Alternate Refactoring Paths Reveal Usability Problems

Mohsen Vakilian and Ralph E. Johnson
University of Illinois at Urbana-Champaign, USA
{mvakili2, rjohnson}@illinois.edu

ABSTRACT

Modern Integrated Development Environments (IDEs) support many refactorings. Yet, programmers greatly underuse automated refactorings. Recent studies have applied traditional usability testing methodologies such as surveys, lab studies, and interviews to find the usability problems of refactoring tools. However, these methodologies can identify only certain kinds of usability problems. The *critical incident technique (CIT)* is a general methodology that uncovers usability problems by analyzing troubling user interactions. We adapt CIT to refactoring tools and show that *alternate refactoring paths* are indicators of the usability problems of refactoring tools. We define an alternate refactoring path as a sequence of user interactions that contains cancellations, reported messages, or repeated invocations of the refactoring tool. We evaluated our method on a large corpus of refactoring usage data, which we collected during a field study on 36 programmers over three months. This method revealed 15 usability problems, 13 of which were previously unknown. We reported these problems and proposed design improvements to Eclipse developers. The developers acknowledged all of the problems and have already fixed four of them. This result suggests that analyzing alternate paths is effective at discovering the usability problems of interactive program transformation (IPT) tools.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Design, Experimentation, Human Factors, Measurement

Keywords

Refactoring, usability, evaluation, critical incident, empirical

ICSE '14, May 31 – June 7, 2014, Hyderabad, India
`http://dx.doi.org/10.1145/2568225.2568282`

1. INTRODUCTION

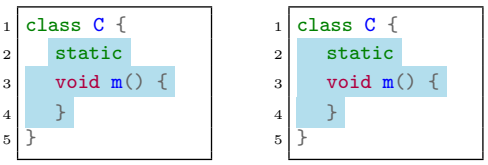
Refactoring is changing code without altering its observable behavior [25, 27, 38]. Major Integrated Development Environments (IDEs), including Eclipse, IntelliJ, NetBeans, Visual Studio, and Xcode, provide tool support to make software refactoring more efficient and reliable. Nonetheless, developers greatly underuse automated refactorings, mostly due to usability problems [35, 36, 42]. ISO [17] defines *usability* as "extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". Anything that damages the usability of a product is a *usability problem.*

Researchers have evaluated refactoring tools with usability testing methodologies such as lab studies and interviews [24, 26, 31, 33–35, 41, 42]. These methodologies find real usability problems by recruiting only a small number of participants. However, these methodologies are known to be suitable for identifying only certain usability problems, e.g., those that can be exposed during a short lab study or the ones that a programmer can remember during an interview. This is because such methodologies can evaluate the tool only for a short period of time, limited kinds of tasks, and a small number of participants.

Humans learn from their past mistakes. Certain events severely affect our lives, e.g., accidents and injuries. Reflecting on such events, we try to improve our future strategies. If refactoring tools were human beings, how would they have learned from their past experiences?

The *critical incident technique (CIT)* [23, 39] is a general methodology for revealing problems by analyzing *critical incidents.* A critical incident is a breakdown of a user's interaction with the system that seriously affects the user's task. The Human-Computer Interaction (HCI) community has found that CIT is an effective, complementary methodology for discovering usability problems [18, 28, 29]. However, the notion of "critical incidents" is not well-understood or studied for interactive program transformation (IPT) tools, e.g., refactoring tools. This leaves three research questions open: What are the critical incidents for IPT tools? Are these incidents indicators of the usability problems of IPT tools? How can evaluators infer usability problems from critical incidents?

Our work bridges the gap between two lines of research (program transformation and CIT) by adapting CIT to refactoring tools. We show that *alternate refactoring paths* are indicators of usability problems. An *alternate (refactoring) path* is a sequence of user interactions with the refactoring tool that differs from the *primary path.* The primary path,



(a) This selection results in an invocation of Move Static Members.

(b) This selection results in an invocation of Move Compilation Unit.

Figure 1: The Eclipse refactoring tool detects the type of the Move refactoring based on the code selection. The only difference between these selections is the two leading spaces on line 2 of Figure 1b. Although the code selections shown in this figure are very similar, they result in invocations of different refactorings: Move Static Members and Move Compilation Unit.

also known as the *happy path*, is an ideal sequence of interactions that leads to a successful application of the automated refactoring. Refactoring tool users take alternate paths for various reasons, e.g., making an invalid selection, invoking the wrong automated refactoring, or violating a precondition check.

We hypothesize that events such as *cancellations*, *repeated invocations*, and *reported messages* of a refactoring tool, which correspond to alternate paths, are likely to indicate usability problems. To test this hypothesis, we conducted a field study [42] with 36 programmers who used Eclipse for a total of 2,320 programming hours over the course of three months. By analyzing 145 alternate refactoring paths in this data set, we found 15 usability problems [1–15], 13 of which were previously unknown. We both reported the problems and proposed design improvements to the developers of Eclipse. Consequently, the developers acknowledged all the problems we reported and have already fixed four of them. These results suggest that analyzing alternate refactoring paths is effective at identifying the usability problems of refactoring tools.

The analysis of alternate refactoring paths revealed a variety of usability problems (Section 4). For example, Figure 1 shows how a minor change to a code selection results in the invocation of an unexpected Move refactoring. Reporting this problem [7] to the Eclipse developers led to the discovery of broader consequences of the problem (Section 4.5.2).

There are several advantages to our adaptation of CIT for finding the usability problems of IPT tools. First, our automatic data collection method can scale to many participants and collect data for a long time. This makes it possible to find usability problems from many user interaction paths. Second, our automatic data collectors are unobtrusive. This allows the evaluators to find usability problems without interfering with programmers' work. Third, the large set of collected data can be mined to measure the frequency of usability problems empirically. Finally, analyzing the alternate paths reveals not only usability problems, but also design improvements.

In summary, this work contributes to the field of refactoring in several ways:

- We adapt (Section 3) and evaluate (Section 4) CIT for finding the usability problems of refactoring tools. This adaptation can be used to find the usability problems of other refactoring or IPT tools.
- We find real usability problems (Section 4) of a refactoring tool and propose design improvements to address them. These problems are encountered by programmers in the field and confirmed by the developers of the refactoring tool.
- We provide empirical evidence for the frequency of the usability problems revealed by our method.

