Use, Disuse, and Misuse of Automated Refactorings (Extended Version)

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{*mvakili2, nchen, snegara2, rajkuma1, bpbailey, rjohnson*}*@illinois.edu*

***Abstract*—Though refactoring tools have been available for more than a decade, research has shown that programmers underutilize such tools. However, little is known about why programmers do not take advantage of these tools. We have conducted a field study on programmers in their natural settings working on their code. As a result, we collected a set of interaction data from about 1268 hours of programming using our minimally intrusive data collectors. Our quantitative data show that programmers prefer lightweight methods of invoking refactorings, usually perform small changes using the refactoring tool, proceed with an automated refactoring even when it may change the behavior of the program, and rarely preview the automated refactorings. We also interviewed nine of our participants to provide deeper insight about the patterns that we observed in the behavioral data. We found that programmers use predictable automated refactorings even if they have rare bugs or change the behavior of the program. This paper reports some of the factors that affect the use of automated refactorings such as invocation method, awareness, naming, trust, and predictability and the major mismatches between programmers' expectations and automated refactorings. The results of this work contribute to producing more effective tools for refactoring complex software.**

***Keywords*-Software engineering; Software maintenance; Programming environments; Human factors; User interfaces; Human computer interaction;**

I. Introduction

Refactoring is defined as changing the design of software without affecting its observable behavior [1]. Refactorings rename, move, split, and join program elements such as fields, methods, packages, and classes. Agile software processes such as eXtreme Programming (XP) prescribe refactoring [2], because it enables evolutionary software design and is the key to modifiable and readable code [3]. Programmers refactor their code frequently [4], [5]. Some refactorings are tedious and error-prone to perform manually. Thus, automated refactorings were invented more than a decade ago to make the process of refactoring more efficient and reliable [6]. Today, modern Integrated Development Environments (IDEs), such as Eclipse [7], NetBeans [8], IntelliJ IDEA [9], Xcode [10], and ReSharper [11], support many automated refactorings.

Recently, there has been much interest in improving the reliability of existing automated refactorings and building new ones to automate sophisticated program transformations [12]–[16]. This is not surprising, given the tedium and error-proneness of some refactorings and the perceived benefits of their automation. In spite of the growing interest in improving the usability of automated refactorings [17]–[19], this aspect of refactoring has not received enough attention. For example, the user interfaces of refactoring tools have changed little since they were first introduced, and recent studies suggest that programmers greatly underutilize the existing refactoring tools [5]. We need to understand the problems programmers have with today's refactoring tools to design future generations of these tools that fit programmers' needs.

We conducted a study consisting of both quantitative and qualitative data collection. We studied 26 developers working in their natural settings on their code for a total of 1268 programming hours over three months, and collected data about their interactions with automated refactorings. We observed patterns of interaction in our quantitative data and interviewed nine of our participants to take a more detailed qualitative look at our behavioral data. Then, we adapted a general framework of human-automation interaction [20] to frame the use, disuse, and misuse of automated refactorings. *Use* of automated refactorings refers to programmers applying automated refactorings to perform code changes they might otherwise do manually. *Disuse* of automated refactorings is programmers' neglect or underuse of automated refactorings. *Misuse* of automated refactorings refers to programmers' use of these tools in ways not recommended by the designers.

Our empirical study sheds light on how users interact with automated refactorings. First, we have found that a single context-aware and lightweight method of invoking refactorings accounts for a significant number of refactoring invocations (See Section III). Second, we have found several factors that lead to the underutilization of automated refactorings such as need, awareness, naming, trust, predictability, and configuration (See Section IV). Third, we have found that programmers usually continue an automated refactoring that has reported some error or warning. This finding casts doubt on the main property of automated refactorings, namely, behavior-preservation. In addition, we

have observed some unjustified uses of the refactoring tool (See Section V). Finally, we have proposed alternative ways of designing refactoring tools based on the findings of our study (See Subsections III-B, IV-G, and V-C).

II. Research Methodology

To understand why existing automated refactorings are underused, we analyzed a large corpus of interaction data gathered from 26 programmers over three months. In addition, we conducted a set of nine semi-structured interviews with developers to understand the rationales of their refactoring practices.

A. Participants

We recruited 16 programmers working on research projects at the University of Illinois at Urbana-Champaign. Eleven of these internal programmers were enrolled in Computer Science graduate programs, and the remaining five were research interns. We also recruited 10 external programmers by sending more than 25 individual emails and posting recruitment messages to the mailing lists and IRC channels of over 40 open-source Java projects.

We asked every participant to fill out a brief survey that collected some demographic information including their years of programming experience and projects. We received the survey results of 24 participants. Based on the survey, 2, 3, 13, and 6 of our participants had 1–2, 2–5, 5–10, and more than 10 years of programming experience, respectively. Our participants reported that they had been working on a diverse range of projects such as banking, business process management, marketing, database management, and projects of six research labs at the university.

B. Data Collection

We gathered the interaction data using our two minimally intrusive data collectors for the Eclipse IDE: CodingSpectator and CodingTracker [21]. Our participants used these tools for about 1268 hours (mean = 49, sd = 46).

Our data collectors were developed to capture data regarding the failure of automated refactorings, context of the failure, configuration overhead, and invocation methods.

CodingSpectator captures data about the use of automated refactorings while CodingTracker collects all manual edits. CodingSpectator collects three kinds of events: *canceled*, *performed*, and *unavailable*. Canceled events are triggered when the programmer cancels an automated refactoring. Performed events occur when the programmer applies an automated refactoring, and unavailable events are triggered when the programmer tries to invoke an inapplicable automated refactoring and Eclipse reports an error.

Eclipse creates *refactoring descriptor* objects for some invocations of automated refactorings and stores them in an XML format. CodingTracker captures the descriptors of all refactorings created by Eclipse, and CodingSpectator creates refactoring descriptors of its own, which capture more data than those of Eclipse. CodingSpectator supported 23 of the 33 automated refactorings supported by Eclipse during the study.

CodingSpectator records the following information in its refactoring descriptors:

1) the time of occurrence of every refactoring event
2) the identifier of the automated refactoring
3) configurations, e.g. input elements, project, and settings that affect the result of the tool
4) information about the selection used to invoke the automated refactoring and its context
5) whether the refactoring tool was invoked using Quick Assist
6) the problems reported by each invocation of an automated refactoring
7) the time spent on each page of the refactoring wizards

Figure 1 illustrates the refactoring descriptor that CodingSpectator captures for an application of the Extract Method refactoring. Due to privacy issues, we use our own examples instead of our participants' data.

```
void printInfo(double amount) {
  printBanner();
  System.out.println("Amount: " + amount);
}
```

```
<refactoring
❶ stamp="1317326947775"
❷ id="org.eclipse.jdt.ui.extract.method"
❸ comment="Extract method 'private void
printDetails(double amount)' from 'Printer.printInfo()'
to 'Printer'" exceptions="false" input="/src<{Printer.java"
name="printDetails"
❹ code-snippet="
void printInfo(double amount) {
  printBanner();
  System.out.println("Amount: " + amount);
}"
selection-text="
System.out.println("Amount: " + amount);"
❺ invoked-by-quickassist="false"
❻ status="<OK>"
❼ navigation-history="
{[ExtractMethod,BEGIN_REFACTORING,1317326935617],
[ExtractMethodInputPage,Preview>,1317326940477],
[PreviewPage,OK,1317326947379],}"
/>
```

Figure 1. The descriptor captured by CodingSpectator for an Extract Method refactoring invoked on the highlighted statements of method `printInfo()` in the top box. See the numbered list of items in Subsection II-B for a description of each group of attributes.

