Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation

Yun Young Lee, Nicholas Chen, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{lee467, nchen, rjohnson}@illinois.edu

***Abstract*—Refactoring is a disciplined technique for restructuring code to improve its readability and maintainability. Almost all modern integrated development environments (IDEs) offer built-in support for automated refactoring tools. However, the user interface for refactoring tools has remained largely unchanged from the menu and dialog approach introduced in the Smalltalk Refactoring Browser, the first automated refactoring tool, more than a decade ago. As the number of supported refactorings and their options increase, invoking and configuring these tools through the traditional methods have become increasingly *unintuitive* and *inefficient*. The contribution of this paper is a novel approach that *eliminates* the use of menus and dialogs altogether. We streamline the invocation and configuration process through direct manipulation of program elements via drag-and-drop. We implemented and evaluated this approach in our tool, *Drag-and-Drop Refactoring* (DNDRefactoring), which supports up to 12 of 23 refactorings in the Eclipse IDE. Empirical evaluation through surveys and controlled user studies demonstrates that our approach is *intuitive*, *more efficient*, and *less error-prone* compared to traditional methods available in IDEs today. Our results bolster the need for researchers and tool developers to rethink the design of future refactoring tools.**

I. Introduction

Refactoring is a *disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior* [1]. The concept of refactoring has been studied and improved since it was first introduced by Opdyke and Johnson in 1990 [2], and is now a well-accepted programming practice. Almost all popular IDEs, such as Eclipse, IntelliJ, NetBeans, Visual Studio, and Xcode, include support for automated refactoring tools. Though no IDE supports all 72 refactorings that Fowler cataloged in his book [1], the number of refactorings that IDEs support has only been increasing. For example, Eclipse 2 (as of 2004) supported 14 refactorings but the most recent version of Eclipse (version 4.2) contains 23 refactorings for Java. The current version of NetBeans supports 18 refactorings and IntelliJ supports more than 30 refactorings.

As automated refactoring tools become more mainstream, there has been much research analyzing their usage patterns. Murphy-Hill et al. analyzed Eclipse refactoring tool usage and concluded that almost 90% of refactorings are performed manually without the help of the tool [3]. Our prior work concluded that programmers, on average, are aware of only eight refactorings in Eclipse [4]. These numbers are discouraging and suggest that refactoring tools are used infrequently. One of the main causes behind their disuse is that the current tools suffer from deep usability problems.

Prior research identified at least three dominant usability problems when using automated refactoring tools [4]–[9]. First, programmers have trouble *identifying* opportunities for using the tool. Second, programmers have difficulty *invoking* the right refactoring from a lengthy menu of available refactorings. Programmers often find the names and the position of the refactorings in the menu confusing. Third, programmers find *configuring* the refactoring dialog complicated. Configuration dialogs disrupt the programming workflow and impose an overhead by requiring the programmer to understand the options. Our prior work estimates that programmers frequently spend up to eight seconds on the dialogs [4]. We term the second and third problems the *invocation* and *configuration* problems respectively (Section II-A). Indeed, in our own user study, we have observed multiple instances where programmers struggle with these very problems, confirming their prevalence and severity (Section V-B2).

We argue that the invocation and configuration problems stem from the overreliance on menus and dialogs in current refactoring tools. Consider the following scenario. Once a programmer decides on a refactoring to perform, she still has to complete two steps. First, to *invoke* the tool, she has to navigate through a lengthy and confusing menu (recall that Eclipse, NetBeans and IntelliJ support at least 18 refactorings) and select the appropriate refactoring. She could memorize an elaborate keyboard shortcut but unless it is a refactoring that she frequently uses, she is unlikely to do so (only 1 out of 10 participants in our controlled user study used keyboard shortcuts). Second, to *configure* it, she has to interact with a dialog containing many detailed options that she might not require and only serve to distract her from her goals (90% of users do not modify the default settings [3]). Thus, there exists a *gap* between *what* she wants to accomplish and *how* she needs to do it through the current user interface.