2. RELATED WORK

We discuss two main lines of research related to our work.

2.1 Usability Studies on Refactoring Tools

Any useful software continuously evolves. Refactoring tools aim to reduce the cost and risk of evolving software. Nonetheless, recent studies show that usability problems deter programmers from using automated refactorings [35,42]. These studies have uncovered some of the usability problems of refactoring tools by a combination of quantitative and qualitative data analyses. Our work differs from these studies in two ways. First, we derive a systematic usability evaluation method for refactoring tools. Second, our evaluation method relies mostly on automatically collected usage data rather than qualitative data.

Researchers have proposed alternative designs to improve the usability of refactoring tools [24, 26, 31–34, 41]. These proposals seek to address specific usability problems such as developers' unawareness of automated refactorings [34], poor methods of invoking automated refactorings [24, 26, 31, 32], and low predictability of automated refactorings [41]. While this line of work has generated promising ideas for improving the usability of refactoring tools, they lack a general mechanism for finding usability problems encountered by programmers in the field.

2.2 The Critical Incident Technique

2.2.1 The Origins of CIT

The critical incident technique (CIT) is a general technique developed in its current form by Flanagan and published in *Psychological Bulletin* in 1954 [23]. The technique is believed to have been founded even earlier by Galton (circa 1930).

Flanagan defined CIT as a set of procedures for collecting and analyzing human behaviors that have critical significance (positive or negative). Typically, the respondents are asked to describe their significant experiences. Variations of this method have been widely used in Human Factors [39]. Despite all the variations of CIT, a common definition of the critical incident still holds. A critical incident is an event during a task that is a significant indicator of some aspect of the objective of the study.

Flanagan describes CIT as an outgrowth of the studies conducted as part of the Aviation Psychology Program of the United States Army Air Forces in World War II. One of the early studies in this program that employed CIT was analyzing the reasons of disorientation while flying. In this study, the pilots were asked to think of occasions during their flight that they experienced disorientation, i.e., they

felt uncertain about their spatial position. Then, they were asked to describe what they saw, heard, or felt that caused that experience. This study resulted in a number of recommendations for changing the cockpit design and training pilots to avoid disorientation.

2.2.2 CIT in Human-Computer Interaction

del Galdo *et al.* adapted CIT to HCI in 1986 [22]. They conducted a study to evaluate the documentation of a conferencing system. The participants were asked to perform a task using the system. They were also asked to report any incident (success or failure) that they encountered while using the documentation. The experimenters observed the participants during the study. This variation of CIT in which the participants report the incidents as they are encountered is called the *user-reported critical incident technique (UCIT)*. del Galdo *et al.* made recommendations for improving the documentation based on the reported incidents.

Hartson *et al.* adapted UCIT to *remote usability evaluation*, where the users and evaluators are in different physical locations [28, 29]. In this variant of CIT, the experimenters train the users to identify and report critical incidents. The users report the critical incidents as they are encountered during the task. Later, the evaluators analyze the reports and the accompanied contextual information (e.g., video recordings) of the incidents. Hartson *et al.* showed the effectiveness of CIT through several studies on web applications. They also reported a problem with UCIT—users tend to delay reporting the incidents. While Hartson *et al.* used CIT to enable remote usability evaluation, their studies were mostly in the lab.

Akers *et al.* devised a variant of CIT for evaluating the usability of applications like Google SketchUp and Adobe Photoshop [18]. They conducted a lab study, in which they instructed the participants to perform predefined tasks. The system automatically recorded operations such as undo and erase along with screen capture video. To obtain more contextual information about these events, the participants were paired up to discuss their captured video episodes centered around the recorded events. They found that the participants sometimes failed to report problems because they forgot or blamed themselves rather than the application. An interesting difference in the results of our studies is that we did not find undo a good indicator of the usability problems of refactoring tools (Section 5).

Yoon *et al.* studied developers' backtracking strategies (e.g., removing inserted code or restoring removed code) [44]. This study can be viewed as an application of CIT with implications for better tool support for backtracking strategies [45].

Studies have confirmed the effectiveness of remote usability evaluation in different settings [19, 20, 30]. Nevertheless, practitioners do not practice it much, although when they do, they appreciate its value [21].

While similar to this line of research we propose a variant of CIT, there are several major differences. First, we focus on IPT tools, for which the notion of critical incident is not well-understood. Second, we seek to avoid interference with programmers' workflow. Thus, we capture more detailed information automatically to reduce the involvement of the users in identifying the usability problems. Finally, we evaluate our method by analyzing the authentic data collected from programmers performing real tasks in their normal working environments.

Table 1: Number of years of programming experience of the participants as reported by the 34 who completed the demographic survey.

Years	1–2	2–5	5–10	> 10
Participants	2	6	17	9

3. METHODOLOGY

The critical incident technique (CIT) consists of two major phases: data collection and data analysis. The rest of this section describes how we adapted each of these phases for identifying the usability problems of IPT tools.

3.1 Data Collection

Evaluators can collect the critical incidents through surveys, interviews, observing the participants, or asking the participants to report the incidents during the task. These data collection techniques are not scalable to many users, are based on artificial tasks, or interfere with users' work. So, we made our data collection automatic to collect a *large* set of data that covers many usage scenarios of the refactoring tool in a form that is amenable to *automatic data analysis*. We made the data collection unobtrusive to avoid altering programmers' behavior. Finally, instead of collecting the data from predefined tasks performed at the lab, we decided to collect the data from real tasks that are more representative of how the refactoring tool is used in practice.

We conducted a field study on 36 programmers for three months, collecting usage data for a total of 2,320 programming hours.

3.1.1 Participants

We recruited participants from the industry ($N = 17$) and academia ($N = 19$). We advertised the study to the open-source community via mailing lists, IRC, and e-mail. In addition, we invited researchers from six research labs at the computer science department of the University of Illinois at Urbana-Champaign. We did not remunerate the participants. Instead, we explained to them the potential contributions of our study. We asked the participants to fill out a demographic survey. Based on the survey results, 26 participants had at least five years of programming experience (Table 1). Moreover, the participants indicated that they worked on a variety of domains such as banking, database management systems, business process management, and marketing.