CodingTracker records the edits made inside the Java editors of Eclipse so precisely that it can later replay them to show the code evolution in action. We replayed some of the code edits to obtain more context about some of the refactoring events and estimate the number of lines and files

affected by each automated refactoring. We also used CodingTracker to estimate the number of programming hours of each participants. We computed the number of programming hours by adding up the time intervals between consecutive CodingTracker events that were at most half an hour long. Both CodingSpectator and CodingTracker are open source and available at http://codingspectator.cs.illinois.edu.

The analysis of the interaction data was complemented by conducting semi-structured interviews with nine of our internal participants. Each interview lasted about an hour. During the interviews, we asked questions about participants' awareness and use of automated refactorings. In addition, we prompted the participants with the detailed data that our data collectors had captured and asked them questions about their specific usage patterns such as the refactorings they had performed or canceled, the pieces of code they had refactored, the selections and methods they had used to invoke the refactorings, and the refactoring problems they had received from the tools. The interview script is in the appendix. In this paper, we refer to the i-th interviewee as I_i.

C. Data Analysis

We used theoretical sampling [22] to decide what data to collect and whom to interview. For instance, based on the results of our pilot study on 14 undergraduate students at the university, we decided to study more experienced programmers for a longer period of time. We used an inductive approach for analyzing the qualitative data to reliably decide whether two interviewees had provided equivalent responses. The first author coded the interview scripts to derive the common themes of the interview responses. He listed all responses belonging to each code, and constantly compared and revised the codes until they saturated. Sections III, IV, and V present the core categories of our data, namely, use, disuse, and misuse, and their related categories in subsections.

III. Use of Automated Refactoring Tools

Decisions about the use of automated tools depend on a complex interaction of a variety of factors and are subject to personal differences. The human-automation interaction literature has studied the roles of personal attitudes, mental workload, cognitive overhead, trust, confidence, risk, and other factors on human use of automation [20]. This section discusses the impact of invocation method on the use of automated refactorings.

A. Invocation Method

Eclipse supports several ways of invoking refactorings. The programmer could go to the "Refactor" menu, right click, or use shortcut keys to invoke refactorings. Alternatively, the programmer may use *Quick Fix* or *Quick Assist* (`CTRL+1`) to invoke some of the automated refactorings. Quick Fix assists programmers in resolving compilation problems. Eclipse shows a small icon close to the location of each compilation problem if a Quick Fix is available for the problem. Quick Fix sometimes offers a refactoring to resolve compilation warnings. On the other hand, Quick Assist is not tied into compilation problems, and programmers can use it to perform some common changes such as Rename and Extract Method (See Figure 2 for an example of Quick Assist).

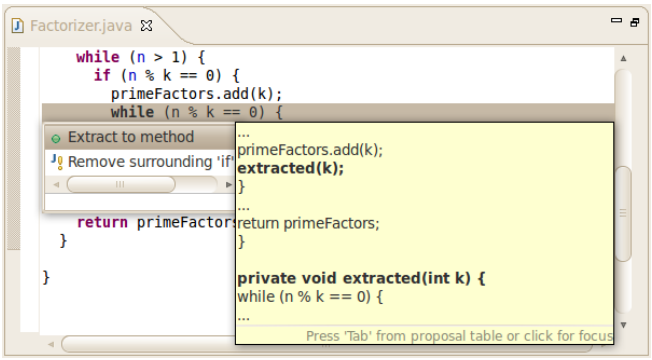

Figure 2. Quick Assist proposes some of the transformations that are applicable to the selected context. The user can single click on each proposed item to preview its effect on the code.

Our results suggest that programmers prefer to quickly apply an automated refactoring and tweak its outcome later rather than spend time configuring the tool up front. Based on the data in Table I, our participants used Quick Assist to perform the refactorings that it supports 35% of the time. Our participants relied on Quick Assist to perform the Rename refactoring less than other refactorings perhaps because Rename is so frequent that they had learned its shortcut key. If we exclude the Rename refactorings, our participants used Quick Assist to invoke 65% of the refactorings. This paper reports the first quantitative results on the use of Quick Assist for performing refactorings.

Of the six interviewees who were aware of Quick Assist, five used it as their primary method of invoking the refactorings supported by it. Quick Assist is a popular method of invoking automated refactorings because it can be quickly invoked via keyboard, narrows the decision space by proposing only a handful of transformations that are applicable to the selected context, and makes it easier to configure refactorings by using some default settings and not opening a dialog. However, we noticed that at least two of our interviewees were not aware that Quick Assist had used some non-default settings from the last configuration of the refactoring.

All of our interviewees were aware of Quick Fix (See Subsection III-A). This awareness could be a result of the visual element that indicates the availability of Quick Fix. Three of our interviewees did not know about the Quick Assist feature. Nevertheless, those who were aware of it

Table I

DATA ABOUT THE USAGE OF AUTOMATED REFACTORINGS FROM 26 PROGRAMMERS FOR ABOUT 1268 HOURS OVER THREE MONTHS. THE CS SUBSCRIPT INDICATES THE DATA CAPTURED USING CODINGSPECTATOR, AND THE CT SUBSCRIPT INDICATES THE DATA CAPTURED USING CODINGTRACKER. IN THE COMPLEXITY COLUMN, S = SIMPLE, M = MODERATE, AND C = COMPLEX. CONFIG_{CS} IS THE AVERAGE CONFIGURATION TIME (SECONDS) OF 788 REFACTORINGS. LINES_{CT} AND FILES_{CT} ARE THE AVERAGE NUMBERS OF AFFECTED LINES AND FILES COMPUTED USING THE DATA AVAILABLE FOR 93% OF THE PERFORMED REFACTORINGS CAPTURED BY CODINGTRACKER. $\text{PR(P | W)}_{CS} = \text{PR(PERFORMED | WARNING)}$ AND $\text{PR(P | E)}_{CS} = \text{PR(PERFORMED | ERROR)}$. THE SYMBOL "-" INDICATES AN UNKNOWN OR UNDEFINED VALUE. PERFORMED_{CS} IS LESS THAN PERFORMED_{CT} FOR CHANGE METHOD SIGNATURE BECAUSE CODINGSPECTATOR DID NOT SUPPORT THIS REFACTORING FROM THE BEGINNING OF THE STUDY.