To bridge this gap, we allow the programmer to directly manipulate program elements, e.g., variables, expressions, statements, methods, etc. in the IDE, eliminating the need for menus or dialogs. The programmer only needs to identify a program element to serve as the *drag source* and another program element to serve as the *drop target*. For instance, to perform an Extract Method refactoring, the programmer would drag the selected expression (source) and drop it into the

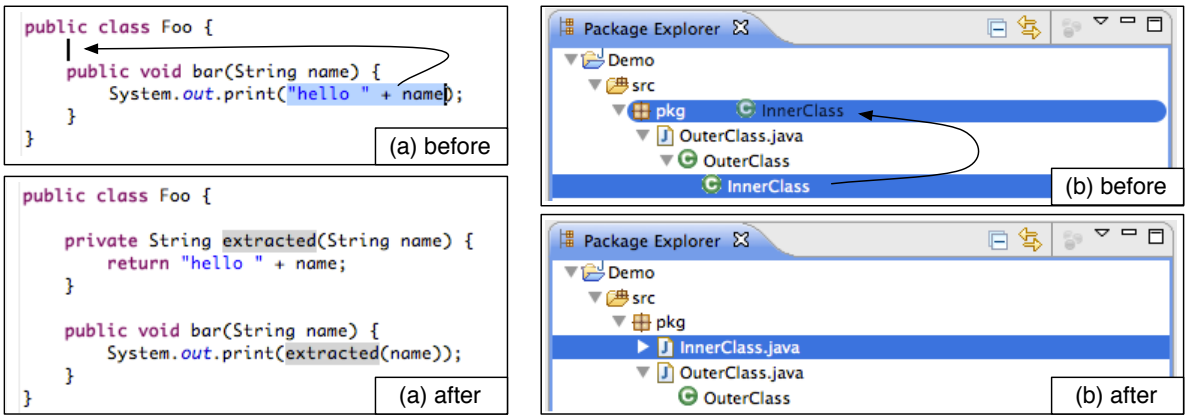


Fig. 1. Drag-and-drop gestures in (a) Java editor for Extract Method refactoring, and (b) Package Explorer for Extract Type to New File refactoring.

enclosing class (target) (Figure 1a). Similarly, to perform the Move Type to New File refactoring, she would drag the inner class (source) and drop it into the desired package (target) (Figure 1b).

Using drag-and-drop has two advantages. First, it eliminates the need to navigate through lengthy menus of refactorings. Second, it eliminates the need for a separate configuration step. Through a single movement of selecting the appropriate source and target elements, the programmer is able to both invoke and configure the desired refactoring. Our approach works for all move and extract based refactorings, and our tool supports up to 12 of the 23 refactorings available in Eclipse. These 12 also happen to be some of the most commonly invoked refactoring tools in Eclipse [3], [4].

Our work makes the following contributions for improving the state of refactoring tools:

1. Approach: We introduce a novel refactoring invocation and configuration approach that relies on drag-and-drop of program elements. This technique leverages the *drag source* and the *drop target* of program elements to invoke and configure the refactoring in a single step. The approach is generalizable to different refactorings and different programming languages.

2. Mappings: For our approach to work, we needed to come up with a suitable set of mappings for drag sources and drop targets. To make it more intuitive, we derived the mappings based on the survey responses of 74 participants. Tables I and II detail the drag sources and drop targets for the supported refactorings. The mappings serve as useful reference for future researchers and tool developers.

3. Tool: We implemented our approach using the mappings in our open source tool, DNDRefactoring, for the Eclipse IDE. DNDRefactoring is supported (i) within a Java editor, or (ii) within and between Package Explorer and Outline views. The Package Explorer and Outline views show a Java element hierarchy tree of the Java projects and source files. We encourage readers to watch a demo of the tool in action at [10].

4. Evaluation: We evaluated our tool for its efficiency and usability. To evaluate the efficiency and usability, we conducted a within-group controlled user study where we asked participants to perform a non-trivial refactoring task using both the existing Eclipse tools and DNDRefactoring. Our results show that DNDRefactoring is not only intuitive but also increases invocation efficiency in terms of decreased configuration time and error rates compared to traditional refactoring tools, which may in turn invite programmers to use the automated refactoring tools more frequently.