3.1.2 Automatic Data Collector

We developed CodingSpectator [43], an unobtrusive tool for collecting the usage data of the Eclipse refactoring tool. The only interaction that the participants had with CodingSpectator was to install it like any Eclipse plug-in, and enter their username and password when prompted to submit their data to our central repository. We chose to make the data collection process unobtrusive to study software evolution practices in the wild [37, 42].

CodingSpectator captures more data about the usage of the refactoring tool than what Eclipse already does. It

captures invocations of 23 automated refactorings of the 33 supported by Eclipse. The Eclipse refactoring history captures the following information about only the automated refactorings that are *performed*:

timestamp The invocation time of the automated refactoring

kind The kind of the automated refactoring (e.g., Rename Local Variable, Extract Method)

selection offsets The start and length of the code selection used to invoke the refactoring

configuration options Information about how the programmer configured the automated refactoring (e.g., the name and accessibility of the new method created by Extract Method)

Eclipse records data only about the *primary* paths that users take while using the refactoring tool. In other words, it only records the eventually performed automated refactorings. However, troubling interactions may make the user take *alternate* refactoring paths by *canceling* or *undoing* a refactoring that has reported a *message* or is hard to use.

Since our goal was to adapt and evaluate CIT for refactoring tools, we made CodingSpectator augment the Eclipse refactoring history in two ways. First, CodingSpectator records *canceled*, *undone*, and *redone* automated refactorings in addition to the *performed* ones along with any *messages* reported by the refactoring tool. Second, it captures contextual information about refactoring activities that an evaluator can use to derive usability problems. The contextual information consists of the following:

navigation history A description of how and when the user navigated through the refactoring wizard

invocation method Whether the user has invoked the refactoring tool through the wizard

selection The piece of code selected to invoke the refactoring, and a larger slice of code surrounding the selection[1]

messages All problems that the refactoring tool has reported to the programmer (Section 3.1.3)

Since Eclipse does not provide a reusable API to capture the above information, we instrumented the Eclipse source code. During the installation process, CodingSpectator replaces several existing Eclipse plug-ins, such as LTK and JDT, by our instrumented versions of these plug-ins.

3.1.3 Refactoring Messages

A *refactoring precondition* is a property that the refactoring tool checks at various stages, e.g., selection, invocation, configuration, and commit, to guarantee that the change will preserve the behavior of the program. If a precondition fails, the refactoring reports a message whose type depends on the severity of the problem and the stage of refactoring. We refer to such a message as a *refactoring message* or just a *message* in this paper.

The Eclipse refactoring tool may report any of about 640 messages of four types to its user [16]:

[1]We made CodingSpectator capture this information because the selection offsets captured by Eclipse do not always reflect exactly the ones used by the programmer due to some normalization that Eclipse applies on the selections.

Unavailable The refactoring tool refuses to open the refactoring wizard due to the failure of a precondition.

Warning This kind of message attempts to predict compilation problems and can often be ignored safely.

Error The Eclipse documentation recommends not to continue an automated refactoring that has reported a message of this kind, because it is very likely to break the code.

Fatal Error The refactoring tool refuses to perform the change on the code.

The refactoring tool may report an Unavailable message only before the refactoring wizard is open, while it may report the other types of messages only during the later stages of refactoring.

3.2 Data Analysis

The data analysis phase of CIT consists of two major phases: identifying critical incidents and inferring usability problems. We automated the identification of critical incidents and their accompanying contextual information. However, inferring the usability problems from critical incidents is a manual process that an evaluator can do. The rest of this section describes how we adapted these two phases of CIT to IPT tools.

3.2.1 Identifying Critical Incidents

Collecting the critical incidents. We first collected a set I of critical incidents, i.e., refactoring invocations that reported a message (Section 3.1.3) as well as canceled, undone, and redone refactorings. Then, we add to I any refactoring that occurred within five minutes of a critical incident in I.

Finding the most frequent refactoring messages. We computed the frequency of each refactoring message by counting the number of times it occurred in I. We consider the frequency of a refactoring message a measure of the criticality of the message as an incident. We have released the data about the frequencies of the refactoring messages at `http://hdl.handle.net/2142/47414`.

The number of all messages that the Eclipse refactoring tool may report is large ($N = 640$). It is tedious to investigate all possible messages for usability problems. Besides, it is often impossible to infer a usability problem from a message without having other contextual information about the refactoring invocation. CodingSpectator's data indicates that the Eclipse refactoring tool reported only 83 different kinds of messages to the participants during the study. Therefore, we focus on these messages that the refactoring tool reported in practice. In addition, we analyze the most frequent messages in the contexts that they appeared. The contextual information captured by CodingSpectator allows us to identify the conditions under which the message is reported and how the programmers react to the message.

Extracting refactoring batches. The *refactoring batch* of a critical incident is a subset of I containing the events that are semantically related to the critical incident. Examples of semantically related events are cancellations and invocations of the same refactoring or refactorings on related program entities. A refactoring batch provides the context necessary for inferring usability problems from a critical incident. We

extracted the refactoring batches of the most frequent refactoring messages. To extract the refactoring batch of a critical incident, we manually inspected the events in I that occurred within 20 minutes of the critical incident and extracted those that were semantically related to the critical incident. We found that a window of 20 minutes was large enough to cover all events that were semantically related to a critical incident. For each event in the refactoring batch of a critical incident, we took a note of how the event was related to the critical incident. We referred to these notes while inferring usability problems from refactoring batches.

3.2.2 Inferring Usability Problems

Analyzing the refactoring batches. We manually inspected 145 refactoring batches of the most frequent refactoring messages to infer usability problems. We used the contextual information in each refactoring batch (e.g., code snippet, selection, and invocation method) to reproduce the behavior of the refactoring tool. For each refactoring batch, we examined programmers' reactions to the reported messages. If programmers dismissed the message, we checked if the message was poor, e.g., vague, difficult to understand, or uninformative. We evaluated the effectiveness of the message ourselves based on general usability guidelines, and we did not contact the participants. If the programmer canceled the refactoring, we checked if the refactoring was later repeated. If the refactoring was repeated, we checked for the possible changes to the configurations of the refactoring or any manual changes made to the code since the previous invocation. Such changes often revealed why the refactoring failed in the first attempt and how the refactoring tool could be improved to avoid the failure. If the refactoring was repeated multiple times unsuccessfully, we considered it a stronger indication of a usability problem.