Automated Refactoring	Complexity	Performed$_{CS}$	Performed$_{CT}$	Canceled$_{CS}$	Warning$_{CS}$	Error$_{CS}$	Fatal Error$_{CS}$	Quick Assist$_{CS}$	Preview$_{CS}$	Config$_{CS}$ (sec)	Lines$_{CT}$	Files$_{CT}$	Pr(P | W)$_{CS}$	Pr(P | E)$_{CS}$
Change Method Signature	C	45	49	8	1	9	0	-	0	8.1	7.48	2.31	1.00	1.00
Convert Anonymous Class to Nested	S	-	3	-	-	-	-	-	-	-	35.00	1.00	-	-
Convert Local Variable to Field	S	97	97	1	0	0	1	83	0	5.6	3.65	1.00	-	-
Encapsulate Field	M	-	225	-	-	-	-	-	-	-	10.44	1.10	-	-
Extract Class	C	-	15	-	-	-	-	-	-	-	120.75	2.25	-	-
Extract Constant	S	29	29	0	0	1	0	26	0	6.9	3.72	1.00	-	1.00
Extract Interface	M	2	2	4	2	0	0	-	1	25.6	36.00	1.50	0.00	-
Extract Local Variable	S	606	606	14	6	8	0	475	0	3.3	2.55	1.00	1.00	0.75
Extract Method	M	186	186	30	0	13	0	62	5	12.2	21.21	1.00	-	0.54
Extract Superclass	C	0	0	3	1	0	0	-	0	3.0	-	-	0.00	-
Generalize Declared Type	M	-	0	-	-	-	-	-	-	-	-	-	-	-
Infer Generic Type Arguments	C	-	7	-	-	-	-	-	-	-	1.29	0.57	-	-
Inline Constant	S	38	38	0	0	0	0	-	0	0.5	1.00	1.00	-	-
Inline Local Variable	S	182	182	1	0	0	0	73	0	0.4	3.26	1.00	-	-
Inline Method	S	63	63	8	0	3	1	-	0	1.5	9.97	1.13	-	0.67
Introduce Factory	M	-	0	-	-	-	-	-	-	-	-	-	-	-
Introduce Indirection	M	-	2	-	-	-	-	-	-	-	20.50	4.50	-	-
Introduce Parameter	C	-	46	-	-	-	-	-	-	-	4.74	1.52	-	-
Introduce Parameter Object	C	-	0	-	-	-	-	-	-	-	-	-	-	-
Move	C	147	147	8	0	3	3	-	10	5.5	12.01	2.19	-	1.00
Move Method	C	0	0	10	3	0	0	-	3	16.5	-	-	0.00	-
Move Static Member	M	6	6	1	1	0	0	-	0	13.0	45.20	3.20	1.00	-
Move Type to New File	S	-	7	-	-	-	-	-	-	-	54.50	1.00	-	-
Pull Up	C	9	9	0	5	0	0	-	1	8.9	11.78	2.89	1.00	-
Push Down	C	8	8	3	2	3	0	-	9	39.0	32.25	12.75	0.50	0.33
Rename Class	M	276	276	37	41	16	5	20	5	8.9	12.13	3.06	0.93	0.62
Rename Enumeration Constant	S	3	3	0	2	0	0	3	0	-	6.00	4.00	1.00	-
Rename Field	M	125	125	9	16	3	6	6	2	2.8	4.52	1.47	0.94	0.67
Rename Local Variable	S	465	465	14	4	16	6	20	0	2.3	3.37	1.00	0.50	0.62
Rename Method	M	260	260	17	33	16	9	15	0	7.4	3.45	2.20	0.94	0.69
Rename Package	M	12	12	0	4	0	0	0	0	6.8	4.67	2.75	1.00	-
Rename Type Parameter	S	6	6	0	0	0	0	0	0	-	2.17	1.00	-	-
Use Supertype Where Possible	C	0	0	7	0	0	0	-	1	5.6	-	-	-	-
		2565	2874	175	121	91	31	783	37	6.30	6.71	1.47	0.88	0.68
		Total Counts								Weighted Averages			Overall Pr	

heavily relied on it to both discover and invoke automated refactorings. I_1 told us:

> Most of them [the automated refactorings] I know about by using Quick Assist. I very seldom go into the refactoring menu and only when there is a refactoring that I cannot reach through Quick Assist and I don't know about [...]. Quick Assist will tell me if they are applicable in a certain context. [...]. It always annoys me when they [automated refactorings] are not available through Quick Assist like Change Method Signature. [...] I really like Quick Assist.

As another example, when we introduced I_2 to the Introduce Parameter refactoring, he commented:

> That's [the Introduce Parameter refactoring] actually pretty cool. I never knew about the existence of this. I've done this a few times manually, and I always wondered if it's possible to do this automatically. Yeah, I'll probably try it. Does this show up in Quick Assist?

B. Implications

We have found that programmers prefer lightweight methods of invoking refactorings like Quick Assist. Therefore, we suggest that other IDEs such as IntelliJ IDEA [9] and NetBeans [8] support refactoring invocation methods similar to Quick Assist.

We noticed that Quick Assist was a somewhat hidden feature of Eclipse. Some programmers will not know about this feature until they somehow learn about the magic shortcut key. More programmers know about Quick Fix because it

has a visual representation. This observation suggests that recommending refactorings similar to the way Quick Fix recommends fixes for compilation problems might promote the use of automated refactorings. While Quick Fix removes compilation problems, automated refactorings remove *code smells*. Code smells are common deficiencies of code that make it less readable and reusable [3, p. 75]. Several tools have been proposed for detecting code smells [23]–[26]. If a code smell detector has a low rate of false alarm and suggests automated refactorings that remove the code smells [20], [27], it may encourage programmers to use the refactoring tool more often. However, metrics for detecting code smells do not rival informed human intuition in practice [3, p. 75]. Perhaps we need systems that facilitate programmers' collaboration on detecting code smells.

IV. Disuse of Automated Refactoring Tools

In the human-automation interaction literature, disuse refers to underutilization of automation [20]. Disuse of automated refactorings occurs when a programmer performs a refactoring manually even though the IDE supports it.

Murphy-Hill et al. inspected a sample of the version control and refactoring histories of Eclipse developers and found that the developers had performed about 90% of their refactorings manually instead of using the refactoring tool [5].

Our interviews provided qualitative evidence for disuse of automated refactorings. For each of the following 15 refactorings, more than half of our interviewees sometimes performed the refactoring manually: Extract Method, Extract Class, Extract Super Class, Extract Interface, Extract Constant, Change Method Signature, Infer Generic Type Arguments, Generalize Declared Type, Use Supertype Where Possible, Encapsulate Field, Introduce Factory, Introduce Parameter, Move Instance Method, Move Static Member, and Pull Up.

