring Annotations for EXTRACT METHOD are faster, more accurate, and more satisfying to use than their textual counterparts; and
- a set of guidelines for building error visualizations for future refactoring tools.

This article elaborates on these contributions, and in addition presents

- a taxonomy of refactoring preconditions, which we derive from refactoring tools for 4 different languages;
- an application of our guidelines to the taxonomy, showing that the guidelines can be applied to a variety of refactoring tools; and
- an evaluation that suggests that Refactoring Annotations are more usable than error messages in a variety of refactoring tools.

## 2 A Formative Study of Refactoring

We hypothesized that the error messages emitted by existing refactoring tools are non-specific and unhelpful. This hypothesis about error messages has been explored in similar domains, such as compilation errors. Such studies have shown that programmers have difficulty understanding compilation error messages [12], [19], so it is natural to expect that programmers may also have difficulty with refactoring error messages.

To explore this hypothesis, and to better understand the usability problems that exist in modern refactoring tools, we observed several programmers use the Eclipse Extract Method refactoring tool. We focused on Extract Method in Eclipse because it is a mature, non-trivial refactoring tool and because many refactoring tools, like Eclipse, use error messages to communicate precondition violations.

In this formative study, we observed eleven programmers perform a number of Extract Method refactorings. Six of the programmers were Ph.D. students and two were professors from Portland State University; three were commercial software developers. We asked the participants to use the Eclipse Extract Method tool to refactor parts of several large, open-source projects:

- Vuze, a peer-to-peer file-sharing client (http://azureus.sourceforge.net);
- GanttProject, a project scheduling application (http://ganttproject.biz);
- JasperReports, a report generation library (http://jasperforge.org);
- Jython, a Java implementation of the Python programming language (http://www.jython.org); and
- the Java 1.4.2 libraries (http://java.sun.com/j2se/1.4.2/download.html).

We chose these projects because of their size and maturity. Programmers were free to refactor whatever code they thought necessary; they were allowed to use a tool to help find long methods, which can be good candidates for refactoring. Each refactoring session was limited to 30 minutes; programmers successfully extracted between 2 and 16 methods during that time.

We made the following conjectures based on our observations of programmers struggling with refactoring error messages during this study:

- Many programmers encounter refactoring errors. In all, 9 out of 11 programmers experienced at least one error message while trying to extract code. The 2 exceptions performed some of the fewest extractions in the group, so were among the least likely to encounter errors. Surprisingly, these 2 exceptions were among the most experienced programmers in the group; we hypothesize that they performed so few extractions because they avoided refactoring code that might possibly generate error messages.
- Some programmers encounter errors frequently. For example, one programmer attempted to extract 34 methods and encountered errors during 23 of these attempts.
- Programmers encounter refactoring errors from a variety of sources. In the study, programmers encountered errors regarding invalid or inappropriate selections, multiple assignments, and control flow problems.
- Error messages are insufficiently descriptive. Programmers, especially refactoring tool novices, may not understand an error message that they have not seen before. When we asked what an error message was saying, several programmers were unable to explain the problem correctly.
- Programmers have difficulty assessing the amount of work required to resolve an error. This was partially because even if multiple precondition violations existed during a particular application of the tool, Eclipse reported only a single violation.
- Programmers confuse error messages. All the errors were presented as graphically-identical text boxes with identically formatted text. At times, programmers interpreted one error message as an unrelated error message because the errors appeared identical at a quick glance. Improving the message text would not solve this problem: the clarity of the message text is irrelevant when the programmer does not take the time to read it.
- Error messages discourage programmers from refactoring at all. For instance, if the tool said that a method could not be extracted because there were multiple assignments to local variables (Figure 3), the next time a particular programmer came across any assignments to local variables, the programmer did not try to refactor, even if no precondition was violated.

This study shows that there is room for two kinds of improvement to Extract Method tools. First, to prevent a large number of mis-selection errors, programmers need support in making a valid selection; an implementation of this is described and evaluated in our ICSE paper [15]. Second, to help programmers recover successfully from violated preconditions, programmers need expressive, distinguishable, and understandable feedback that conveys the meaning of precondition violations; this is the focus of the remainder of this article.

## 3 An Alternative to Textual Error Messages

We have built a plugin for the Eclipse environment that addresses the problems with error messages that were revealed by the formative study. The plugin is called Refactoring Annotations, and can be downloaded from http://multiview.cs.pdx.edu/refactoring/refactoring_annotations. In general, Refactoring Annotations can be thought of as graphical error messages; specifically, the current plugin displays violated preconditions for the Extract Method refactoring.

In our prototype, the programmer uses Refactoring Annotations by invoking a specific hotkey or toolbar button.

```
boolean areWheelsTrue(){

  Wheel front = bike.getFrontWheel();
  Wheel rear = bike.getRearWheel();

  boolean trued =isWheelTrue(front);
  trued = trued && isWheelTrue(rear);

  return trued;
}
```

Fig. 4. Refactoring Annotations overlaid on program text. The programmer has selected two lines (between the dotted lines) to extract. Refactoring Annotations show how the variables will be used: `front` and `rear` will be parameters, as indicated by the arrows into the code to be extracted, and `trued` will be returned, as indicated by the arrow out of the code to be extracted.

After doing so, the development environment visually annotates the currently selected text in the program editor. Thus, the programmer can use either Refactoring Annotations or refactoring error messages, via the standard EXTRACT METHOD tool. In a practical implementation, however, Refactoring Annotations would augment or replace the standard error messages, and thus would be shown just before the appearance of the EXTRACT METHOD configuration dialog. The prototype is currently not integrated in this way for two reasons. First, it allowed us to more easily perform a comparative evaluation, discussed in Section 4. Second, the prototype uses a data- and control-flow anlaysis engine that we have built ourselves, rather than the existing Eclipse precondition checking engine.

Refactoring Annotations overlay the program text to express control- and data-flow information, also known as program dependence graphs [4], in the programmer's selection. Each variable is assigned a distinct color, and each occurrence of the variable is highlighted, as shown in Figure 4. Across the top of the selection, an arrow points to the first use of a variable whose value will have to be passed as an argument into the extracted method. Across the bottom, an arrow points from the last assignment of a variable whose value will have to be returned. Variables that are declared or assigned to have black boxes around them. An arrow to the left of the code indicates that program control flows from the beginning of the selected code to end of it.

These annotations are intended to be most useful when preconditions are violated, as shown in Figure 5. When the selection contains assignments to more than one variable, multiple arrows are drawn leaving the bottom, showing multiple return values (Figure 5, top). When a selection contains a conditional return, an arrow is drawn from the return statement to the left, crossing the beginning-to-end arrow (Figure 5, middle). When the selection contains a branch (`break` or `continue`) statement, a line is drawn from the branch statement to its corresponding target (Figure 5, bottom). In each case, Xs are displayed over the

```
void goOnVacation(){

  Bike roadBike = getRoadBike();
  Bike mountainBike = getMtnBike();

  loadOnCar(roadBike,mountainBike);
}

boolean curbHop(int curbHeight){
  int hopHeight = liftFrontWheel();

  if(hopHeight<curbHeight){
    endo();
    return FAILURE;
  }

  liftRearWheel();
  return SUCCESS;
}

boolean goForRide(){

  while(!tired()){
    rotatePedals(10);
    if(this.hasCrashed())
      break;
  }

  return SUCCESS;
}
```

Fig. 5. Refactoring Annotations display three kinds of violated preconditions.

arrows, indicating the location of the violated precondition.