Confirming the usability problems. Finally, we reported the usability problems that we inferred to Eclipse developers to confirm that they are indeed considered usability problems from the Eclipse developers' points of view. We included in our reports some empirical data about each usability problem, e.g., the number of refactoring batches with the same usability problem, the number of times programmers canceled or performed a refactoring as well as a summary of the strategies that the programmers employed to remedy the usability problem. We also made our reports actionable by making concrete suggestions on how to resolve the usability problems.

4. USABILITY PROBLEMS

We were able to infer usability problems by analyzing alternate refactoring paths. This shows that alternate refactoring paths are indicators of usability problems. This section presents some of the usability problems that we identified. For each usability problem, we present its frequency in our data set, how we identified the usability problem, and what suggestion we made to Eclipse developers to resolve the usability problem.

CodingSpectator recorded detailed information for 92% (4,245 of 4,611) of the refactorings that were performed and recorded by Eclipse. Table 2 illustrates the frequencies of automated refactoring events. Table 3 introduces several symbols that we use in the rest of the paper. Table 4 lists

Table 2: The frequency of each kind of automated refactoring and refactoring message in the CodingSpectator data set.

Event	Performed	Canceled	Undone	Redone	Warning	Error	Fatal Error	Unavailable
Occ.	4,245	267	284	34	191	124	49	85

Table 3: Meanings of the symbols used to describe an alternate refactoring path quantitatively. *Occ.* can be greater than *Per.* + *Can.*, because a refactoring with an Unavailable message is not counted as either performed or canceled.

Symbol	Meaning
Occ.	The number of occurrences of the alternate path.
Par.	The number of participants affected by the alternate path
Bat.	The number of refactoring batches containing the alternate path
Per.	The number of instances of the alternate path that the participants performed
Can.	The number of instances of the alternate path that the participants canceled
Rank	The index of a message in an array of all messages sorted by *Occ.* descendingly

the most frequent messages that the Eclipse refactoring tool reported to our participants.

Due to privacy and confidentiality constraints of the field study, we cannot present the participants' code. Instead, we demonstrate the usability problems using simplified versions of the participants' pieces of code.

4.1 Vague Messages

We analyzed the refactoring batches of the two most frequent messages of the Eclipse refactoring tool (rank one and two in Table 4). When we reproduced these messages, we noticed that they are vague. That is, they do not clearly explain the problem, leaving the programmer confused about the risks of performing the refactoring and the actions required to mitigate the risks. For example, for the most frequent message (rank one in Table 4), neither the message nor any other information on the refactoring wizard indicates what part of the code modification may not be accurate. This Warning is too broad to be of any actionable use. Perhaps this is why the participants continued 96% (108 of 113) of the refactorings despite reporting this Warning. We reported several instances of this usability problem to the Eclipse developers [1, 11–13].

4.2 Overly Strong Preconditions

Precondition checking of refactoring tools is a delicate process. On one hand, the preconditions should be strong enough to prevent the refactoring from breaking the code or altering its behavior in unintended ways. On the other hand, the preconditions should not be overly strong and reject safe refactorings. Our prior study showed that programmers prefer flexible refactorings, often ignore precondition failures,

Table 4: A list of the most frequent messages reported by the Eclipse refactoring tool. Column "Refactorings" lists the refactorings that reported each message. Refer to Table 3 for meanings of the other column headers.

Rank	Message	Type	Refactorings	*Occ.*	*Par.*	*Per.*	*Cam.*
-	-	-	All	4200	29	4032	168
1	Code modification may not be accurate as affected resource ∗ has compile errors	Warning	Rename Compilation Unit, Type, Enum Constant, Field, Method, Package	113	17	108	5
2	Found potential matches. Please review changes on the preview page.	Error	Change Method Signature, Move	44	14	35	9
3	Type ∗ contains a main method—some applications (such as scripts) may not work after refactoring.	Warning	Rename Compilation Unit, Type, Package	41	7	39	2
4	This refactoring cannot be performed correctly due to syntax errors in the compilation unit. To perform this operation you will need to fix the errors.	Fatal Error	Rename Field, Method	22	10	0	22
5	Ambiguous return value: Selected block contains more than one assignment to local variables.	Unavailable	Extract Method	22	6	0	0
6	This name is discouraged. According to convention, names of local variables should start with a lowercase letter.	Warning	Change Method Signature, Extract Local Variable, Rename Local Variable	15	5	12	3
7	Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed	Error	Extract Method	15	6	8	7

and manually fix the problems afterwards [42]. The following discusses two usability problems [2, 10] related to overly strong preconditions that we identified.

Realizing the disadvantages of overly strong preconditions, others have proposed a bounded-exhaustive testing approach [40]. Our approach of finding overly strong preconditions by analyzing critical incidents complements theirs in two ways. First, we find overly strong preconditions that programmers have encountered in real-world, while their approach finds overly strong preconditions in a large number of automatically generated programs. Second, their approach does not currently support refactorings below the method level, e.g., Extract Method.

4.2.1 Ambiguous Return Value

The Extract Method refactoring reported the Unavailable message in Figure 2 to our participants (*Occ.* = 22, *Par.* = 6).

We first reproduced the message using the information captured in its refactoring batches to understand the conditions under which the refactoring tool reports this message. These batches contained an invocation of Extract Method with a code selection similar to the one shown in Figure 3. By studying the message itself and reproducing the repeated refactorings in the refactoring batches of the message, we found why the refactoring tool refuses to continue: the selected code assigns to two local variables (`a` and `b`) that are used outside the selection (line 6). Since a Java method can return at most one value, the refactoring tool cannot infer the return value of the extracted method (`a` or `b`). We identified two usability problems by analyzing the refactoring batches of this Unavailable message.