In the rest of this section, we will discuss the factors that we have found to influence the disuse of automated refactorings.

A. Need

Some automated refactorings are underused just because programmers rarely need them. For instance, Table I shows that Extract and Pull Up are performed more than Inline and Push Down. Five of our interviewees told us that this was because they usually started with a simple design and gradually made it more general and reusable.

Two interviewees said that it was not worth learning some automated refactorings because they rarely performed the refactorings. For example, I_3 said:

> I know that there are many refactorings. But, many times I think that it's easier to just do something manually than try to learn a very particular refactoring that does something that I don't do very often.

B. Awareness

Programmers must be aware of an automated refactoring to use it. Prior survey studies have reported the role of awareness in the use of refactoring tools [5]. Our interviews showed that even experienced programmers do not know about many of the automated refactorings supported by Eclipse. We asked our interviewees the following three questions about each automated refactoring of Eclipse.

- Did you know that Eclipse supported this refactoring?
- Do you know what this automated refactoring does?
- Do you ever perform this refactoring manually? Why?

On average, our interviewees were unaware of the existence of more than nine automated refactorings of Eclipse. For each of the following refactorings, more than half of our interviewees did not know that Eclipse had automated support for the refactoring: Generalize Declared Type, Use Supertype Where Possible, Introduce Factory, Introduce Indirection, Introduce Parameter, Introduce Parameter Object, Move Type to New File, Move Instance Method, and Move Static Member.

We asked our participants how they learned the automated refactorings in Eclipse and why they knew only a subset of the refactorings in Eclipse. Our interviewees told us that they learned automated refactorings by seeing other programmers using them, reading articles, or exploring the IDE. Our findings corroborate the results of prior studies that identified peer interaction as a mechanism of discovering new tools [28], [29].

We found that our interviewees did not always use all the automated refactorings that they knew about. For each one of Extract Method, Extract Class, Change Method Signature, Infer Generic Type Arguments, and Pull Up, at least five of our interviewees said that they sometimes performed the refactoring manually even though they were aware of its automated support in the IDE. In the rest of this section, we will discuss other reasons of disuse.

C. Naming

It has been assumed that recalling the names of automated refactorings is a barrier to using refactoring tools [19]. Our study provided more evidence that automated refactorings whose names are hard to understand, too technical, or confusing are more likely to get underused. I_4 told us:

> Generally, I don't try them if I don't know what they do. I might occasionally try them if I can kind of guess what they do even though I'm not sure, but I don't do that very often.

Our interviewees did not know the goals of more than eight automated refactorings on average. That is, our interviewees did not know what each of these automated refactorings did and were not able to correctly guess what the tool was supposed to do based on its name. For each of the following seven refactorings, more than half of our

interviewees could not describe the transformation automated by the refactoring: Extract Class, Generalize Declared Type, Introduce Factory, Introduce Indirection, Introduce Parameter, Introduce Parameter Object, and Move Instance Method.

In particular, the majority of our interviewees confused the three automated refactorings: Infer Generic Type Arguments, Generalize Declared Type, and Use Supertype Where Possible.

D. Trust

Trust influences the use of automation, and reliability is a factor in the development of trust. If the automation is not reliable, the operators are more likely to lose their trust in the automation and stop using it, especially when the automation fails to perform simple tasks. However, if the automation is highly reliable, operators seem to tolerate its occasional failures and continue to use it [20], [30], [31].

We found usability to be a more important factor than reliability on users' trust in a mature refactoring tool like that of Eclipse. Even though others have found subtle errors in the refactoring tools of mainstream IDEs [12], [13], and there are many open issues about the refactoring tools in the bug tracking systems, none of our interviewees mentioned the existence of bugs in automated refactorings as a reason for not using these tools. Nonetheless, I_2 said that he would be more cautious while changing critical code:

> Most of the time, I don't do [an automated] refactoring if it involves very critical codes. I'd rather do it manually. Only things that are so easy that they cannot possibly break, I would not expect them to break.

On the other hand, four of our interviewees did not use some of the automated refactorings because of their usability problems. I_3 said:

> There is also a notion of not trusting the [refactoring] tool. If the interface of the tool is not good enough, how do I know that the implementation is not sketchy?

Our interviewees did not use automated refactorings that they had found to have complex user interfaces and unclear benefits. In general, if the benefits of automation are not readily apparent, humans are less likely to use the automation because of the cognitive overhead involved in evaluating and using the automation [20].

E. Predictability

We have found that the predictability of outcome is an important factor in the use of automated refactorings. Three interviewees did not use some automated refactorings because of their unpredictability. For example, I_3 said:

> [...] If it affects only one file then I kind of know exactly what the refactoring does and I can look at the result instantly afterwards. So, I don't like refactorings that are ambiguous enough that I am not able to guess the final result. [...] If I cannot guess, I don't use the refactoring. I consider it not worth the trouble. [...] If the thing that the [refactoring] tool does is so complicated that it isn't easy to figure out things are alright, I'm kind of discouraged to use the tool.

In the following, we discuss how the complexity and preview of a refactoring affect its predictability.

1) Complexity: It was a challenge to determine the complexity levels of refactorings. Therefore, we used two approaches to estimate the complexities of refactorings. In the first approach, each of the first three authors individually assessed the complexities of *manually* performing the Eclipse refactorings in the IDE. Then, they compared their results and worked together to resolve the disagreements between assignments. Finally, they categorized the refactorings as simple, moderate, and complex. Table I illustrates the complexity level of each refactoring. We have found that our participants tended to perform simpler refactorings more frequently (See Figure 3).

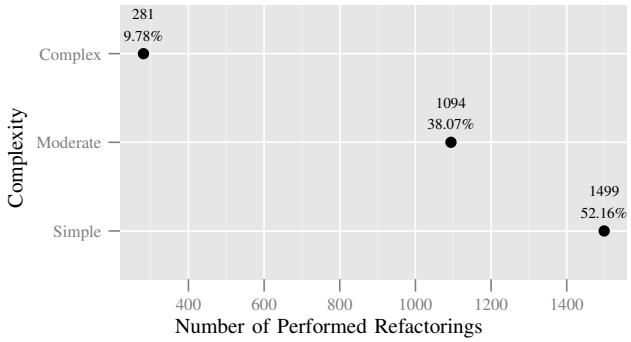

Figure 3. This graph shows the counts and percentages of performed refactorings in each category: simple, moderate, and complex. The counts of performed refactorings in each category are based on the data in columns "Complexity" and "Performed$_{CT}$" of Table I.