When code violates a precondition, Refactoring Annotations are intended to give the programmer an idea of how to correct the violation. Often the programmer can enlarge or reduce the selection to allow the extraction of a method. Other solutions include changing program logic to eliminate `break` and `continue` statements; this is another kind of refactoring.

Refactoring Annotations are intended to scale well as the amount of code to be extracted increases. For code blocks of tens or hundreds of lines, only a few values are typically passed in or returned, and only the variables holding those values are colored. In the case when a piece of code uses or assigns to many variables, the annotations become visually complex. However, we reason that this is desirable: the more values that are passed in or returned, the more coupled the extracted method is to its calling context [9]. Thus, we feel that code with visually complex Refactoring Annotations should probably not have EXTRACT METHOD performed on it. As one programmer has commented, Refactoring Annotations visualize a useful complexity metric.

Refactoring Annotations are intended to assist the programmer in resolving precondition violations in two ways. First, because Refactoring Annotations can indicate multiple precondition violations simultaneously, the annotations give the programmer an idea of the severity of the problem.

Correcting a conditional return alone will be easier than correcting a conditional return, and a branch, and multiple assignments. Likewise, correcting two assignments is likely to be easier than correcting six assignments. Second, Refactoring Annotations give specific, spatial cues that can help programmers to diagnose the violated preconditions.

Refactoring Annotations are similar to a variety of prior visualizations. Our control flow annotations are visually similar to Control Structure Diagrams [8]. However, unlike Control Structure Diagrams, Refactoring Annotations depend on the programmer's selection, and visualize only the information that is relevant to the refactoring task. Variable highlighting is like the highlighting tool in Eclipse, where the programmer can select an occurrence of a variable, and every other occurrence is highlighted. Unlike Eclipse's variable highlighter, Refactoring Annotations distinguish between variables by color; moreover, the variables relevant to the refactoring are highlighted automatically. In Refactoring Annotations, the arrows drawn on parameters and return values are similar to the arrows in the DrScheme environment [5], which draws arrows between a variable declaration and each variable reference. Unlike the arrows in DrScheme, Refactoring Annotations automatically draw a single arrow for each parameter and for each return value. Refactoring Annotations' data-flow arrows are like the code annotations drawn in a program slicing tool built by Ernst [3], where arrows and colors display the input data dependencies for a code fragment. While Ernst's tool uses more sophisticated program analysis than the current version of Refactoring Annotations, it does not include a representation of either output values or control flow. Finally, the EXTRACT METHOD tool in DevExpress' *Refactor! Pro* product (http://www.devexpress.com) visualizes parameters and return values as colored lines in the code editor, much like Refactoring Annotations. However, unlike Refactoring Annotations, this visualization is used for informational purposes during successful refactorings; it is not used to communicate violated preconditions.

Another alternative to displaying error messages is to automatically resolve the problem. Two examples of tools that take this approach are *Code Guide* (http://www.omnicore.com/en/codeguide.htm) and *Xrefactory* (http://www.xref.sk). For example, when multiple return values are detected, Xrefactory creates a new tuple class, creates and initializes a new tuple object with the return values, and extracts the values from the tuple in the caller. Resolving the problem automatically in this way has the disadvantage that the tool, rather than the programmer, chooses *which* resolution to apply. Such tools could conceivably give the programmer a choice of resolutions, but if there were a large number of possible resolutions, the programmer would be burdened with reading all of them. Instead, Refactoring Annotations take a different approach; give the programmer a detailed account of the problem, and then let her make an intelligent decision about how to resolve the problem.

# 4 EVALUATION

To evaluate whether Refactoring Annotations are more usable than their textual counterparts, we asked programmers to use both the standard error message dialogs and Refactoring Annotations (both in Eclipse) to identify problems in a code selection that violated preconditions of the EXTRACT METHOD refactoring. We evaluated subjects' responses for speed and correctness.

## 4.1 Subjects

We recruited subjects from the second author's object-oriented programming class. Students were given the option of either participating in the experiment or completing an alternative assignment on refactoring. 16 out of 18 students elected to participate; most had around 5 years of programming experience, although three had about 20 years. About half the students typically used integrated development environments such as Eclipse, while the other half typically used editors such as vi [10]. All students were at least somewhat familiar with the practice of refactoring.

## 4.2 Methodology

All participants used both precondition violation error messages and Refactoring Annotations. When each subject began this experiment, we showed the subject how the EXTRACT METHOD refactoring works using the standard Eclipse refactoring tool. We then demonstrated and explained each error message shown in a dialog box by this tool; this took about 5 minutes. We then told the subject that her task was to identify each and every violated precondition in a given code selection, assisted by the tool's diagnostic message. We then allowed the subject to practice using the tool until she was satisfied that she could complete the task; this usually took less than 5 minutes. The subject was then told to perform the task on 4 different EXTRACT METHOD candidates from different classes. Half of subjects used error messages with candidate set A, while half of subjects used error messages with set B. This proceedure was then repeated using Refactoring Annotations, where subjects who had used candidate set A with error messages then used set B with Refactoring Annotations, and subjects who used B with error messages then used A with Refactoring Annotations. This counter-balancing helped ensure that any performance differences that we observed were not due to differences in code.

## 4.3 Results

Table 1 counts two kinds of mistakes made by subjects. "Missed Violation" counts subjects who failed to recognize one or more preconditions that were being violated. "Irrelevant Code" counts subjects who identified some piece of code that was irrelevant to the violated precondition, such as identifying a `break` statement when the problem was a conditional `return`.

Table 1 indicates that programmers made fewer mistakes with Refactoring Annotations than with the error messages.

| Tool | Missed Violation | Irrelevant Code | Mean Identification Time |
|---|---|---|---|
| Error Message | 11 | 28 | 165 seconds |
| Refactoring Annotations | 1 | 6 | 46 seconds |

TABLE 1
Mistakes made by all subjects when finding problems during the EXTRACT METHOD refactoring, and the mean time taken to correctly identify all precondition violations. Smaller numbers imply better performance.

Using Refactoring Annotations, subjects were much less likely to miss a violation and much less likely to misidentify the precondition violations. The overall difference in the number of programmer mistakes per refactoring was statistically significant ($p = .003$, $df = 15$, $z = 2.95$, using a Wilcoxon matched-pairs signed-ranks test).

Table 1 also shows the mean time to find all precondition violations correctly, across all subjects. On average, subjects recognized precondition violations more than three times faster using Refactoring Annotations than using the error messages. The recognition time difference was statistically significant ($p < .001$ using a t-test with a logarithmic transform to correct long recognition time outliers).

Figure 6 shows the mean time to identify all precondition violations correctly for each tool and each user. Note that we omitted two subjects from the plot, because they did not correctly identify precondition violations for any code using the error messages. The dashed line represents equal mean speed using either tool. Since all subjects are below the dashed line, all subjects are faster with Refactoring Annotations. Most users were also more accurate using Refactoring Annotations.

Overall, Refactoring Annotations helped the subjects to identify every precondition violation in 45 out of 64 cases. In only 26 out of 64 cases did the error messages allow the subjects to identify every precondition violation. Subjects were faster and more accurate using Refactoring Annotations than using error messages.

We administered a post-test questionnaire that allowed the subjects to express their preferences for the two tools they tried. Significance levels are reported using a Wilcoxon matched-pairs signed-ranks test. Eleven out of 16 subjects ranked Refactoring Annotations higher than error messages with respect to helpfulness, while 12 of 16 ranked Refactoring Annotations higher with respect to likeliness to use again. Differences between the tools in helpfulness ($p = .003$, $df = 15$, $z = 3.02$) and likeliness to use ($p = .002$, $df = 15$, $z = 3.11$) were both statistically significant. Concerning error messages, subjects reported that they "still have to find out what the problem is" and are "confused about the error message[s]." In reference to the error messages produced by the Eclipse tool, one subject quipped, "who reads alert boxes?"