First, while trying to reproduce the message, we found that the message is not descriptive enough. The message indicates that the refactoring tool is unable to proceed because the selected piece of code assigns two local variables. However, this is not a sufficient condition. The refactoring tool refuses to continue only when more than one local variables are both assigned in the selected piece of code and used outside the

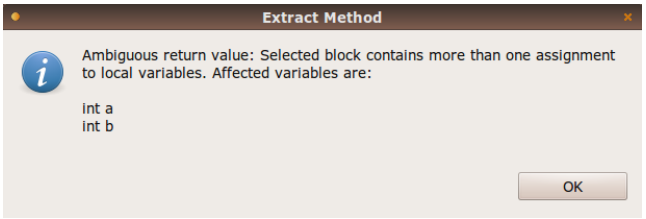


Figure 2: The Eclipse Extract Method refactoring reports the message "Ambiguous return value" for the code selection shown in Figure 3.

```
1 class C {
2   int m() {
3     int a, b;
4     a = 1;
5     b = 2;
6     return a + b;
7   }
8 }
```

Figure 3: The Eclipse Extract Method refactoring reports the message "Ambiguous return value" (Figure 2) for the code selection shown in this figure.

selection. We suggested that the Eclipse developers make the message more descriptive. The Eclipse developers fixed this problem [1].

Second, this precondition check of Eclipse is overly strong. Refusing to continue the refactoring caused additional overhead to our participants as shown by their efforts to repeat the refactoring in the refactoring batches. We examined the cancellations and repeated invocations in refactoring batches to understand how programmers handled this message. We found that programmers used three solutions. They either narrowed the selection, widened it, or converted the local variables to fields and repeated the refactoring. Based on

this observation, we suggest that the refactoring tool be more flexible and let the programmer continue the refactoring in the following ways. One option is to warn the programmers about the use of the local variables outside the selection but still allow them to continue the refactoring and make the new method have no return value. Alternatively, the tool could suggest that it automatically converts the local variables to fields and proceed. We reported this overly strong precondition to Eclipse developers. They acknowledged the value of flexibility saying [2]:

> I have had instances when I have had to perform a refactoring manually because Eclipse would not proceed because of an error. On such occasions I do wish for things to be a bit more flexible.

4.2.2 *Missing Return*

The ERROR message with rank seven in Table 4 was the second most frequent message of Extract Method. Seven of the 11 refactoring batches that contained this message contained at least one cancellation and one repeated invocation. In one refactoring batch, the participant invoked the refactoring four times each time changing a configuration option, selection, or code. These alternate paths indicate the difficulty of using the tool and the possibility of usability problems.

By reproducing the message, we found the underlying reason of the message. Eclipse cannot extract a set of statements that has some control flow paths ending in `return` and some others not (Figure 4a). We examined the code selections in the refactoring batches of this message to see if this precondition check can be relaxed for certain common code selections. As a result, we found that the refactoring could have been more flexible and infer the missing `return` statements (Figure 4b) in seven of the 11 batches that contained this message. The IntelliJ refactoring can infer the missing `return` statement in some cases. We suggested this enhancement to Eclipse developers and they acknowledged its benefits [10].

4.3 Name Conflicts

Refactorings such as Rename and Extract Local Variable, which change the name of an existing program element or introduce a new named program element, may cause name conflicts. Table 5 lists the most frequent refactoring messages that Eclipse reported to the participants due to name conflicts (*Occ.* = 43, *Par.* = 14, *Bat.* = 36, *Per.* = 20, *Can.* = 23).

Our analysis of the refactoring batches showed that the participants either continued the refactoring despite the message and resolved the name conflict manually or canceled the refactoring and invoked it again to enter a different name. Having the programmers navigate back to the configuration page or repeat the invocation of the refactoring with a different name to refactor safely is an additional overhead, which we consider a usability problem.We reported this source of additional overhead to Eclipse developers and suggested that the refactoring tool informs programmers about potential name conflicts earlier [9, 15].

4.4 Unintuitive Configuration Options

The goal of the Move Instance Method refactoring is to move the declaration of an instance method from its enclosing class to another class. Figure 5 shows the effect of applying Move Instance Method on an example piece of code.

```
1  class C {
2    boolean b;
3    boolean m() {
4      if (b)
5        return true;
6      else
7        System.out.
8          println(
9            "else");
10     return false;
11   }
12 }
```

(a) The selection contains a `return` statement but not all possible execution flows end in a `return`.

```
1  class C {
2    boolean b;
3    boolean m() {
4      if (n())
5        return true;
6      return false;
7    }
8    boolean n() {
9      if (b)
10       return true;
11     else
12       System.out.
13         println(
14           "else");
15     return false;
16   }
17 }
```

(b) It is possible to automatically infer the missing `return` statement in some cases.

Figure 4: The Extract Method refactoring of Eclipse results in a compilation problem if it reports the Error message: "Selected statements contain a return statement but not all possible execution flows end in a return. . . .".

Six of the participants invoked the Move Instance Method refactoring for a total of 16 times as parts of ten batches. However, none of the invocations were applied. Either the programmer canceled the refactoring (11 times), or the refactoring tool refused to continue and reported an UNAVAILABLE message (5 times). Two refactoring batches indicated that the participants invoked the refactoring tool three times but did not succeed to perform the Move Instance Method refactoring. These critical incidents led us to identify two usability problems.

First, the configuration dialog provides options that programmers cannot easily interpret. The dialog asks for a pair of "name" and "type" (Figure 6), while the programmer would like to select the destination class. A refactoring batch indicated that a participant spent 27 seconds on the configuration dialog of Move Instance Method, which is higher than the average time our participants spent on this dialog (16.5 seconds). We asked the participant why he spent this time on the configuration dialog and eventually canceled the refactoring. The participant said that he expected the refactoring tool to ask him about the destination class not a pair of "name" and "type". He canceled the refactoring because he could not interpret the required options. However, selecting a destination class is not sufficient in general, because the refactoring tool has to update the call sites as well. The refactoring asks the programmer to select a variable to determine the new receivers of the call sites. For the example shown in Figure 5, the refactoring tool changes the call `c.m(e1, e2)` (line 12, Figure 5a) to `e1.m(c, e2)` (line 8, Figure 5b). Nonetheless, since the configuration dialog does not communicate the necessity of these options well, the programmer gets confused.