II. DNDRefactoring

A. Design Rationale

The driving principle behind the design of DNDRefactoring is to streamline the invocation and configuration mechanisms. The current mechanisms, as implemented in modern IDEs suffer from two problems:

1. Invocation inconsistencies – The dominant mechanism of invoking automated refactorings relies on identifying a refactoring by name and selecting it from a lengthy menu. This mechanism has two shortcomings. First, the names are *non-standard*. For instance, Eclipse adheres to Fowler's naming scheme for Extract Method whereas NetBeans calls it Introduce Method. Second, the grouping of refactorings in the menu is *unpredictable* both within an IDE and across IDEs. For instance, Eclipse places the Rename and Move refactoring in the same category although they are not closely related. Furthermore, while Eclipse groups Extract Superclass and together with Pull Up (because they operate on class hierarchies), IntelliJ groups Extract Superclass with the other extract based refactorings and Pull Up in another category. Both these inconsistencies lead to a *hunt-and-peck* style of invoking a refactoring where the programmer has to spend time searching through the menu. This problem was evident in our user study (Section V-B2) and also corroborated by Murphy-Hill et al. [8].

2. Configuration overload – The dominant mechanism for configuration relies on dialogs. This is a remnant from the design of the first automated refactoring tool for Smalltalk [11]. As more complex refactorings were introduced, more complicated configuration options were also made available. How-

TABLE I
REFACTORINGS WITH DRAG-AND-DROP: WITHIN A JAVA EDITOR.

Drag Source	Drop Target	Refactoring
Local variable	Declaring type	Promote local variable to field (ILE[1])
Expression inside method	Same method	Extract temp variable (ILE)
	Between argument brackets of current method signature	Introduce parameter
	Declaring type	Extract method (ILE)
Statements in method	Declaring type	Extract method (ILE)
Non-static method	Field variable in declaring type	Move instance method to field type
	Argument type in current method signature	Move instance method to argument type
Static method of field	Another type in current editor	Move member to target type
	Field variable in declaring type	Move member to field type
	Local variable type in declaring type	Move member to local variable type
Anonymous class	Declaring type	Convert anonymous to nested type

[1] ILE = In-Line Edit allowed after refactoring is completed.

ever, 90% of refactoring tool users do not modify the default configuration [3]. Thus, these extra options serve only to confuse and prolong the configuration of refactorings since the user is tempted to read all the options. Moreover, we have evidence from our controlled user study (Section V-B1) that some of the options could be erroneously selected by the programmer and could lead to undesired changes to the code.

DNDRefactoring solves both these problems. Because there isn't a universal naming and grouping of refactorings that everyone can agree upon, we dispense with names altogether: the drag source and drop target determines the refactoring to invoke and we do not burden the user with remembering names. Similarly, we do not need dialogs because the drag source and drop target already serve as configuration options to the refactoring tool and we rely on sensible defaults where necessary. Our controlled user study suggests that these options are sufficient; the participants are able to complete the tasks without using more complicated configuration options.

Eclipse already provides a workaround for the configuration overload issue with *Quick Assist* [12], which performs local refactorings with default values and then allows programmers to make changes. Our implementation of DNDRefactoring in Eclipse leverages the Quick Assist paradigm whenever possible, relying on sensible default configurations.

One could argue that the dialog boxes provide more functionality than just configuration and that eliminating them could be problematic. For instance, the dialog boxes also offer a preview feature that shows the code changes to be performed. However, our prior work [4] report that programmers use the preview feature infrequently and prefer to perform the refactoring and view the code changes directly in the editor. If the user is unsatisfied with the changes, she uses the undo feature to revert the refactoring.

B. Tool Features

Our implementation of DNDRefactoring in Eclipse allows programmers to invoke existing refactorings by drag-and-dropping program elements (i) within the Java editor or, (ii) within and between the Package Explorer and Outline View. The drag source is the highlighted selection, either a text selection within a Java editor or a tree node in the Package Explorer or Outline View. The drop target is identified by the position of the cursor when the drag source is dropped. For example, within a Java editor, a cursor located in a whitespace anywhere inside a class, but outside any method and not in any field declaration, will identify the target as the class (Figure 1a). A refactoring is invoked based on the program element types of the drag source and drop target. If no suitable refactoring is found, the drag-and-drop gesture defaults to textual cut-and-paste.