Overall, the subjects' responses showed that they found the Refactoring Annotations superior to error messages for the tasks given to them. More importantly, the responses also showed that the subjects felt that Refactoring Annotations would be helpful during their normal programming activities. The experimenter's notebook from this experiment can be found in Appendix 1, while raw data can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

### 4.4 Threats to Validity

Although the quantitative results discussed in this section are encouraging, several limitations must be considered when interpreting them. One major limitation is that every subject first used the Eclipse error messages and then used Refactoring Annotations; the fixed order may have biased the results to favor Refactoring Annotations due to a learning effect. A second limitation is that the students were volunteers, and may not be representative of programmers in general. A third limitation is that the code samples selected for the experiment may not be representative of code that is the subject of a refactoring tool. Finally, because this experiment only evaluates EXTRACT METHOD, the results may not generalize to other refactorings. We address this last threat in Section 7.

### 4.5 Discussion

The results of the experiment suggest that Refactoring Annotations are preferable to an error-message-based approach for showing precondition violations during the EXTRACT METHOD refactoring. Furthermore, the results indicate that Refactoring Annotations communicate the precondition violations effectively. When a programmer has a better understanding of refactoring problems, we believe that the programmer is likely to be able to correct the problems and successfully perform the refactoring.

## 5 GUIDELINES FOR PRESENTING ERRORS

Although programmers using Refactoring Annotations performed significantly better than those using error messages in our study, our tool is limited to one refactoring in one IDE for one language. How can we generalize this work to help solve the refactoring error problem for EXTRACT METHOD in other contexts, to cover other refactorings, and perhaps to apply to other kinds of error reporting in an interactive development environment?

By comparing how programmers use both error messages and Refactoring Annotations to understand the causes of violated refactoring preconditions, we have induced a number of usability guidelines for refactoring tools. We feel that these guidelines capture the essential attributes of Refactoring Annotations, and will help future toolsmiths build more usable representations for refactoring errors, and perhaps for other kinds of errors.

In this section, we make observations about the experiment or about the design of Refactoring Annotations, and then present the guideline that is intended to capture those observations.

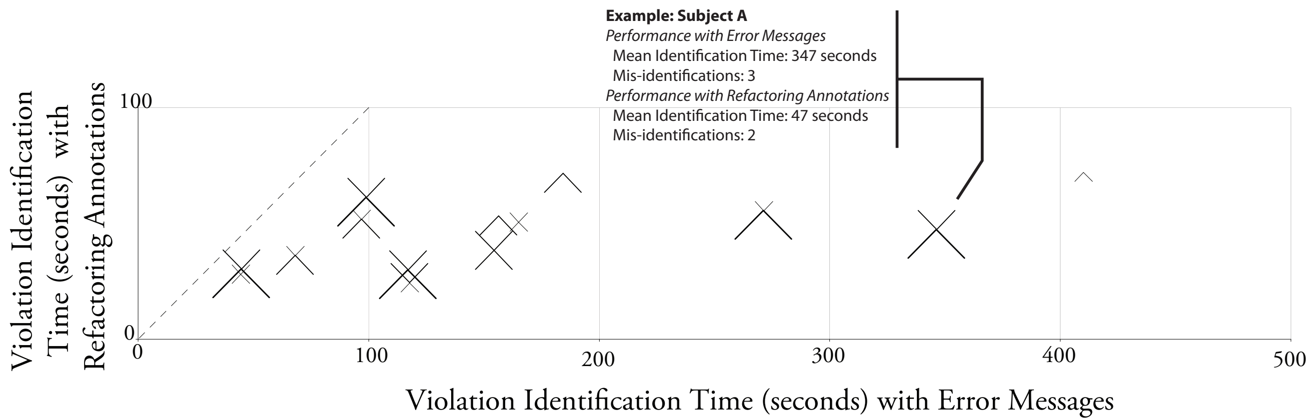


Fig. 6. For each subject, mean time to identify precondition violations correctly using error messages versus Refactoring Annotations. Each subject is represented as an X, where the distance between the bottom legs represents the number of imperfect identifications using the error messages and the distance between the top arms represents the number of imperfect identifications using Refactoring Annotations.

**Expressiveness.** During the experiment with error messages, programmers invested significant time deciphering the message. Refactoring Annotations reduced that time. Thus:

- ◇ A representation of a refactoring error should help the programmer to comprehend the problem quickly by clearly expressing the details: the programmer should not have to spend significant time understanding the cause of an error.

**Locatability.** Because Refactoring Annotations colored the location of precondition violations in the editor, programmers in our experiment could locate problems quickly and accurately. In contrast, with error messages, programmers were forced to find the locations of violated preconditions manually. A tool should tell the programmer what it just discovered, rather than requiring the programmer "to basically compile the whole snippet in my head," as one Eclipse bug reporter complained regarding an EXTRACT METHOD error message [1]. Thus:

- ◇ A representation of a refactoring error should indicate the location(s) of the problem.

**Completeness.** Refactoring Annotations show all errors at once: during the experiment, this allowed programmers to find all violated preconditions quickly. In contrast, programmers who used error messages had to fix one violation before they could find the next. Thus, to help programmers assess the severity of the problem:

- ◇ A representations of a refactoring error should show all problems at once.

**Estimability.** The number of Xs in a Refactoring Annotation gives the programmer a visual estimate of the severity of the problem; error messages do not. For instance, Eclipse's error message did not indicate how many values would need to be returned from an extracted method, just that the number was greater than one. The programmer should be able to tell quickly whether a violation means that the code can be refactored after a few minor changes, or whether the refactoring is nearly hopeless. Thus:

- ◇ A representation of a refactoring error should help the programmer to estimate the amount of work required to fix violated preconditions.

**Relationality.** Violations often arise from the relationship between several pieces of code, not from a feature at a single place. Refactoring Annotations relate the declaration of and references to a variable using a specific color; this allows the programmer to analyze the problem one variable at a time. More generally, relations can be represented using, for example, arrows and colors. Thus:

- ◇ A representation of a refactoring error should display information relationally, when this is appropriate.

**Perceptibility.** Xs in the Refactoring Annotations allowed programmers to quickly distinguish errors from other types of information. Programmers were not left to wonder whether there was a problem with the refactoring. Thus:

- ◇ Representations of refactoring errors should allow programmers to easily distinguish precondition violations (showstoppers) from warnings and advisories.

**Distinguishability.** In the experiment, programmers using error messages conflated one kind of violation with another. This wasted their time because they tried to track down violations that did not exist. Programmers using Refactoring Annotations, which use distinct representations for distinct errors, rarely conflated different kinds of violation. Thus:

- ◇ Representations of refactoring errors should allow the programmer to distinguish easily between different kinds of violation.

Each of our guidelines has precedent in prior work in the domain of usability for user-interfaces. Expressiveness, Locatability, and Relationality are similar to Shneiderman's recommendation that error messages be specific rather than general, so that the user understands the cause of the error [21, p. 59]. Likewise, Locatability, Completeness, and Estimability are all designed to achieve Shneiderman's recommendation for constructive guidance, so that the user

can successfully recover from an error [21, p. 58]. **Perceptibility** and **Distinguishability** are similar to Nielsen's "Help users recognize ... errors" and "consistency and standards;" the latter states that "users should not have to wonder whether different words, situations, or actions mean the same thing" [17].