Table 5: The most frequent refactoring messages that were due to name conflicts. Column "Refactorings" lists the refactorings that reported each message. Refer to Table 3 for meanings of the other column headers.

Rank	Message	Type	Refactorings	*Occ.*	*Par.*	*Per.*	*Can.*
8	Duplicate local variable	ERROR	Extract Local Variable, Rename Local Variable	14	8	5	9
14	A variable with name ∗ is already defined in the visible scope	WARNING	Extract Local Variable	8	2	6	2
17	Package ∗ already exists in this project in folder	WARNING	Rename Package	6	3	6	0
18	Type named ∗ already exists in package	ERROR	Rename Compilation Unit, Type	6	5	0	6
22	Compilation unit ∗ already exists	FATAL ERROR	Rename Compilation Unit, Type	5	4	0	5

```
1  class C {
2    D d1;
3    D d2;
4    void m(E e1,
5           E e2) {
6      d1.m();
7    }
8    void m2() {
9      E e1= new E();
10     E e2= new E();
11     C c= new C();
12     c.m(e1, e2);
13   }
14 }
15
16 class D {
17   void m() {
18   }
19 }
20
21 class E {
22 }
```

(a) Original code

```
1  class C {
2    D d1;
3    D d2;
4    void m2() {
5      E e1= new E();
6      E e2= new E();
7      C c= new C();
8      e1.m(c, e2);
9    }
10 }
11
12 class D {
13   void m() {
14   }
15 }
16
17 class E {
18   void m(C c,
19          E e2) {
20     c.d1.m();
21   }
22 }
```

(b) Refactored code

Figure 5: The programmer selects instance method `m` (line 4) to move it from class `C` to `E`.

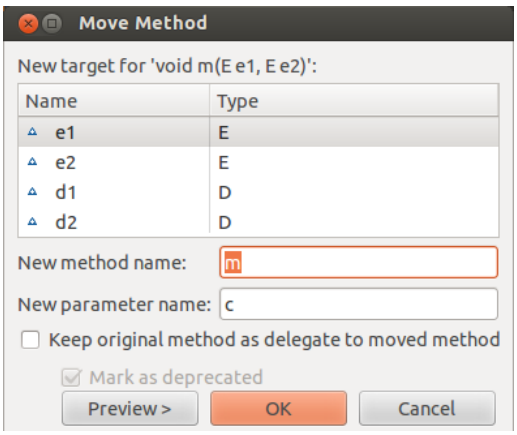


Figure 6: The configuration options of the Move Instance Method refactoring for the code example shown in Figure 5. The refactoring requires the new "target" of the method as a pair of "name" and "type". However, this requirement is neither easy to interpret by programmers nor necessary.

Second, the configuration dialog requires more options than what is necessary for moving certain kinds of methods. We found that fewer, simpler options were sufficient to support a common class of attempted refactorings captured in refactoring batches. We say an instance method is *effectively* `static` if it can be made `static` without introducing any compilation problems. For example, in Figure 5, method `C.m2()` (lines 8–13, Figure 5a) is effectively `static`. However, method `C.m(E, E)` (lines 4–7, Figure 5a) is not because it depends on the instance field `C.d1`. In five batches, the participants tried to move effectively `static` methods. If the method is effectively `static`, it would be sufficient for the dialog to ask the destination class from the programmer. However, the Eclipse refactoring tool always requires the programmer to select a field or parameter as the new target of the method. This design is restrictive because it does not allow the programmer to move an effectively `static` method to a class other than those reachable from a field or parameter.

We reported both of the above usability problems to Eclipse developers [3, 4]. The developers acknowledged these problems and made an improvement accordingly. To address the first usability problem, we suggested that the configuration dialog asks the destination class and new receiver separately and clarify why it requires a new receiver. We provided the developers with prototypes of alternative designs of the configuration dialog. To resolve the second usability problem, we suggested that the refactoring tool automatically detects effectively `static` methods and allows the programmer to move them to any writable class without requiring a new receiver.

4.5 Invalid Code Selections

Programmers have to select pieces of code to invoke most automated refactorings. However, selecting a valid piece of code can be error-prone, especially when the selection is long. Table 6 lists the most frequent refactoring messages reported because of invalid code selections. In total, 30 refactoring messages were reported due to invalid code selections. We identified several usability problems [5–8, 14] related to invalid selections.

4.5.1 *Trailing Semicolon*

The inclusion or exclusion of a semicolon can make a selection invalid for Extract Method. Figure 7 illustrates a selection that causes Extract Method to report the UNAVAILABLE message with rank 23 in Table 6. The Extract Method refactoring expects the trailing semicolon to be included in

Table 6: The most frequent refactoring messages due to invalid code selections. Column "Refactorings" lists the refactorings that reported each message. Refer to Table 3 for meanings of the other column headers.

Rank	Message	Type	Refactorings	*Occ.*	*Par.*	*Per.*	*Can.*
19	The end of the selection contains characters that do not belong to a statement	UNAVAILABLE	Extract Method	5	2	0	0
21	An expression must be selected to activate this refactoring.	UNAVAILABLE	Extract Local Variable, Constant	5	4	0	0
23	Cannot extract the left-hand side of an assignment.	UNAVAILABLE	Extract Method	5	2	0	0

```
1 class C {
2   void m() {
3     int i = 0;
4     i = 10 ;
5   }
6 }
```

Figure 7: Excluding a semicolon can make a selection invalid for Extract Method.

the selection because it expects a set of statements not expressions as its input. The participants that received this message (*Occ.* = 5, *Par.* = 2, *Bat.* = 3) eventually extended the selection to include the trailing semicolon and performed the refactoring successfully. Nonetheless, in two batches, the participants repeated the refactoring with the incorrect selection until they noticed the trailing semicolon. These repeated invocations indicate the subtlety of this message. We examined the refactoring batches of this message to see if the refactoring tool could be improved to avoid the alternate refactoring paths. We found that if the refactoring tool had automatically expanded such selections to include the trailing semicolon, the cancellations and repeated invocations of all refactoring batches of the message would have been avoided. We reported a usability problem to Eclipse developers [5] suggesting that the refactoring tool automatically expands such selections. Eclipse developers fixed this problem.