Tables I and II list all the drag-and-drop refactorings that we have implemented for the Eclipse IDE. To the best of our knowledge, the mappings in the tables are *new* and serve as the *first* canonical set of drag-and-drop gestures for refactorings. Other mappings for the stated refactorings are possible, but the current mappings were determined based on the survey responses (Section IV).

In addition to providing a new method of invocation and configuration, DNDRefactoring also supports two new and useful features that can only be accomplished through drag-and-drop gestures.

1. Collated refactorings: A single drag-and-drop gesture can effectively collate several refactorings together. Consider dragging a nested class and dropping it in the *current* package. This gesture can be translated into Move Type to New File refactoring in Eclipse. What happens if the nested class was dropped in a *different* package? Naturally, the extended gesture can be interpreted as Move Type to New file refactoring followed by Move type to target package refactoring. This collated refactoring is supported intuitively and effortlessly in a single drag-and-drop gesture using DNDRefactoring. Such a simple collated refactoring is impossible to invoke using the existing invocation and configuration mechanisms in Eclipse. Programmers using the traditional invocation mechanisms are

TABLE II
REFACTORINGS WITH DRAG-AND-DROP: WITHIN AND BETWEEN PACKAGE EXPLORER AND OUTLINE VIEW.

Drag Source	Drop Target	Refactoring
Non-static Method	Type of field variable in declaring type	Move instance method to target field type
	Type	Pull-up, Push-down or Move method to target type
Nested Type	Package	Move nested type to new file + Move type to target package
Anonymous Type	Type	Convert anonymous to nested type
	Package	Convert anonymous to nested type + Move nested type to new file + Move type to target package
Field	Type	Pull-up, Push-down or Move field to target type
Static Members	Another type declared in current editor	Move members to target type
	Type of field variable in declaring type	Move members to target field type
	Type of local variable in declaring type	Move members to local variable type
Non-static fields	Package	Extract data class + Move type to target package
Non-static methods	Package	Extract interface
Static & non-static methods	Package	Extract super class

forced to perform two separate refactorings in succession. Collated refactorings are annotated with "+" in Table II.

2. Precise control: Another advantage of drag-and-drop is the ability to precisely choose where a drag source is dropped. For example, Extract Method refactoring in Eclipse always creates a new method *below* the method from which the expression or statements were extracted. However, with DNDRefactoring, programmers' natural expectation would be to see the extracted method appear exactly where the expression was dropped (Figure 1a). DNDRefactoring supports such precise control and allows programmers to decide where to move or extract program elements.

C. Supporting Floss Refactoring

Murphy-Hill and Black introduced the term *floss refactoring* to describe refactorings that occur frequently in small steps, intermingled with other kinds of program changes [13]. They also proposed five principles to characterize a tool that supports floss refactoring. They suggest that such tools should let the programmer:

1. Choose the desired refactoring quickly,
2. Switch seamlessly between program editing and refactoring,
3. View and navigate the program code while using the tool,
4. Avoid providing explicit configuration information, and
5. Access all the other tools normally available in the development environment while using the refactoring tool.

Current refactoring tool in Eclipse violates all five principles [13]. The tools by Murphy-Hill et al. help programmers' code selection process (i) with syntactic highlights, (ii) by visualizing nested statements as a series of nested boxes, and (iii) with control and data-flow annotations [9]. While the tools were proven to help reduce time and errors during refactoring, they violate Principles 1 and 4 because the tools do not assist programmers with refactoring selection or configuration. The same limitation applies to tools that alert programmers of code smells and opportunities for refactorings [6] [7]. Murphy-Hill et al. introduced other tools that help with refactoring selection, by mapping directional gestures to refactorings [8]. The tool displays a radial menu with four quadrants, and maps directional gestures (up, down, left or right quadrants) to refactorings. The tool adheres to Principles 1 and 4 because the radial menu displays a more concise set of applicable refactorings and performs the selected refactoring without requiring explicit configuration from programmers. However, the radial menu is a modal window menu that covers up part of the Java editor and thus violates Principles 2 and 3.