This set of guidelines is complementary to Mealy and colleagues' existing guidelines for presenting refactoring errors [13, Appendix C.2]. Because Mealy and colleagues' guidelines were developed by distilling and augmenting existing sets of general usability guidelines, the focus of Mealy and colleagues' guidelines are different. Specifically, while our guidelines focus on making the content of error representations better, Mealy and colleagues tend to focus on avoiding, tolerating, and recovering from errors. The guideline E3 is an exception, "Provide understandable, polite, meaningful, informative error messages," which is comparable to our Expressiveness guideline.

# 6 A Taxonomy of Refactoring Precondition Violations

One of the motivations for introducing the above guidelines was to generalize our work to precondition violations for other refactorings. To assess how well we have succeeded, we now characterize precondition violations for a wide variety of refactorings and explain how our guidelines can be applied to them.

## 6.1 Methodology for Deriving a Precondition Taxonomy

In order to draw general conclusions about how to represent precondition violations, we classified all the precondition violations that are detected by four different refactoring tools, each for a different language.

- **Eclipse JDT.** This is the standard Eclipse Java refactoring tool (http://www.eclipse.org/jdt). We gathered precondition violations from a key-value text file used to store each precondition error message. This is the most mature refactoring tool of the four we studied, containing 537 error messages in total. We inspected a version from the Eclipse CVS repository checked in on 4 August 2008.
- **Eclipse CDT.** This tool is built for C++ as a plugin to the C/C++ environment for Eclipse (http://r2.ifs.hsr.ch/cdtrefactoring). We gathered precondition error messages in the same way as with Eclipse JDT. This refactoring tool contained a total of 77 error messages. We inspected version "0.1.0.qualifier".
- **Eclipse RDT.** This tool is built for the Ruby environment for Eclipse (http://rubyeclipse.sourceforge.net). We gathered precondition error messages in the same way as with Eclipse JDT, although the error messages were spread across several files. This refactoring tool contained a total of 73 error messages. We inspected Subversion revision 1297.
- **HaRe.** This tool is built for refactoring Haskell programs (http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html). We gathered precondition error messages by searching for invocations of the `error` function, which was typically followed by an error message that indicated a violated refactoring precondition. This refactoring tool contained a total of 204 error messages. We inspected the 27 March 2008 update of HaRe 0.4.

To create the taxonomy, the first author of this paper served as the coder, manually classifying messages from these four projects. We did not attempt to automate this process because we noticed that many error messages could practically refer to the same problem, yet included significantly different text. The four error messages in Section 6.2.1 exhibit this problem.

The coder categorized the messages in the following manner. First, the coder classified messages based on what he believed was the root cause of error, assigning each error message to an emergent category. The coder then identified super-categories that encapsulated several subcategories; a few messages were ambiguous in their root cause, but could be placed into a super-category. The coder finally re-read each error message to make sure that it appeared in its appropriate category in the taxonomy.

## 6.2 Taxonomy Description

Table 2 displays our taxonomy. Categories are indented when they are a subcategory; for instance, *inaccurate analysis* is a kind of *analysis problem*. Note that the number of error messages in each taxonomy category is not indicative of the importance of a particular category. This is because some general error messages that apply to several refactorings appear just once, and also because some tool categories are unpopulated because of the relative immaturity of the tool.

Due to space constraints, we cannot describe each category and how we applied our guidelines to it. Instead, in Sections 6.2.1 through 6.2.3, we explain three of the categories, give example error messages, and describe how the guidelines apply in each category. We also provide mockups of how Refactoring Annotations can be extended to precondition violations in the taxonomy. An explanation of every category listed in Table 2 can be found elsewhere [14, p. 155–184]. Additionally, the reader can find the original error messages along with their categories at http://multiview.cs.pdx.edu/refactoring/error_taxonomy.

We found that the guidelines applied to all categories except *inaccurate analysis*, *internal errors*, *identity configuration*, *property*, *vague*, and *unknown*. These errors in these categories were either too diverse to say generally how the guidelines apply, or deeply tried to explain the inner workings of the refactoring tool itself. Guidelines that address the latter type of errors remain future work.

### 6.2.1 *Precondition Category: Illegal Name*

*Illegal name* occurs when a programmer is choosing a name for a program element to be created, but that name violates the rules of the programming language.

| Category | JDT | CDT | RDT | HaRe |
|---|---|---|---|---|
| analysis problem | 0 | 0 | 0 | 0 |
| inaccurate analysis | 35 | 4 | 2 | 2 |
| incompatibility | 5 | 1 | 0 | 0 |
| compilation errors | 27 | 1 | 3 | 0 |
| internal error | 24 | 5 | 0 | 36 |
| inconsistent state | 15 | 2 | 0 | 0 |
| unsaved | 4 | 1 | 0 | 0 |
| deleted | 4 | 0 | 0 | 0 |
| misselection | 0 | 0 | 0 | 0 |
| selection not understood | 30 | 26 | 19 | 33 |
| improper quantity | 0 | 5 | 0 | 2 |
| misconfiguration | 3 | 0 | 0 | 0 |
| illegal name | 6 | 7 | 1 | 15 |
| unconventional name | 11 | 0 | 0 | 0 |
| identity configuration | 4 | 0 | 7 | 3 |
| unbound configuration | 7 | 2 | 0 | 2 |
| unchangeable | 3 | 0 | 0 | 0 |
| unchangeable reference | 2 | 0 | 0 | 0 |
| unchangeable source | 16 | 0 | 0 | 0 |
| unchangeable target | 12 | 0 | 0 | 0 |
| unbinding | 12 | 0 | 0 | 1 |
| control unbinding | 35 | 2 | 4 | 10 |
| data unbinding | 18 | 5 | 3 | 4 |
| name unbinding | 20 | 0 | 0 | 2 |
| inheritance unbinding | 11 | 0 | 1 | 0 |
| clash | 6 | 5 | 0 | 24 |
| control clash | 17 | 3 | 5 | 9 |
| data clash | 16 | 0 | 3 | 3 |
| name clash | 38 | 3 | 0 | 2 |
| inheritance clash | 9 | 0 | 0 | 0 |
| inherent | 0 | 0 | 0 | 0 |
| context | 38 | 0 | 7 | 4 |
| own parent | 4 | 0 | 0 | 0 |
| structure | 17 | 0 | 13 | 9 |
| property | 45 | 3 | 3 | 0 |
| vague | 37 | 1 | 0 | 22 |
| unknown | 6 | 1 | 2 | 21 |

TABLE 2
Our taxonomy of precondition violations (column 1), with counts and bars indicating the number of error messages in each category for each refactoring tool (columns 2–5).

Example: *Illegal Name* Errors

| Tool | Refactoring | Message |
|---|---|---|
| JDT | Multiple | Type name cannot contain a dot (.). |
| CDT | Multiple | ____ contains an unidentified mistake. |
| RDT | RENAME | Please enter a valid name for the variable. |
| HaRe | RENAME | The new name should be an operator! |

The **expressiveness** of representations of an *illegal name* violation can be improved by indicating what character or character combinations are invalid and, if possible, what characters are valid. **Locatability** can be improved by pointing at which entered character or characters are invalid; **estimability** can be improved by pointing at each and every invalid character. Figure 7 shows an example of what such a user interface might look like.

### 6.2.2 Precondition Category: Inheritance Unbinding

*Inheritance unbinding* occurs when the refactoring tool tries to modify some code that contains inheritance relationships, but doing so would break those relationships.