By analyzing alternate refactoring paths, we found a similar problem caused by including a trailing semicolon in the selection. This usability problem was already reported and fixed by the Eclipse developers [6].

4.5.2 *Equivalent Selections*

We inferred a usability problem of the Move refactoring by analyzing the refactoring batches containing the following FATAL ERROR message (*Occ.* = 4, *Par.* = 3, *Bat.* = 4, *Per.* = 0, *Can.* = 4): `A file or folder cannot be moved to its own parent.`

The Move Compilation Unit refactoring reports the above message when a compilation unit is about to be moved to its enclosing package—the default destination.

The configuration options that CODINGSPECTATOR captured for these refactorings indicated that in three of the batches the participants selected fields or methods and invoked the Move refactoring but the tool incorrectly interpreted the refactoring as Move Compilation Unit. We reproduced the attempted refactorings on pieces of code similar to the ones recorded by CODINGSPECTATOR. Surprisingly, we found that slight changes to the code selection result in the invocation of an unexpected kind of Move refactoring.

Figure 1 shows two very similar code selections that are interpreted differently by the Move refactoring. The selection in Figure 1a results in an invocation of Move Static Members, while the selection in Figure 1b results in an invocation of Move Compilation Unit. This is because the former selection tightly covers a method, while the latter covers slightly more than a method (leading spaces on line 2, Figure 1b). The refactoring tool interprets a selection that covers more than a method as a selection of the enclosing compilation unit.

This high sensitivity to code selections is a usability problem. Reviewing our report of this usability problem [7] led the Eclipse developers to discover more problems. The root cause of the problem is that the Move refactoring only considers the start offset of the selection. This observation uncovered another problem: if a programmer selects two methods, the Move refactoring will ignore the second one.

5. LESSONS LEARNED

Although we applied CIT to find the usability problems of an IPT tool, we learned some lessons along the way that are generalizable to other programming tools.

Reported messages. We found that the reported refactoring messages were more likely to indicate usability problems than other events. Nevertheless, we identified at least one usability problem by just examining canceled refactorings. Repeated invocations of an automated refactoring helped us infer how the participant overcame a usability problem. Unlike an adaptation of CIT to applications like Adobe Photoshop [18], we did not find any usability problems by just studying undone and redone refactorings. Nonetheless, undone and redone refactorings in a refactoring batch provided stronger evidence for the usability problems revealed by other events, e.g., reported messages. There are several possible explanations for why undo is an indicator of usability problems for an application like Adobe Photoshop but not a refactoring tool. First, the role of the undo operation may be different in the two applications. Programmers seem to use undo as a natural means of exploring a solution space. Second, inferring usability problems from an undone automated refactoring may require more contextual information that CODINGSPECTATOR does not capture.

Reproducibility. We found that reproducing the critical incidents was required in most cases to infer the usability problems. This suggests that the reproducibility of critical incidents is an important criteria in adapting CIT to IPT tools and other domains. The data collector has to capture enough details about the incidents so that the evaluator can later reproduce the critical incidents and examine them.

User reports. We contacted a participant to infer one usability problem (Section 4.4). We were able to refresh

the participant's memory by showing him CODINGSPECTATOR's data of his refactoring. The lesson here is that it is sometimes necessary to get the participant's report of the critical incident, e.g., what they were trying to achieve, what prevented them from achieving their goal, and how they overcame the problem. When employing CIT in a remote usability evaluation, there are two general ways of getting the participants' reports of the critical incidents: either ask the participants to report the incidents as they occur during the task, or prompt them by presenting enough data about their actions not too long after performing the task.

Design improvements. Although our goal was to infer usability problems from alternate paths, we learned that alternate paths can also suggest design improvements. We analyzed the repeated refactorings to see how the programmers overcame a usability problem manually. For some usability problems (Sections 4.2.1 and 4.5.1), we suggested design improvements that automated these manual strategies.

6. LIMITATIONS

Although we successfully identified usability problems, we do not have an estimate of what fraction of the usability problems we identified. Future work may answer this question by comparing the results of our method by conventional usability evaluation methods. Eclipse developers acknowledged all of the usability problems that we reported. Nonetheless, developers' judgments may not be perfect.

Although the contextual information that CODINGSPECTATOR collects allows us to identify usability problems, our identification process was mostly manual and tedious. Future research could explore automated techniques to reduce the burden on the evaluator. We used heuristics such as the number of occurrences of the events and affected participants to prioritize our evaluation efforts. Another possibility is to ask the users report the critical incidents that they encounter. The study of the pros and cons of such an approach is left to future work.

We adapted and evaluated CIT for the Eclipse refactoring tool. Since we were able to extend some of the results to other refactoring tools, we expect our results to generalize. Nonetheless, we did not thoroughly investigate the generalizability of our method to other IPT tools.

CODINGSPECTATOR captures snippets of code close to the program elements under refactoring. We found that this information was crucial for deriving usability problems. Nonetheless, recording such sensitive pieces of information raises privacy and confidentiality issues. We faced difficulty in recruiting participants because of these issues. There are several challenges in scaling our method to a large number of programmers. One challenge is to make the data collection transparent. Another challenge is to design an incentive mechanism for programmers to share their data.

7. FUTURE WORK

This work opens up future research in several directions. One direction is to adapt other variants of CIT, e.g. UCIT, to IPT tools. This would provide insight about the quantity and quality of user reports and their effect on the number and severity of inferred usability problems.

Another future line of research is to extend our method to other IPT tools, e.g., code generators, bug fixers, and other refactoring tools.

Finally, our vision is that programming environments adopt data collection frameworks like CODINGSPECTATOR to make remote asynchronous usability evaluation possible at a large scale. This large scale will raise new research challenges such as privacy assurance and automatic clustering of similar critical incidents.