Our second approach for studying the effect of complexity on the use of automated refactorings was quantitative. We used the number of files and lines affected by an automated refactoring as an indicator of its complexity. Big refactorings can potentially alter many lines of code across many files. Such big refactorings are tedious and error-prone to do manually. Therefore, one might expect programmers to use automated refactorings for performing bigger changes. However, our data show that 82% of the automated refactorings that our participants performed affected at most six lines and 84% of the performed automated refactorings affected only one file.

There could be various reasons for the low use of automated refactorings that make complex changes. First, there might just be less opportunity for performing large refactorings (See Subsection IV-A). Second, we found that the current design of refactoring tools are not suitable for automating big refactorings. Two of our interviewees mentioned the problem with large refactorings. I_1 said:

> If it does too much, then it will overwhelm me. I will get too many changes at once. I don't like looking at diffs if I don't have to. And, if it does too much for me, I feel like I'm pushed out of the loop. Suddenly, it changes my program in a lot of ways. I will have to go and read these things while I prefer it to do a little for me.

Since the tools for small refactorings affect a narrow piece of the code, it is easier to understand the changes and verify their correctness. In contrast, programmers may worry that the tools for performing big changes may transform their code in unpredictable ways. Disuse of automated support for complex refactorings is consistent with findings from the human-automation interaction literature that imply humans prefer to take ownership of complex tasks and delegate simple ones to the machine [20]. More studies are required to understand the variables that affect the trend in programmers' use of complex refactorings.

2) Preview: Eclipse automated refactorings allow the programmer to preview the changes before applying them. Quick Assist highlights the changes in its preview window (See Figure 2), and refactoring wizards show the code before and after the change side by side. Others identified a usability problem of preview windows based on a survey [5]. CodingSpectator's data provides more evidence for the underutilization of preview windows. Our interviews and quantitative results (See Table I) show that our participants rarely previewed their automated refactorings. We asked five of our interviewees who had used Quick Assist whether they had previewed the refactorings in the Quick Assist menu. All of them told us that they had not previewed the changes using Quick Assist. I_5 told us:

> The scope of the preview is quite small and there is also no highlighting or indentation. So, if the code is a bit more complex, it can get quite difficult to understand.

I_1 gave the following reason for not previewing refactorings.

> [...] quick assist actions are very quick to execute and I can just look at them in the browser [...]. I admit I don't really enjoy looking at diffs and I prefer to get a sense of the change (if it is local) by undoing/redoing, often several times.

Our interviewees mentioned several reasons for not previewing refactorings. First, since they usually used the refactoring tool to perform small changes that were localized to a single method or class (See Subsection IV-E1), they did not need to preview the change. Second, one interviewee said that the preview window was not very useful because it always showed a small portion of the code. Third, the overhead of inspecting the big changes presented in the previews is high. Finally, two of the interviewees said that they could better review and evaluate their refactorings as they performed them manually. For instance, I_6 said:

> Doing it [a refactoring] manually gives me a sense of how things have changed as a design review so that I can see the different options and reevaluate my choices.

F. Configuration

Eclipse lets the programmer perform slight variations of every refactoring by providing a few options. For example, the Extract Method refactoring in Eclipse 3.7 lets the programmer control the access modifier or declared exceptions of the extracted method.

CodingSpectator recorded the time of opening and closing every refactoring wizard (See Subsection II-B). We use the amount of time a refactoring wizard is open to estimate the time needed for configuring an automated refactoring. Based on the data, our participants configured the refactoring tool in at most eight seconds in 82% of the time.

Three of our interviewees complained about the complexity of refactoring wizards. I_4 said:

> To me, whenever you go into these refactorings you have some dialogs and you have to figure out what it's doing and if there's one or two call sites, you can still do it simply manually I think.

Configuration dialogs break the programming workflow and impose an overhead by requiring the programmer to understand the options. More configuration options may make the automated refactoring more powerful but also more complex and harder to understand. Our results provide more evidence for the disruptiveness of refactoring tools that others identified in a survey [5].

G. Implications

Better training on refactorings and their tools may persuade programmers to use automated refactorings more. However, there are other obstacles to the adoption of existing automated refactorings. For instance, designers should choose more intuitive and consistent names for automated refactorings.

Tools that facilitate the exchange of knowledge between programmers can raise awareness of refactoring tools. For example, a tool that uses social indicators to notify members of a software development team about the refactoring activities of other members might encourage programmers to learn more tools from each other [32].

A main motivation of automated refactorings is to reduce the human burden and error in making complex changes to the source code. Researchers have been proposing automated support for complex refactorings [14], [15], [33]–[35]. However, our results suggest that programmers are reluctant to use automated refactorings whose outcomes are difficult to foresee. One may expect previews to help programmers predict the results of automated refactorings. In contrast, we have shown that the current previews of refactorings are not effective. Perhaps more radical ways of reviewing refactorings are needed. The challenge would be to present the changes distributed across the code base in a concise and precise manner. An alternative way of reviewing the changes of refactorings is to provide facilities to inspect the changes after they are performed rather than before. One way to

present the changes after they are performed is to mark up the changes in the editor. It might also be useful to help the programmer navigate through each part of the code that is affected by the refactoring tool. Alternatively, a graphical representation of a refactoring may be more effective for understanding the impact of the refactoring.

High cost of configuration diminishes the value of automated refactorings. Therefore, the designers should make the configuration of refactorings seamless.

V. Misuse of Automated Refactoring Tools

Parasuraman and Riley defined misuse of automation as user's overreliance on automation. According to their definition, misuse of automation occurs when the user relies on the automation even though it would have been better to perform the task manually [20]. We sometimes found it challenging to judge whether a use of an automated refactoring was an overuse or clever use. Therefore, we qualify the definition of misuse to better explain the phenomenon in the context of refactoring tools. We define the misuse of an automated refactoring as use of the automated refactoring in ways not recommended by the designers.

Refactoring tools are designed to preserve the behavior of the program as much as possible except when certain features of the language such as reflection or native code are involved. The Eclipse refactoring tool checks a few preconditions to ensure that it will not introduce compilation problems or change the behavior of the program. If a precondition fails, the refactoring tool reports a problem with a severity level of *information*, *warning*, *error*, or *fatal error*. Warnings of automated refactorings attempt to predict compilation warnings. Errors of automated refactorings predict compilation errors and non-behavior-preserving changes. Thus, Eclipse does not recommend performing a refactoring with errors [36]. Fatal errors indicate that the refactoring tool is unable to carry out the transformation and prevent the programmer from continuing the refactoring. The rest of this section discusses some of the possible misuses of automated refactorings that we have identified.

A. Unsafe Refactorings

When an automated refactoring reports a problem, it is no longer guaranteed to be behavior-preserving. Therefore, we call such a refactoring an *unsafe refactoring*. Traditionally, there has been an emphasis on the behavior-preservation property of refactorings [1], [3]. Our study provides the first quantitative and qualitative results about programmers' use of unsafe refactorings.