Example: *Inheritance Unbinding* Errors

| Tool | Refactoring | Message |
|---|---|---|
| JDT | RENAME FIELD | Cannot be renamed because it is declared in a supertype |
| RDT | INLINE CLASS | The inlne [sic] target is subclassed and thus cannot be inlined. |

**Estimability** can be improved by showing the number of problematic inheritance relationships. **Relationality** can be improved by extending the top/bottom metaphor used in the data flow indicators in Refactoring Annotations for EXTRACTMETHOD (as in Figure 4). In the case of inheritance, “up” would mean superclass or supertype, while “down” would mean subclass or subtype. Thus, precondition violations relating to supertypes or superclasses could use an “up” relationship, and vice-versa. Xs could then be placed where such relationships are broken, and the programmer could click on the unattached end of the relationship arrow to go the source. This could improve **locatability** as well, since the programmer would be able to use the relation to navigate to the relevant program elements. Figure 8 shows an example mockup for an unsuccessful MOVE METHOD refactoring.

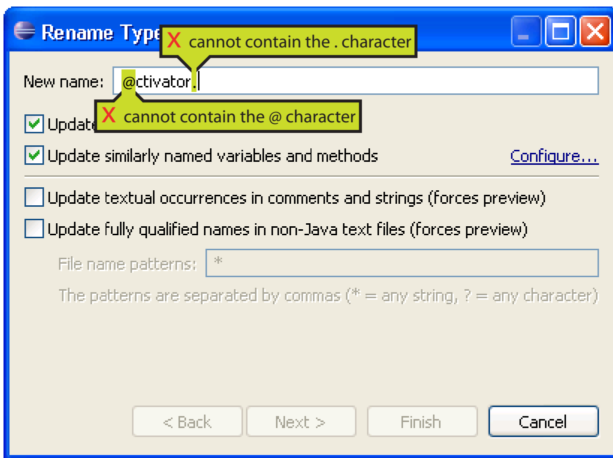


Fig. 7. How *illegal name* violations could be implemented following our guidelines. The green violation indicators indicate that two invalid characters were typed into the new name text field. The original Eclipse error message states only "Type name cannot contain a dot (.)."

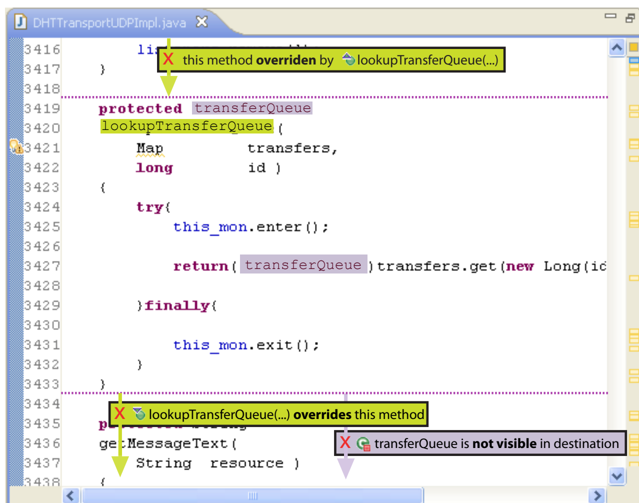


Fig. 8. A mockup of how the guidelines inform the display of *name unbinding* (in purple) and *inheritance unbinding* (in green) for an attempted MOVE METHOD refactoring, where the destination class is the class of `this_mon`. The purple annotations indicate that this method relies on a field `transferQueue`, which will not be accessible in the destination. The green annotations indicate that the current `lookupTransferQueue` method overrides a superclass method (top) and a subclass method (bottom), so the method cannot be moved.

### 6.2.3 Precondition Category: Structure

*Structure* violations occur when a refactoring cannot proceed because of the structure of the program element being refactored.

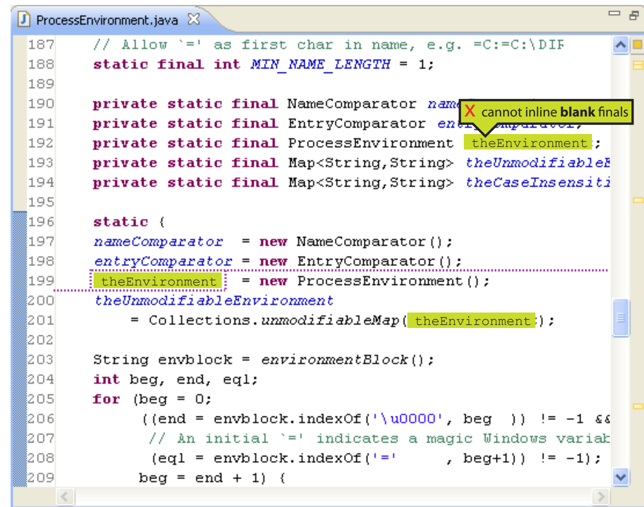


Fig. 9. A mockup of how the guidelines inform the display of *structure* for an attempted INLINE CONSTANT refactoring, pointing out that the selected constant `theEnvironment` is blank, meaning that it is not assigned to at its declaration. The original Eclipse modal error message states "Inline Constant cannot inline blank finals."

Example: *Structure* Errors

| Tool | Refactoring | Message |
|---|---|---|
| JDT | INLINE METHOD | Method declaration contains recursive call. |
| RDT | MERGE WITH EXTERNAL CLASS PARTS | There is no class in the current file that has external parts to merge |
| HaRe | MERGE DEFINITIONS | The guards between the two functions do not match! |

**Locatability** can be improved by making the structure of the program element to be refactored more explicit. **Relationality** can be improved by relating relevant pieces of the structure to one another; Figure 9 shows an example. The **expressiveness** of Figure 9 might also be improved by an explanation of the term "blank final."

### 6.2.4 Limitations to the Taxonomy

We should be aware of several caveats before drawing conclusions about how our guidelines apply to the taxonomy of refactoring preconditions.

First, the taxonomy is imperfect. Although it is a best-effort attempt to classify real-world error messages, similar error messages may appear in different categories. The size of the error message corpus (891 messages in total) combined with the textual inconsistencies between messages meant that doing a completely accurate classification was nearly impossible. For example, several completely different EXTRACT METHOD error messages appear to refer to the same violation: "No statement selected," "There is nothing to extract," and "Cannot extract a single method name." While we assumed that these messages have identical causes, they may refer to subtly different errors. In general, messages are difficult to classify because they can be stated in ways that are positive or negative, constructive

or declarative, programmer-oriented or tool-oriented.

Second, some messages could conceptually be placed in more than one category. For example, the error message "Removed parameter ____ is used in method ____ declared in type ____" could appear in either *inheritance unbinding* or in *data unbinding*, because it refers to both types and data flow issues. However, we placed each message into exactly one category. Moreover, because we created the taxonomy as we inspected the error messages, the category into which each message was placed depended on what categories were known when we inspected each message; a slightly different categorization might have resulted had we inspected the messages in a different order.

# 7 EVALUATION

Thus far, we have discussed how our guidelines for improving the representation of precondition violations *can* be applied to a variety of violations. Now we address the question of whether they *should* be applied: is there reason to believe that applying the guidelines will lead to an improvement over the standard error message? In this section, we describe a paper-based experimental evaluation that suggests that following the guidelines does indeed improve usability.

This experiment is similar to the previous experiment (Section 4) in that we asked programmers to use both a new method of communicating error information, which we call Refactoring Annotations, and conventional error messages, to diagnose violations of refactoring preconditions. We compared their ability to identify the source of those violations correctly, and asked for their opinions of both tools. Unlike the previous experiment, it explores the use of Refactoring Annotations for several refactorings, instead of for just EXTRACT METHOD. Another difference from the previous experiment is that in this evaluation we did not train subjects with either kind of error indication. The experimenter's notebook from this experiment can be found in Appendix 2, while raw data can be found at http://multiview.cs.pdx.edu/refactoring/experiments. We now describe the experiment in more detail.