8. CONCLUSIONS

Interactive program transformation (IPT) tools, such as refactoring tools, aim to make the evolution of software more economical and reliable. Despite the automation of many recurring or sophisticated changes, refactoring tools are heavily underused [35, 36, 42]. Recent studies suggest that usability problems are major obstacles to a widespread use of refactoring tools [35, 42]. We advocate continuous collection of usage data to analyze the interactions of programmers with a refactoring tool. This technique is used in other application domains, e.g., web applications. The challenge is to find usability problems from a large corpus of usage data, which is like finding a needle in a haystack.

We adapted the critical incident technique (CIT) to refactoring tools. We examined *alternate refactoring paths* to find the usability problems of refactoring tools. Alternate refactoring paths are paths of user interactions that differ from the *primary* or *happy* path of using an automated refactoring. An alternate refactoring path contains events such as cancellations, repeated invocations, and error messages. We mined alternate refactoring paths in a large, real-world refactoring usage data set and analyzed a subset of it to identify usability problems. As a result, we found 15 usability problems, all of which have been acknowledged by the Eclipse developers and four have already been fixed. This result shows that alternate refactoring paths reveal usability problems.

9. ACKNOWLEDGMENTS

We thank David Akers, Brian Bailey, Nicholas Chen, Milos Gligoric, Hamid Jahani, Stas Negara, Cosmin Radoi, Samira Tasharofi, and Roshanak Zilouchian Moghaddam for their valuable feedback on a draft of this paper. This work is partially supported by NSF CCF 11-17960 and DOE DE-FC02-06ER25752.

10. REFERENCES

[1] https://bugs.eclipse.org/365664.
[2] https://bugs.eclipse.org/365129.
[3] https://bugs.eclipse.org/364947.
[4] https://bugs.eclipse.org/365286.
[5] https://bugs.eclipse.org/366281.
[6] https://bugs.eclipse.org/324237.
[7] https://bugs.eclipse.org/401032.
[8] https://bugs.eclipse.org/366280.
[9] https://bugs.eclipse.org/401262.
[10] https://bugs.eclipse.org/365820.
[11] https://bugs.eclipse.org/363257.
[12] https://bugs.eclipse.org/363242.
[13] https://bugs.eclipse.org/401233.
[14] https://bugs.eclipse.org/365372.
[15] https://bugs.eclipse.org/355568.
[16] Eclipse Documentation on the Refactoring Wizard. http://help.eclipse.org/indigo/topic/org.

`eclipse.jdt.doc.user/reference/ref-wizard-refactorings.htm`.

[17] Ergonomics of human-system interaction—Usability methods supporting human-centred design. ISO/TR 16982:2002.

[18] D. Akers, R. Jeffries, M. Simpson, and T. Winograd. Backtracking Events as Indicators of Usability Problems in Creation-Oriented Applications. *ACM Transactions on Computer-Human Interaction*, pages 16:1–16:40, 2012.

[19] M. S. Andreasen, H. V. Nielsen, S. O. Schrøder, and J. Stage. What Happened to Remote Usability Testing? An Empirical Study of Three Methods. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, pages 1405–1414, 2007.

[20] A. Bruun, P. Gull, L. Hofmeister, and J. Stage. Let Your Users Do the Testing: A Comparison of Three Remote Asynchronous Usability Testing Methods. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, pages 1619–1628, 2009.

[21] P. K. Chilana, A. J. Ko, J. O. Wobbrock, T. Grossman, and G. Fitzmaurice. Post-Deployment Usability: A Survey of Current Practices. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, pages 2243–2246, 2011.

[22] E. M. del Galdo, R. C. Williges, B. H. Williges, and D. R. Wixon. An Evaluation of Critical Incidents for Software Documentation Design. In *Proc. Human Factors and Ergonomics Society*, pages 19–23, 1986.

[23] J. C. Flanagan. The Critical Incident Technique. *Psychological Bulletin*, pages 327–358, 1954.

[24] S. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *Proc. International Conference on Software Engineering (ICSE)*, pages 222–232, 2012.

[25] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[26] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling Manual and Automatic Refactoring. In *Proc. International Conference on Software Engineering (ICSE)*, pages 211–221, 2012.

[27] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University Washington, 1991.

[28] H. R. Hartson and J. C. Castillo. Remote evaluation for post-deployment usability improvement. In *Proc. Working Conference on Advanced Visual Interfaces (AVI)*, pages 22–29, 1998.

[29] H. R. Hartson, J. C. Castillo, J. Kelso, and W. C. Neale. Remote Evaluation: The Network as an Extension of the Usability Laboratory. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, pages 228–235, 1996.

[30] D. Kelly and K. Gyllstrom. An Examination of Two Delivery Modes for Interactive Search System Experiments: Remote and Laboratory. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, pages 1531–1540, 2011.

[31] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *Proc. International Conference on Software Engineering (ICSE)*, pages 23–32, 2013.

[32] E. Murphy-Hill, M. Ayazifar, and A. Black. Restructuring Software with Gestures. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 165–172, 2011.

[33] E. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *Proc. International Conference on Software Engineering (ICSE)*, pages 421–430, 2008.

[34] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving Software Developers' Fluency by Recommending Development Environment Commands. In *Proc. Symposium on Foundations of Software Engineering (FSE)*, pages 42:1–42:11, 2012.

[35] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, pages 5–18, 2011.

[36] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A Comparative Study of Manual and Automated Refactorings. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576, 2013.

[37] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 79–103, 2012.

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

[39] L. G. Shattuck and D. D. Woods. The Critical Incident Technique: 40 Years Later. In *Proc. Human Factors and Ergonomics Society*, pages 1080–1084, 1994.

[40] G. Soares, M. Mongiovi, and R. Gheyi. Identifying Overly Strong Conditions in Refactoring Implementations. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pages 173–182, 2011.

[41] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A Compositional Paradigm of Automating Refactorings. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 527–551, 2013.

[42] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. In *Proc. International Conference on Software Engineering (ICSE)*, pages 233–243, 2012.

[43] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The Need for Richer Refactoring Usage Data. In *Proc. ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 31–38, 2011.

[44] Y. Yoon and B. A. Myers. An Exploratory Study of Backtracking Strategies Used by Developers. In *Proc. International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*, pages 138–144, 2012.

[45] Y. Yoon, B. A. Myers, and S. Koo. Visualization of Fine-Grained Code Change History. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126, 2013.