A programmer can handle an unsafe refactoring in two ways. First, the programmer might cancel the refactoring, fix the code to satisfy the preconditions, and try the tool again. Second, the user could perform the refactoring and fix the problems afterwards. The former approach provides stronger behavior-preservation guarantees. However, we have found

```
public int getNextNumber() {
   if (new Random().nextBoolean()) return 0;
  return 1;
}
```

Figure 4. If the Extract Method refactoring tool is invoked on the highlighted piece of code, the tool will report the error "Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed."

the latter to be the prevalent approach in dealing with unsafe refactoring. According to the data collected by CodingSpectator, our participants performed 79% of automated refactorings that had reported some error or warning. Table I illustrates the probability of our participants performing each kind of refactoring in spite of a reported problem.

Our participants received a total of 70 different messages from the Eclipse refactoring tool. The following are four of the 15 most frequent problems that the refactoring tool reported to our participants:

1) WARNING: Code modification may not be accurate as affected resource '`resource name`' has compile errors.
2) ERROR: Found potential matches. Please review changes on the preview page.
3) ERROR: Selected statements contain a return statement but not all possible execution flows end in a return. Semantics may not be preserved if you proceed.
4) WARNING: A variable with name '`variable name`' is already defined in visible scope.

Thirteen participants received the first message for a total of 89 times and performed the refactoring 94% of the time. Twelve participants received the second message for a total of 31 times and performed the refactoring 77% of the time.

Figure 4 illustrates how to reproduce an error message that the Extract Method refactoring reported to four of our participants for a total of 13 times. In 54% of the cases, our participants chose to continue the refactoring and manually adjust the compilation problems of the resulting code.

Figure 5 illustrates an example where the Extract Local Variable refactoring warns the programmer about name shadowing. Two participants received this warning for a total of six times and performed the unsafe refactoring five times.

One might argue that programmers perform unsafe refactoring because it is easier to interpret and resolve compilation problems than unfamiliar refactoring ones [37]. We asked our interviewees to explain the refactoring problems that they had received from the tool. Our interviewees had understood almost all of the error messages of automated refactorings. Only one interviewee confused two error messages of the Extract Method refactoring, and at least two of our participants struggled with a strange error message of Extract Method. In every case, they were able to understand and resolve the problem eventually.

```
public class C {
  static int i = 0;
  public static void main(String[] args) {
    System.out.print( 1 );
    System.out.print(i);
  }
}
```

Figure 5. If the programmer uses the Extract Local Variable refactoring tool to extract the highlighted expression to a local variable named `i`, the tool will report the warning "A variable with name '`i`' is already defined in visible scope." If the user continues the refactoring, the compiler will not report any problems, but, the output of the program will change from `10` to `11`.

Our interviewees mentioned that they relied on the compiler, visual inspection, and sometimes running their programs and tests to identify the possible problems of refactorings. However, visual inspection, compiler checks, runs of programs and tests, and code reviews may not catch the subtle errors of refactorings [3, p. 391] (See Figure 5).

The interviewees gave us several reasons for performing unsafe refactorings. First, there is an overhead associated with canceling the refactoring tool and reconfiguring it. Second, the chance of introducing an error when the tool reports a warning is low. Third, our interviewees claimed that they were well aware of the limitations of the refactoring tool and could easily detect and fix the errors introduced by the tool. Fourth, our participants ignored non-descriptive messages. For example, none of our interviewees knew what "potential matches" meant in the aforementioned error message.

Five interviewees said that they sometimes manually performed a few steps of a refactoring and intentionally introduced some compilation problems to find all other places that needed to get updated. Even though this way of performing a refactoring is slower than using the refactoring tool, it is more interactive and gives more control to the programmer.

B. Unjustified Uses

We suspect that at least two of our participants overused the refactoring tool because they told us that they always used an automated refactoring if one was available for their desired task. However, it is not always optimal to use the automated refactorings. For example, one of our participants used the Change Method Signature refactoring to change the visibility of a method. Visibility changes could lead to subtle changes of a program's behavior. Nevertheless, Eclipse does not currently perform the necessary checks to guarantee the behavior-preservation of such refactorings [38]. Thus, the use of the Eclipse refactoring tool to change the visibility of a method, especially in simple cases, is questionable. When we asked the participant about this particular use of the tool, he did not have any justifications.

A combination of excessive trust in the refactoring tool and low confidence in one's coding abilities might lead to the misuse of the tool. For example, an interviewee told us that he always used the tool to perform refactorings because he was afraid of making mistakes in performing the refactoring manually. While it is error-prone to perform some refactorings manually, programmers can perform the rest easily.

C. Implications

We suggest a few techniques for designers of refactoring tools to avoid misuse.

A high rate of false alarms may lead to mistrust of the warnings [20]. Thus, reducing the number of false positives might mitigate the misuse of automated refactorings when they report warnings.

Although it is valuable to communicate the error messages better [18], our participants performed unsafe refactorings even though they had understood the messages. One way to mitigate the risk of unsafe refactorings is to provide specialized tools that verify the results and assist the programmers in completing the transformation.

Our participants' use of unsafe refactorings and reliance on the incremental compiler to perform a refactoring in small steps suggest that predictability and interactivity may be more important factors in the design of refactoring tools than behavior-preservation. If automated refactorings present some of their intermediate results, become more interactive, and give more control to the programmer, they will become more transparent and predictable. As a result, programmers will gain a better understanding of the limitations of the automated refactorings and use them appropriately.

Another technique to reduce the misuse of automated refactorings is to make the tools more flexible. Such a flexible tool will attempt to change the code to satisfy the preconditions or propose possible fixes to the programmer instead of just reporting the problem [13].

Also, we suggest that trainers warn programmers about the possible excessive trust and misuses of refactoring tools. Trainers could make programmers aware of their excessive trust in certain automated refactorings and show them how to avoid or mitigate the consequences of their misuses.

VI. Limitations

Even though a study such as ours captures authentic data, it raises privacy issues and makes recruitment challenging. As a result, the majority of our participants and all of our interviewees were at the university (See Section II). The confidentiality issues might have also affected the projects our participants have enabled our data collectors on.

Because of the uncontrolled nature of our study, the numbers of programming hours of our participants vary a lot (See Section II), and the number of opportunities to perform refactorings on the projects might have been different.

CodingSpectator did not capture data about ten automated refactorings in Eclipse. We prioritized which automated refactorings to study based on the usage statistics reported by others [5], [39].

We fixed some of the bugs of our data collectors during the study and we discovered some bugs in Eclipse that may have affected our data. However, to the best of our knowledge these bugs introduce negligible noise in our data.

We collected data only from Java programmers who used Eclipse. However, we were able to generalize and suggest improvements to other IDEs based on our results.