## 7.1 Subjects

We drew subjects from three upper-level graduate and undergraduate courses: Scholarship Skills, Advanced Programming, and Languages and Compiler Design. We encouraged students to participate in the experiment by offering 10 dollar gift cards from the Portland State bookstore to participants, requiring only that the participants had previously programmed in Java. Thirteen students volunteered to participate, but two had scheduling conflicts and one was somewhat familiar with our research: these three were excused. As a result, a total of 10 students participated.

Subjects reported a mean of about 6 years of programming experience and 19 hours per week of programming over the last year. Eight of the subjects used Integrated Development Environments at least part of the time when programming: these environments included Visual Studio, Eclipse, Netbeans, and Xcode. Six subjects were at least somewhat familiar with the concept of refactoring, and two of them had used refactoring tools. All subjects were at least somewhat familiar with Java.

## 7.2 Methodology

We randomly placed subjects into one of four groups to ensure that average task difficulty was balanced, as shown in Table 3. Half of the subjects used Refactoring Annotations to help them diagnose violated preconditions on 8 individual refactorings during the first phase of the experiment; they then used error messages to help diagnose violated preconditions on 8 other individual refactorings in the second phase. The other half used error messages in the first phase, then Refactoring Annotations in the second phase. Additionally, half of the subjects analyzed violations in one order (call it "A") in the first phase, then in another order (call it "B") in the second phase, and vice-versa for the other half of subjects. Of the 10 subjects who participated, we assigned two to Group 1, three to Group 2, two to Group 3, and three to Group 4.

We chose 3 kinds of refactoring for subjects to analyze; the number 3 was an attempt to balance having a sufficient variety of refactorings with having few enough that subjects would not find it difficult to remember how the refactorings worked. We chose the refactoring kinds RENAME, EXTRACT LOCAL VARIABLE, and INLINE METHOD because they are currently among the most popular refactorings performed with tools [16]. Within these kinds of refactoring, we selected precondition violations using the following criteria:

- The chosen violations should span several precondition categories (Table 2), so that we evaluate a substantial cross-section of the precondition taxonomy.
- Preference should be given to violation categories to which the guidelines apply, because when the guidelines do not apply, error messages and Refactoring Annotations do not differ. Thus by including only categories for which the guidelines apply, the results of the experiment will evaluate the difference between the two violation representations.
- Some violation categories should appear more than once, for different refactorings. By doing this, we hoped to simulate the situation where a subject understood a precondition violation for one refactoring, and could transfer that understanding to a violation during another refactoring.
- Some refactorings should violate two different preconditions at the same time, to provoke the situation where the refactoring tool informs the programmer of only the first violation that it finds (Section 2).

Table 4 displays the precondition violations that we chose. The Refactoring Kind column refers to one of the three chosen kinds of refactorings, Category refers to the category in which the violation is found, and Message lists the specific error message that the Eclipse refactoring tool displays when that violation occurs. Refactoring number 3

| | Group 1 | | Group 2 | | Group 3 | | Group 4 | |
|---|---|---|---|---|---|---|---|---|
| | Tool | Order | Tool | Order | Tool | Order | Tool | Order |
| Phase 1 | Refactoring Annotations | A | Refactoring Annotations | B | Error Messages | A | Error Messages | B |
| Phase 2 | Error Messages | B | Error Messages | A | Refactoring Annotations | B | Refactoring Annotations | A |

TABLE 3
The order in which the four groups of subjects used the two refactoring tools over the two code orders.

| Refactoring Kind | Refactoring Number | Category | Message |
|---|---|---|---|
| RENAME | 1 | Data Clash | A field with this name is already defined. |
| | 2 | Illegal Name | Type name cannot contain a dot (.). |
| INLINE METHOD | 3 | Structure | Method declaration contains recursive call. |
| | | Inheritance Unbinding | Method to be inlined implements method from interface ____. |
| | 4 | Property | Cannot inline abstract methods. |
| | 5 | Stucture | Cannot inline a method that uses qualified this expressions. |
| EXTRACT LOCAL | 6 | Illegal Name | ____ is not a valid identifier. |
| | 7 | Data Clash | A variable with name ____ is already defined in the visible scope. |
| | 8 | Context | Cannot extract assignment that is part of another expression. |

TABLE 4
Refactorings and precondition violations used in the experiment.

(in the third row of Table 4) violates two different preconditions; Eclipse version 3.2 reports only the upper one to the programmer.

We first randomized the order of appearance of the three kinds of refactorings (RENAME, INLINE METHOD, and EXTRACT LOCAL) and then randomized the order of each kind. In other words, we did not mix the different kinds of refactorings to reduce the likelihood that subjects would be confused about which refactoring was being performed. The two orders in which subjects were asked to diagnose violations were A=(1,2,7,6,8,5,3,4) and B=(1,2,4,3,5,8,6,7). We selected example code for subjects to refactor (and thus to encounter the violations) from PyUnicode, a unicode Java class from the Jython project (described in Section 2), revision number 5791. This class contains 5 private inner classes that can be used to iterate over collections. This code was selected because 5 of the 9 precondition violations shown in Table 4 could naturally arise when refactoring that code. We manually changed the code in two ways:

- we inserted code that would cause the programmer to violate the remaining four preconditions when refactoring, and
- we changed code to avoid making the cause of a violation trivially apparent. For example, if the error message says “A field with this name is already defined,” then the field with the same name should not appear directly adjacent to the renamed field.

This code spanned 202 lines, small enough to fit on three 8.5×11 inch sheets of paper, but large enough to generate two violations of each refactoring precondition in Table 4 (one for error messages, one for Refactoring Annotations).

### 7.3 Example Experiment Run

When a subject arrived to participate in the experiment, the first author, acting as experiment administrator, offered the subject a refreshment and gave her a letter of informed consent. The administrator then asked her to complete a pre-experiment questionnaire in which she noted her programming and refactoring experience.

The administrator gave the subject a brief overview of the experiment, and a short review of the 3 refactorings. The administrator then told her that she was going to see 16 attempted refactorings on the same code base, but that none of these refactorings would be successful. Instead, the tool would indicate why the refactoring could not be performed, using either an error message in a dialog box or a graphical annotation on top of the code. The administrator told the subject that her task was to diagnose the violated precondition, indicate the pieces of relevant source code, and give an explanation of the problem.

Assume, for example, that a subject is assigned to Group 3 (see Table 3), and thus uses error messages first with violation ordering A, then uses Refactoring Annotations with violation ordering B. For the first task, the administrator gives the subject the code in the form of 3 pieces of 8.5×11 inch paper, glued vertically on a flip chart. Figure 10 depicts a simulated experiment situation. The administrator points out what piece of code the programmer selected, which refactoring was attempted, and the error message that the programmer encountered. This error message is also paper-based, printed on top of code without obscuring relevant parts of the program. The administrator then asks the subject to place small sticky notes next to the places in the code, or in the refactoring tool configuration window, that they feel are responsible for the violation, and to explain their actions aloud.

In this case, the first error message encountered is number 1 (Table 4), and a correct answer is to place a sticky note next to the declaration of the field with which the new name clashes. If the sticky note placement is ambiguous, the

Fig. 10. A simulation of an experimental run. The experiment subject (at left), considers where to place a sticky note on the code responsible for the violation. The administrator (at right), records observations about the participant's reasoning regarding where the subject places the note.

subject is asked to both clarify her intent verbally and more precisely place the sticky note. For refactoring numbers 2 and 6, the correct answer is to place the sticky note next to each illegal character; for 3, next to the recursive call and method declaration in the interface; for 4, next to the abstract method declaration; for 5, next to the qualified this expression; for 7, next to the existing variable declaration; and for 8, next to the equal sign in the assignment.