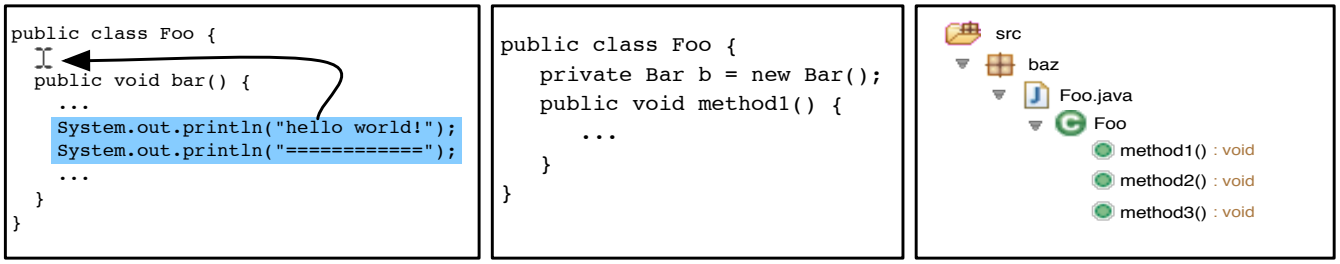
In contrast, we claim that DNDRefactoring satisfies all five principles. DNDRefactoring eliminates the need for programmers to browse through a long list of refactoring menu items and decode refactoring names that aren't always obvious, therefore Principle 1 is satisfied. In addition, because programmers choose source and target program elements in the editors and views that they are currently working on, Principles 2 and 3 are satisfied. DNDRefactoring does not show modal windows during refactoring, so it also adheres to Principles 5. Lastly, DNDRefactoring also adheres to Principle 4 because it does not interrupt refactoring processes with pop-up prompts, but uses default values to complete the refactoring and then invites programmers to make in-line changes.

III. EVALUATION METHODOLOGY

To measure the utility of DNDRefactoring, we ask and answer the following research questions:

RQ1: [Intuitiveness] How intuitive are the drag-and-drop gestures for users?

Given that there is a large set of possible drag sources and drop targets that can be used to invoke each refactoring, the main challenge is to build a set of mappings that is intuitive to most users. To answer whether drag-and-drop gestures are intuitive, we conducted a survey that asked participants unfamiliar with the drag-and-drop approach to suggest drag-and-drop gestures for 5 randomly selected move and extract based refactorings,



a) Mapping from drag-and-drop to refactoring: What refactoring would you expect to invoke with the following drag-and-drop gesture?

b) Mapping from refactoring to drag-and-drop: How would you move Foo.method1() to Bar class?

c) Mapping from refactoring to drag-and-drop: How would you extract an interface to baz package with method1 and method2?

Fig. 2. Examples of Survey Questions

and to select refactorings given 5 drag-and-drop gestures. If the majority of users agree on the drag sources and drop targets for each refactoring, it would strongly suggest that there is a set of drag-and-drop gestures that is universally applicable to all users (Section IV).

RQ2: [Efficiency] How efficient is it to invoke and configure drag-and-drop refactoring?

One of the main challenges of automated refactoring tools is the burden of invocation and configuration. To answer whether drag-and-drop refactoring is efficient, we implemented DNDRefactoring, an Eclipse plug-in that supports the set of gestures that we determined from RQ1. We then conducted a controlled user study comparing DNDRefactoring to the default Eclipse invocation and configuration mechanisms (baseline). Participants were asked to complete a non-trivial refactoring task using Eclipse with and without DNDRefactoring. We recorded videos of these user study sessions, and analyzed them to measure and compare the time taken to invoke and configure both tools. If the results show that DNDRefactoring is more efficient, then it indicates that DNDRefactoring could be compelling and complementary addition to the existing tools (Section V). In addition, we analyzed the physical effort involved in using the refactoring tools, in terms of number of keyboard and mouse actions. The paper omits this data for brevity, but it is available at [10].

RQ3: [Usability] How usable is drag-and-drop refactoring?

A tool can be very efficient to invoke and configure, and yet have very little users because of the difficulties involved in using the tool. We wanted to identify the main challenges of using drag-and-drop refactoring compared to the default Eclipse refactoring tools. To answer this question, we (i) asked the participants to provide feedback on DNDRefactoring and (ii) analyzed the videos of the controlled user study that we captured as part of RQ2 to identified obstacles each participant encountered when using both tools. We then coded and merged those obstacles into key categories following the standard data analysis procedure for open-ended survey responses [14]. By comparing the categories of obstacles identified in each tool, we can objectively discuss the advantages and disadvantages of each tool and suggest room for improvement (Section V). All survey and study materials, and results are available at [10].