VII. Related Work

Parasuraman and Riley discussed humans' use, misuse, disuse, and abuse of automation [20]. Our work adapted their framework in three ways to automation of refactorings. First, they defined abuse as enforcing automation by designers and managers without considering the consequences on the users. Since we did not find an evidence of abuse in the context of refactorings, we excluded it from our framework. Second, we used a slightly different definition of misuse. They defined misuse as overreliance on automation. We considered uses of automated refactorings in ways not recommended by designers as misuse. Finally, we have identified the factors that pertain to use, misuse, and disuse of automated refactorings specifically and not automation in general.

Murphy et al.'s study of 41 developers using the Java tools in Eclipse stimulated research in this area [39]. Their study collected frequency data on the invoked perspectives, views, and commands (refactorings being a subset). Their data provided a holistic view of how often various features of Eclipse were used and raised questions about how users were using the features of the IDEs. Our interaction data collection methods are similar because we both collect data from programmers in their natural settings. However, the focus of our work was on refactorings and we supplemented our quantitative data by qualitative ones.

Murphy-Hill et al. analyzed a pool of existing data about refactorings from Murphy et al. [39], the Eclipse foundation [40], and the refactoring histories of 12 Eclipse developers. In addition, they surveyed five Eclipse developers. They were the first to show some evidence for the underuse of automated refactorings, and concluded that further studies are required to understand why developers sometimes choose not to use the refactoring tools. Our study builds upon theirs and discusses the impact of many factors on the underuse of automated refactorings. For example, our results suggest that trust in the state-of-the-art refactoring tools is influenced more by usability than reliability.

In another study, Murphy-Hill et al. examined barriers to using the Extract Method refactoring [18]. They instructed their participants to apply the Extract Method refactoring on a few open-source projects. They observed that users frequently made mistakes in selecting the code snippet to extract and that the error messages from the tools were hard to understand. Based on this observation, they proposed visual representations of the error messages. We also observed that selection problems were common: ten of our participants encountered such problems. However, our interviewees informed us that they were able to interpret the messages. This difference in results may be due to the differences of the studied populations in their levels of expertise or familiarity with the code under refactoring. While some of the results of the two studies overlap, the focus of our work was identifying the factors that deterred programmers from using automated refactorings rather than resolving a specific usability issue, i.e. bad messages, of the Extract Method refactoring.

Mealy et al. listed 38 guidelines for refactoring tools by analyzing the literatures of industrial usability standards and human factors [17]. Our approaches are complementary because we proposed improvements to the design of refactoring tools based on the actual usability problems that our participants had encountered.

VIII. Conclusions

Our quantitative data and interviews revealed many factors that affect the appropriate and inappropriate uses of automated refactorings. We found that programmers do not use some automated refactorings because they are unaware of them, the overhead of learning and configuring some automated refactorings does not justify the few opportunities to use them, the names of some automated refactorings are confusing, and programmers cannot predict the outcomes of complex tools. On the other hand, programmers appreciate the tools that propose applicable refactorings, and are willing to use automated refactorings even when they may change the program's behavior. Our study shows that the major barrier to the adoption of refactoring tools is their usability problems not their rare bugs. These results suggest that designers should aim for flexible, predictable, and truly interactive automated refactorings in the design of next generations of refactoring tools.

Acknowledgment

We thank Chris Aniszczyk, Deepak Azad, Danny Dig, Milos Gligoric, Nathaniel Hirtz, Darko Marinov, and Roshanak Zilouchian Moghaddam for their assistance and our participants for taking part in the study. This work is partially supported by NSF CCF 11-17960, DOE DE-FC02-06ER25752, the Institute for Advanced Computing Applications and Technologies (IACAT), and the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois.

Appendix

This appendix contains the script of our semi-structured interviews.

Section A presents the script common to all interviewees, and Section B presents the questions customized for each interviewee based on CodingSpectator's data.

Instructions delimited by "[ ]" are for the interviewers.

A. Common Script

Thank you very much for participating in our study and accepting our interview invitation. You do not have to be audio recorded and you can quit the audio-taping and/or this interview any time without any penalty. Recording this interview session will help us in referencing and analyzing the interview better in future. Please confirm that you are willing to be audio recorded at this time.

1) Eclipse supports a couple of dozen refactorings. Our data collector has recorded the automated refactorings that you have used. We would like to know more about your familiarity with the refactoring tool and how you use it. Your answers will help us identify the problems of the refactoring tool. I am going to read the list of Eclipse automated refactorings and ask you three questions about each automated refactoring. I will read the refactorings one by one and repeat the following three questions to you for each refactoring:
   a) **Were you aware of this automated refactoring in Eclipse?** Did you know that Eclipse supported this automated refactoring? The purpose of this question is to see if you knew about the existence of this automated refactoring in Eclipse. You might have come across this automated refactoring in the "Refactor" menu or heard about it somehow else but do not know much about it and never used it. Knowing that Eclipse supported a refactoring with this name is sufficient to say that you were aware of it.
   b) **Do you know what this automated refactoring mainly does?** We are interested to know if you are familiar with the goal of the automated refactoring. Even if you have not ever used the automated refactoring and do not know how it exactly works, you might still know the general goal of the automated refactoring. If you do not know what the automated refactoring does, we will ask you to guess by its name. If you cannot tell what an automated refactoring is supposed to do, we will briefly describe it to you. [Refer to http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm for a brief description of each automated refactoring, and do a live demo if necessary.]
   c) **Have you ever performed a similar transformation manually?** You might not know about the automated refactoring or the name of the refactoring but you still could have made similar changes to your code. If you perform the refactoring manually even though Eclipse supports it, we will ask you why you perform the refactoring manually.

   [Ask the above three questions about each of the following automated refactorings. Shuffle the list below for each interviewee.]

   - Rename
   - Extract Method
   - Extract Class
   - Extract Superclass
   - Extract Interface
   - Extract Local Variable
   - Extract Constant
   - Change Method Signature
   - Convert Anonymous Class to Nested
   - Infer Generic Type Arguments
   - Generalize Declared Type
   - Use Supertype Where Possible
   - Inline Constant
   - Inline Method
   - Inline Local Variable
   - Encapsulate Field
   - Introduce Factory
   - Introduce Indirection
   - Introduce Parameter
   - Introduce Parameter Object
   - Move Type to New File
   - Move Instance Method
   - Move Static Member
   - Convert Local Variable to Field
   - Pull Up
   - Push Down
2) How did you learn about some automated refactorings but not the rest? Did you take any training on refactorings? Do you read articles about refactorings or hear about them from your colleagues?
3) How do you decide to use some automated refactorings but not the others? When do you decide to perform a refactoring manually?
4) Do you try the automated refactorings that you cannot tell what they do by their names? In other words, do you try out refactorings whose names you do not recognize?
5) How do you invoke the refactorings, do you use the shortcut key, do you go to the menu, use the context menu, or invoke Quick Assist? [Ensure that the interviewee knows what we mean by Quick Assist, e.g. by showing a screenshot or doing a live demo.] Why?
6) How do you handle automated refactorings that report a message to you? Do you cancel, perform, undo, or preview? Why? How do you verify the changes

performed by the refactoring tool?