The subject then indicates that she is satisfied with her response, and moves on to the next individual refactoring. This process is repeated with 7 additional individual refactorings. To switch to the next refactoring, the administrator switches to another paper copy of the code, where each copy is printed with a different refactoring dialog box. The administrator records the total time taken for the subject to complete the 8 tasks.

The process is repeated 8 more times, but with Refactoring Annotations printed on top of the code instead of error messages. The same code base is used, again printed on three sheets of paper. The same refactorings are applied to different pieces of the code to mitigate learning effects. Precondition violations are presented to the subject according to ordering B.

After the tasks were complete, the administrator used a post-experiment questionnaire to solicit the subject's opinions of the tools; this was followed by a brief interview. The subject was then thanked and released.

### 7.4 Results

Eight out of 10 subjects reported that Refactoring Annotations helped them understand violations better than error messages. The difference between subject ratings is statistically significant ($p = .046$, $df = 9$, $z = 2.00$)[1]. The measured results confirm this opinion; Table 5 shows the total number of locations that subjects did not mark with a sticky note (Missed Location), as well as the number of irrelevant code fragments that subjects did mark with a sticky note (Irrelevant Location). The difference in missed locations was statistically significant ($p = .007$, $df = 9$, $z = 2.70$) as was the difference in choosing irrelevant locations ($p = .017$, $df = 9$, $z = 2.39$).

| Tool | Missed Location | Irrelevant Code | Mean Identification Time |
|---|---|---|---|
| Error Message | 54 | 19 | *109 seconds |
| Refactoring Annotations | 23 | 4 | 83 seconds |

TABLE 5
The number and type of mistakes made when diagnosing violations of refactoring preconditions, for each tool. The right-most column lists the mean time spent diagnosing preconditions for each of the 8 refactorings. Subjects diagnosed errors in a total of 80 refactorings with each tool. Smaller numbers indicate better performance.

| Refactoring Number | Error Messages ML | Error Messages IC | Refactoring Annotations ML | Refactoring Annotations IC |
|---|---|---|---|---|
| 1 | 4 | 2 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 10 | 3 | 5 | 0 |
| 4 | 4 | 3 | 1 | 0 |
| 5 | 3 | 2 | 3 | 2 |
| 6 | 3 | 2 | 0 | 1 |
| 7 | 4 | 1 | 1 | 0 |
| 8 | 10 | 3 | 8 | 1 |

TABLE 6
The number of subjects making at least one missing location (ML) and irrelevant code (IC) mistake for each refactoring. Smaller numbers indicate better performance.

Six out of 10 subjects reported that they would be more likely to use Refactoring Annotations than error messages and 4 out of 10 said that they would be equally likely to use either. The difference between subject ratings for the two tools is statistically significant ($p = .026$, $df = 9$, $z = 2.23$).

Nine out of 10 subjects reported that they felt that Refactoring Annotations helped them figure out what went wrong faster than with error messages. The difference between subject ratings is statistically significant ($p = .026$, $df = 9$, $z = 2.26$). The measured results confirm this opinion; on average subjects took $(109 - 83)/109 = 24\%$ less time with Refactoring Annotations than with error messages (Table 5). The asterisk * indicates that a timing was not obtained for one subject, so we could not include it in

1. This and the the remaining significance tests in this article were performed using a Wilcoxon matched-pairs signed-ranks test.

the mean. However, this difference was not statistically significant ($p = .051$, $df = 8$, $z = 1.96$), probably because the size of the effect depended on which tool the subject used first. When a subject used Refactoring Annotations first and then used error messages, on average she took about 3% less time using Refactoring Annotations. When a subject used error messages first, on average she took about 41% less time using Refactoring Annotations.

Five out of 10 subjects reported that they felt that Refactoring Annotations made them more confident of their judgements than error messages, and 4 out of 10 said that they were equally confident with either tool. The difference between subject ratings is not statistically significant ($p = .084$, $df = 9$, $z = 1.73$). The one subject that said that she was less confident using Refactoring Annotations said that the reason was that "they give such great information, I feel like, 'Am I missing something?"'

The subjects' ability to identify the causes of violations varied from refactoring to refactoring, as shown in Table 6. Refactorings 1 and 7 in Table 4, in which the programmer attempts to make a new program element that conflicts with an existing element, were usually understood with both error messages and Refactoring Annotations. However, subjects were sometimes unable to *find* the existing program element. It appeared that this task was difficult because subjects generally performed linear searches through the given code, which not only took a significant amount of time, but often required repeated passes over the same code until they found the conflicting element. Several subjects mentioned that they would usually enlist the help of a find tool, such as Unix `grep`, to find candidates and then sort through those candidates manually to find the conflicting element. While useful, this technique would likely include false positives. Two subjects mentioned that they would use Eclipse's "Open Declaration" tool to help in the task, but this would be possible only after they had found a reference to the conflicting element in the code.

Refactoring 2, in which an illegal identifier was typed that contained two dots, and refactoring 6, where an illegal identifier was typed that contained a # or @ sign and began with a number, were sometimes problematic for subjects, especially those using error messages. With Refactoring Annotations, only once did a programmer select some irrelevant code when she thought there was a problem with the original code selection (she apparently saw a == sign, interpreted that as assignment, and thought that there was a violation similar to that shown in refactoring 8). With error messages, 3 subjects noticed that the # or @ was a problem, but failed to also notice that the identifier started with a digit. Moreover, 2 subjects erroneously thought that Java identifiers could not contain *any* digits, and thus incorrectly said that digits inside an identifier were a problem. One programmer said that she would use Google to find out whether # and @ were legal characters in Java identifiers.

In refactoring 3, INLINE METHOD is attempted on a method that contained two recursive calls. The refactoring also attempted to remove the inlined method, which was a problem because it implemented a method from the `Iterator` interface; if it were deleted, the containing class would no longer satisfy the `Iterator` interface, and a compilation error would be produced. Subjects rarely identified the problems in this refactoring perfectly. With both Refactoring Annotations and error messages, most subjects appeared to understand why recursion was a problem. However, finding both recursive calls was much more difficult with error messages. With error messages, 7 subjects missed at least one recursive call; with Refactoring Annotations, only one subject missed the recursion, although that person missed both calls. As for the method implementing a method from an interface, no programmer using error messages noticed this problem. This appeared to be because the refactoring tool didn't tell them about it; it told them only about a recursive call. When using Refactoring Annotations, five subjects noticed the problem with the interface, although only one of them could provide a coherent explanation of why it was a problem.

Refactoring 4, inlining an abstract method, appeared to be generally understood by all subjects, regardless of whether they used Refactoring Annotations or error messages. However, subjects occasionally could not locate the abstract method, four times with error messages and once with Refactoring Annotations.

Refactoring 5 was about inlining a method that contains a qualified `this` expression (such as `ClassName.this.member`). On this task, subjects performed roughly equivalently with Refactoring Annotations and with error messages. With both tools, 7 out of 10 subjects were able to locate the problem, but apparently neither representation of the error was expressive enough to help subjects understand why refactoring that code was a problem. This may be because the qualified `this` notation is relatively obscure. Only one programmer was able to explain the problem correctly; the others who successfully located the problem appeared to have guessed.

On refactoring 8, extracting a local variable from an expression containing an assignment, subjects performed about equivalently with both Refactoring Annotations and error messages. It appeared that neither representation helped programmers to understand the problem.