IV. Evaluating Intuitiveness

A. Survey Design

We conducted a survey asking participants to suggest drag-and-drop gestures for refactorings. The survey contained five questions asking participants to suggest a refactoring given a drag-and-drop gesture, and five questions asking the reverse mapping. Figure 2 shows actual samples of the questions asked. Participants were given a 5-minute summary of the study and were asked to complete the survey in 10 minutes.

The survey was conducted in a graduate-level software engineering class. At least 95% of the students have taken a prerequisite course in previous semesters that familiarizes them with Java, Eclipse, and refactoring. All participants were new to the drag-and-drop approach. The survey was completely voluntary and anonymous.

B. Results and Observations

We collected 74 survey responses in total. Of those 74 participants, 60 participants (93%) indicated that they have more than 2 years of Java experience, and 58 participants (77%) have more than 2 years of experience with Eclipse. Also, 17 (23.0%), 52 (70.3%), and 5 (6.8%) participants regarded themselves as novice, intermediate, and expert users of the automated refactoring tools in Eclipse, respectively.

We manually coded the responses for each question into two main categories: the majority (the most common response) and minority (Table III). On average 72% of the responses formed the majority. More specifically, on average, 62%, 74% and 90% of refactoring novice, intermediate, and expert users of Eclipse refactoring tool agreed on a mapping, respectively. This result strongly suggests that there is a set of universal drag-and-drop gestures that is applicable for all users. We analyzed all the responses in the majority category and found all of them to be feasible gestures for drag-and-drop. We used these gestures to implement our tool DNDRefactoring[1].

To better understand the range of responses, we divided the minority category into three sub-categories: alternate,

[1]We implemented gestures for 12 refactorings overall. 10 were based on the survey responses, and the remaining two refactorings were conceived by the authors after the survey.

TABLE III
SURVEY RESULTS BY CATEGORY. EACH COLUMN NAME CORRESPONDS TO THE PARTICULAR REFACTORING ASKED IN THE SURVEY.

	Extract Field	Move Nested to New File	Extract Method	Extract Temp Variable	Introduce Parameter	Convert Anonymous to Nested	Push Down	Pull Up	Move Method	Extract Interface
Majority	60(81%)	40(54.1%)	46(62.2%)	59(79.7%)	59(79.7%)	55(73.3%)	49(66.2%)	54(73.0%)	51(68.9%)	61(82.4%)
Alternate	0(0%)	0(0%)	1(1.4%)	0(0%)	0(0%)	7(9.5%)	15(20.3%)	8(10.8%)	3(4.1%)	0(0%)
Infeasible	11(14.9%)	25(33.8%)	19(25.7%)	9(12.2%)	7(9.5%)	5(6.8%)	5(6.8%)	6(8.1%)	7(9.5%)	1(1.4%)
Empty	3(4.1%)	9(12.2%)	8(10.8%)	6(8.1%)	8(10.8%)	7(9.5%)	5(6.8%)	6(8.1%)	13(17.6%)	12(16.2%)

infeasible and empty. The *alternate* category contains different but reasonable alternative refactorings that could be interpreted from the drag and drop gesture. For example, for a question depicting Extract Method refactoring by dragging a set of statements from inside a method and dropping it just above the method declaration (Figure 2a, and Table III), a surveyee answered "[create a] *static* method for class Foo". These responses in the *alternate* group may be supported in future versions of DNDRefactoring. The *infeasible* category includes responses that either conflict with existing refactorings in Eclipse, are not refactorings, or involve infeasible drag sources and drop targets. Lastly the *empty* category contains blank responses.

V. EVALUATING EFFICIENCY AND USABILITY

A. Controlled User Study Design

We conducted a controlled user study with 10 participants to evaluate the efficiency and usability of DNDRefactoring on several refactoring tasks. Each participant carried out the refactoring tasks twice, once using the default tools in Eclipse and once using DNDRefactoring. The order of the tools was randomized to mitigate the learning effect. Each user study session was recorded in its entirety using either a screencasting software or a video camera. To minimize unfamiliarity with different machines, each participant used their own computer or laptop for the user study.