B. Customized Scripts

This section instructs the interviewers to ask question about the specific usage patterns of each interviewee.

Look for the following usage patterns or any other interesting ones in the interviewee's usage of the refactoring tool and ask questions about them. Collect all of CodingSpectator data related to these usage patterns and present them to the interviewee to provide him or her with more context about the event in the question.

- If the interviewee has previewed the refactoring, then ask him about it. Specifically, if the interviewee has used the preview window ask how he has used the preview window and what he has been looking for in the preview window. Also, ask the interviewee when and how he uses the preview window. Otherwise, if the interviewee has rarely previewed, ask why he does not use this feature of the refactoring tool.
- If the interviewee has canceled the refactoring tool, especially several times, ask the interviewee why he has canceled the refactoring.
- If the interviewee has performed and undone an automated refactoring, especially when he has undone the refactoring several times, ask the interviewee to explain this usage pattern and its reasons.
- If the refactoring tool was unavailable on an input that the interviewee selected or the refactoring tool has reported a warning, error, or a fatal error, ask the interviewee about this event. First, ask the interviewee if he has understood the message. Ask enough questions to find out whether or not he has been able to interpret the message. For example, ask the interviewee to interpret the message to you. Then, ask how the interviewee has used the message to resolve the problem. Ask the interviewee how he has handled the message. If the interviewee has performed the refactoring in spite of the reported problem, ask why and how he has made sure that the resulting code was correct and fixed it if necessary. If the interviewee has canceled the automated refactoring, ask why he has decided to do so. If he has changed the inputs to the refactoring tool and tried it again, ask about the change. Regardless of what strategy the participant has employed to deal with the messages, ask him about his rationals in this specific case and in general.
- If it has taken the interviewee a long time to configure the refactoring tool, e.g. more than 20 seconds, ask him to explain the reasons of the long configuration time.
- If the interviewee has performed the same refactoring with similar configurations multiple times, ask for more explanation about this pattern.
- If the interviewee has changed the configuration options of the refactoring tool from their default values, ask if the interviewee was aware of the changes to the configuration options. Note that Quick Assist/Fix do not show the configuration options of refactorings. Ask the interviewee to explain his changes to the configuration options and why he has made these changes. Also, ask the interviewee about his use of the preview feature in such cases.
- If the interviewee has performed a refactoring more often than other participants have, ask him for explanation.
- If the interviewee has performed several automated refactorings in a short period of time, ask him how he composed several automated refactorings to perform a bigger change.
- Examine the data to see what refactorings the user has invoked using Quick Assist. Then, ask him when and why he invokes an automated refactoring using Quick Assist or other means. If the interviewee uses Quick Assist to invoke refactorings, ask if he previews the refactoring in the Quick Assist menu and why.

REFERENCES

[1] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, 1992.

[2] K. Beck, *Extreme Programming Explained*. Addison-Wesley, 2010.

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

[4] Z. Xing and E. Stroulia, "Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study," in *ICSM*, 2006.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Software Eng.*, 2011.

[6] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theor. Pract. Object Syst.*, 1997.

[7] "Eclipse," http://www.eclipse.org/.

[8] "NetBeans," http://netbeans.org/.

[9] "IntelliJ IDEA," http://www.jetbrains.com/idea/.

[10] "Xcode," http://developer.apple.com/xcode/.

[11] "ReSharper," http://www.jetbrains.com/resharper/.

[12] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *ESEC-FSE*, 2007.

[13] M. Schäfer, T. Ekman, and O. de Moor, "Sound and Extensible Renaming for Java," in *OOPSLA*, 2008.

[14] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for Reentrancy," in *ESEC/FSE*, 2009.

[15] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for Class Immutability," in *ICSE*, 2011.

[16] J. L. Overbey and R. E. Johnson, "Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools," in *ASE*, 2011.

[17] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving Usability of Software Refactoring Tools," in *ASWEC*, 2007.

[18] E. Murphy-Hill and A. P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," in *ICSE*, 2008.

[19] E. Murphy-Hill, M. Ayazifar, N. Carolina, and A. P. Black, "Restructuring Software with Gestures," in *VL/HCC*, 2011.

[20] R. Parasuraman and V. Riley, "Humans and Automation: Use, Misuse, Disuse, Abuse," *J. Human Factors and Ergonomics Soc.*, 1997.

[21] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson, "The Need for Richer Refactoring Usage Data," in *PLATEAU*, 2011.

[22] J. M. Corbin and A. L. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 2008.

[23] N. Tsantalis, "Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings," Ph.D. dissertation, Univ. of Macedonia, 2010.

[24] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," in *ICSM*, 2004.

[25] E. Murphy-Hill and A. P. Black, "An Interactive Ambient Visualization for Code Smells," in *SoftVis*, 2010.

[26] C. Parnin, C. Görg, and O. Nnadi, "A Catalogue of Lightweight Visualizations to Support Code Smell Inspection," in *SoftVis*, 2008.

[27] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Comm. ACM*, 2010.

[28] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," *XP*, 2000.

[29] E. Murphy-Hill and G. C. Murphy, "Peer Interaction Effectively, yet Infrequently, Enables Programmers to Discover New Tools," in *CSCW*, 2011.

[30] J. D. Lee and K. A. See, "Trust in Automation: Designing for Appropriate Reliance," *J. Human Factors and Ergonomics Soc.*, 2004.

[31] P. Madhavan, D. A. Wiegmann, and F. C. Lacson, "Automation Failures on Tasks Easily Performed by Operators Undermine Trust in Automated Aids," *J. Human Factors and Ergonomics Soc.*, 2006.

[32] N. S. Shami, M. Muller, and D. Millen, "Browse and Discover: Social File Sharing in the Enterprise," in *CSCW*, 2011.

[33] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java Programs for Flexible Locking," in *ICSE*, 2011.

[34] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," in *ICSE*, 2009.

[35] M. Vakilian, D. Dig, R. L. Bocchino, Jr., J. L. Overbey, V. Adve, and R. Johnson, "Inferring Method Effect Summaries for Nested Heap Regions," in *ASE*, 2009.

[36] "Eclipse Documentation on Refactoring Wizard," http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/reference/ref-wizard-refactorings.htm.

[37] E. Murphy-Hill, "A Model of Refactoring Tool Use," in *WRT*, 2009, http://people.engr.ncsu.edu/ermurph3/papers/wrt09.pdf.

[38] F. Steimann and A. Thies, "From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility," in *ECOOP*, 2009.

[39] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, 2006.

[40] "Eclipse Usage Data," http://www.eclipse.org/epp/usagedata.