When we interviewed subjects after the experiment, all of them seemed to prefer Refactoring Annotations, although to differing degrees. Subjects described them as "unobtrusive," "a little more helpful," "more informative," as giving "more specific information," as being "across the board helpful," as "sav[ing me] some seconds," as showing "more errors, more like a compiler," as indicating "where the problem is," and "as mak[ing] refactoring part of my usual error fixing strategy: read all, fix all at once." Subjects disliked error messages because they "cover too much area," "tend to get in the way," required the programmer to scan the code manually, and because "they are modal [and they say]: 'Tackle Me before You Do Anything Else!"'

### 7.5 Discussion

The results of the evaluation suggest that these generalized Refactoring Annotations are preferred by programmers, and help them find the causes of precondition violations more accurately and more quickly, when compared to error messages. Because our paper prototypes of Refactoring Annotations were designed according to the guidelines discussed in Section 5, these results also suggest that the guidelines can help improve the usability of refactoring tool precondition violations.

For refactorings 5 and 8, Refactoring Annotations were not sufficiently detailed to help subjects understand why the tool could not perform the refactoring. It appears that a more detailed explanation of the problem might be helpful, including descriptions of language semantics. For instance, a help document might be useful for explaining what a qualified `this` statement is, and what the consequences would be if the method containing it were to be inlined. However, the fact that programmers sometimes do not spend much time trying to understand a single sentence about a refactoring error (Section 2) suggests that they may not spend any time reading a detailed help document.

Reflecting upon the experiment, it appears that refactoring 5, which we classified as a *structure* violation, might better be classified as a *name unbinding* violation. Why? A qualified `this` statement cannot be inlined largely because it literally references a member of a containing class, and that member might not be accessible at all of the places where the method would need to be inlined.

Thus, future replications of this experiment might show that subjects understand this violation better if it is represented as a *name unbinding*. This is because our guidelines suggest that *name unbinding* (unlike *structure*) should display what program element the refactored code is being unbound from; this information may help programmers better understand the violation.

### 7.6 Threats to Validity

While the results of this experiment encouraging, there are several threats to the validity. First, we chose code by hand and modified it so that it would cause a violation of at least one refactoring precondition. So while the original code was selected from open-source code bases, the code may not be representative of code found in the wild. Second, subjects were volunteers from several classes at Portland State University, and may not be representative of the average programmer. Third, we sampled categories of errors from the taxonomy for the experiment by hand; programmers' performance may be different when presented with other categories of errors in the taxonomy. Fourth, the task was given to the subjects on paper; their behavior in a real development environment might well be different. Fifth, when placing sticky notes on code to indicate the location of violations, subjects may have simply been parroting what Refactoring Annotations (and to a lesser extent, error messages) were telling them, without any real understanding of the violation. To mitigate this threat, we had subjects briefly explain *why* they put a sticky note where they did, and tried to discern to what extent they understood the problem. However, we observe that even if a programmer does not understand a violation fully, locating the pieces of code that are part of the problem is valuable because it focuses the programmer's attention on the relevant code. Finally, we observed a strong learning effect, as evidenced by the differences in performance depending on which error representation the subject used first. This effect was a confounding factor in the results; a future study that conducted this experiment with a between-subjects design would eliminate this confounding factor.

## 8 Future Work

A study of refactoring errors in the wild could determine which categories are encountered with the highest frequency by programmers. The results would help to determine which categories are deserving of further research. A deeper empirical evaluation of which precondition violations are most commonly conflated by programmers would also help to drive future research.

In addition to helping define how our guidelines apply to other kinds of error messages, the precondition taxonomy may also help designers of refactoring tools. For example, while the *clash* category spans many refactorings, the guidelines tell us that a representation of any *clash* violation should show the relationship between the clashing program elements. Rather than a string describing the violation, a violation display method in the development environment could take as arguments both of the program elements, and display their relationship, regardless of which refactoring generated the violation. This might help amortize the development cost of implementing a graphical error system such as Refactoring Annotations.

Refactoring Annotations may also prove to be an opportunity to help the programmer not just to understand precondition violations, but also to resolve them. This is because the graphics provide convenient "hooks" for implementing further functionality for highly-focused, detailed programmer interaction. For example, consider again Figure 8. It may be useful to allow the programmer to interact with the two bottom arrows and messages to open those elements in the existing editor, a new editor, or an embedded editor, or to provide a "quick fix" that changes the visibility of `transferQueue`, so that it can be accessed in the destination class. In this way, graphical representations of precondition violations could provide not only a way to enhance understanding of the precondition violations, but also a method of resolving them.

## 9 Conclusions

In this article, we have discussed how toolsmiths can improve the user interface of refactoring tools to make it easier for programmers to understand why a tool refuses to perform a refactoring. We have presented a tool called Refactoring Annotations and distilled usability guidelines from it. The results of our two evaluations suggest that

Refactoring Annotations do indeed improve usability, in terms of speed, correct understanding, and programmer preference. Some violations of refactoring preconditions may continue to be difficult for programmers to understand, but we believe that this work is a beginning that will help to aid that understanding.

## 10 ACKNOWLEDGEMENTS

For their reviews and advice, we would like to thank Barry Anderson, Robert Bauer, Paul Berry, Iavor Diatchki, Tom Harke, Brian Huffman, Mark Jones, Jim Larson, Chuan-kai Lin, Ralph London, Philip Quitslund, Suresh Singh, Tim Sheard, and Aravind Subhash. Special thanks are due to participants in the user study and our anonymous reviewers for detailed, insightful criticism. We also thank the National Science Foundation for partially funding this research under grant CCF-0520346.

## REFERENCES

[1] T. R. Andersen. Extract method: Error message should indicate offending variables. Bug Report, 4 2005. https://bugs.eclipse.org/89942.

[2] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

[3] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.

[5] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12:369–388, 2002.

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

[7] W. G. Griswold. *Program restructuring as an aid to software maintenance*. PhD thesis, University of Washington, Seattle, WA, USA, 1992.

[8] T. D. Hendrix, J. H. C. II, S. Maghsoodloo, and M. L. McKinney. Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java. In *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 382–386, New York, NY, USA, 2000. ACM.

[9] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proc. Intl. Sym. on Applied Corporate Computing*, 1995.

[10] W. Joy and M. Horton. An introduction to display editing with vi. University of California, Berkeley, 1984.

[11] G. Kniesel and H. Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52:9–51, August 2004.

[12] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. In *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, pages 105–111, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

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

[14] E. Murphy-Hill. *Programmer-Friendly Refactoring Tools*. PhD thesis, Portland State University, 2009.

[15] E. Murphy-Hill and A. 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.

[16] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009.

[17] J. Nielsen. Ten usability heuristics. Internet, 2005. http://www.useit.com/papers/heuristic/heuristic_list.html.

[18] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[19] P. C. Rigby and S. Thompson. Study of novice programmers using Eclipse and Gild. In *Eclipse '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 105–109, New York, NY, USA, 2005. ACM.

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

[21] B. Shneiderman. System message design: Guidelines and experimental results. In A. Badre and B. Shneiderman, editors, *Directions in Human/Computer Interaction*, Human/Computer Interaction, chapter 3, pages 55–78. Ablex Publishing Corporation, 1982.

[22] Z. Xing and E. 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.

**Emerson Murphy-Hill** Emerson is an assistant professor at North Carolina State University. His research interests include human-computer interaction and software tools. He holds a Ph.D. in Computer Science from Portland State University. Contact him at emerson@csc.ncsu.edu; http://www.csc.ncsu.edu/faculty/emerson.

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