All 10 participants were computer science graduate students majoring in various sub-disciplines, including software engineering and software testing. All participants had at least 2 years of experience in Java; 6 participants had more than 5 years of Java experience. The majority of participants had from 2 to 5 years of experience in Eclipse. 2, 7, and 1 participants regarded themselves as novice, intermediate and expert users of the Eclipse refactoring tool, respectively. After the user study, each participant was asked to complete a post-study qualitative survey to evaluate their experience with DNDRefactoring. Participation was strictly voluntary with no rewards offered, and invitations to the study was sent through individual emails and departmental mailing lists.

The refactoring tasks given to the participants are based on the Refactoring Lab Session exercise developed at LORE [15]. The exercise involves multiple refactorings for a Local Area Network simulation program. The individual refactorings are small and independent, thus are more like floss refactoring than root canal refactoring. We made minor modifications to the refactoring tasks in order to remove some duplicated refactorings and include a wider variety of refactorings.

Prior to their individual user study sessions, all participants were given a group tutorial on DNDRefactoring and the official reference on Eclipse's refactoring tool [16]. The DNDRefactoring tutorial showed a short video demonstrating three refactorings, none of which were repeated in the user study. Participants were encouraged to ask questions and to try using both tools on their own code.

We collected data for two metrics: the configurations times (quantitative) and the obstacles encountered (qualitative). All measurements were done *post*-user-study from the video recordings so as not to affect the participant's performance on the tasks.

For Eclipse's existing refactoring tool, the configuration time starts from pressing the Refactor menu item (either in the tool bar or the mouse button menu) and ends with pressing the Finish button in pop-up modal windows. For Quick Assist, we counted time from the moment the small options window showing to selecting one option. Lastly for drag-and-drop (for DNDRefactoring or existing simple refactoring support in Eclipse), the time was counted from when the programmer starts her selection to dropping the selection.

We define obstacles as programmers' actions that are incorrect or unnecessary for invoking desired refactorings, for example, when a programmer selects a wrong refactoring, cancels a refactoring, invokes a refactoring with an irrelevant program element, or when results do not match programmers' expectations.

B. Controlled User Study Results and Observations

1) Efficiency: Table IV shows each participant's configuration times. There was an outlying case where participant #10 introduced a fault when using Eclipse that caused a unit test to fail. She attempted to fix the fault both manually and by using the refactoring tool and thus skewed the results. We felt that while this case may give an insight to the complexity of current refactoring invocation mechanisms in Eclipse, it is not a fair representation of them. Therefore the data from participant #10 was dropped from our following analysis.

Overall, DNDRefactoring reduced the time spent on configuration by *up to* 9 times. On average, participants performed the refactorings 3 times faster with DNDRefactoring compared to Eclipse. To validate that this result is statistically significant,

TABLE IV
RESULTS OF THE CONTROLLED USER STUDY[1] - CONFIGURATION TIME IN SECONDS.

	Refactoring:	Extract Method	Move Method 1[2]	Move Method 2[2]	Move Method 3[2]	Move Method 4[2]	Anon. Class to Nested	Move Type to New File	Move Class[3]	Extract Class	TOTAL
PARTIC #1	Eclipse	42.3	48.3	22.9	18.5	16.6	1.1	21.6	6.4	42.1	219.8
	DNDR	18.6	12.7	13.4	13.8	14.3		20.9		*miss*	93.7
#2	Eclipse	106.4	71	10.7	7.3	4.4	40.3	52.9	20.2	32.5	345.7
	DNDR	13.8	6.7	2.8	13.2	4.9		22.8		17.6	81.8
#3	Eclipse	18.6	65.4	10.7	8.1	4	13.5	6.2	4.2	151.9	228.1
	DNDR	33.5	3.7	2.9	2.3	1.4		16.8		8.9	69.5
#4	Eclipse	53.3	11.7	18.4	13.1	4	33.5	40.4	13.2	40.5	228.1
	DNDR	55.9	5.8	3.5	2.1	2.3		39.5		23.5	132.6
#5	Eclipse	23.7	93	8.9	29.8	8.2	44.1	42.3	9.5	41.5	301
	DNDR	63.1	5	6.4	1.8	2		13.9		11.4	103.6
#6	Eclipse	10	100.5	3	10.2	2.7	50.8	43.5	6.1	24.2	251
	DNDR	31.3	26.8	1.5	1	1.1		15		15.3	92
#7	Eclipse	22.6	46.3	2.8	10.7	5.2	22.3	39.2	7.5	25.1	181.7
	DNDR	22.8	3.4	1.6	1.6	0.9		23		7.6	60.9
#8	Eclipse	18.8	136.7	4.1	6.7	2.7	28.9	44.8	3.8	23.7	270.2
	DNDR	17.6	1	2.1	1.7	4.1		6.8		21.8	55.1
#9	Eclipse	7	50.7	3.1	4.7	2.3	22.4	5.3	15.3	24	134.8
	DNDR	12.6	1.5	1.5	1.6	1.6		13.7		12.9	45.4
Average	Eclipse	*33.6*	*69.3*	*9.4*	*12.1*	*5.6*	*28.5*	*32.9*	*9.6*	*66.75*	*246.1*
	DNDR	*29.9*	*7.4*	*4.0*	*4.3*	*3.6*		*19.2*		*14.9*	*81.6*
Ave. Time Save[4]		1.1	9.4	2.4	2.8	1.5		3.7		4.5	3.0

[1] Participant #10 introduced a bug while refactoring with Eclipse, thus her data is not included in our analysis.
[2] Not all participants performed these refactorings in the same order.
[3] Time recorded for DNDR is a collated time of Anon. Class to Nested + Move Type to New File + Move Class refactoring.
[4] Calculated by dividing Eclipse Average by DNDR Average, per refactoring.

we used the Wilcoxon Signed Rank Test (WSRT) to do a pair-wise comparison between the configuration times for Eclipse and DNDRefactoring. We used WSRT to (i) compare the total times for the *entire* study and (ii) compare the configuration times for *each* refactoring. WSRT was used instead of the t-test because we cannot assume that the data (configuration time) is normally distributed. Participant #1 did not complete the Extract Class refactoring, so her data was excluded from the calculation of Total Time and Extract Class refactoring. Configuration times for all four Move Method refactorings were combined for simplicity, and Anonymous Class to Nested Class, Move Type to New File, and Move Class refactorings for Eclipse were also collated because the three refactorings can be performed as one refactoring with DNDRefactoring. The p values are reported in the following table; all except Extract Method show statistical significance ($p < 0.01$).

Total Time	Extract Method	Move Methods	Collated Refactor-ings	Extract Class
p = 0.004	0.715	0.002	0.002	0.004

The results suggest that DNDRefactoring is more efficient compared to Eclipse, except for Extract Method. There are two possible explanations for the inefficiency with Extract Method refactoring. First, the method from which subjects were asked to drag an expression was particularly long, and some found it difficult to drag the expression out of the method while having to scroll the editor. Second, Extract Method is one of the most popular refactorings [3], and as such, many of the subjects may be familiar and efficient enough with its configuration details.

The error case of participant #10 provided an insightful opportunity to observe how programmers may introduce bugs while interacting with Eclipse's refactoring interfaces. We were able to retrace and replay her refactoring actions by using Eclipse's refactoring history [17] and interviewing her after the user study. The bug was introduced while she was moving a method from one class to another, and when one of the references to the moved method was not updated. She invoked the Move refactoring and followed the modal instructions, and opted to view the preview of the changes. Eclipse's refactoring preview window shows a list of Java source files that will be changed by the current refactoring, and allows programmers

TABLE V
RESULTS OF THE CONTROLLED USER STUDY - OBSTACLES.

Obstacles	PARTIC #1	#2	#3	#4	#5	#6	#7	#8	#9
ECLIPSE									
Cancels	1	2	2	0	2	1	0	3	0
Manual Changes	3	1	2	1	1	0	1	1	2
Wrong Refactoring Selected	0	1	0	0	2	4	2	2	0
Correct Refactoring Unavailable	0	0	1	0	0	0	0	0	0
Cannot Choose a Refactoring	0	0	0	4	5	2	0	1	1
Incorrect Configuration	0	0	0	0	0	1	0	2	0
TOTAL	7	4	5	5	10	8	3	9	3
DNDRefactoring									
Cancels	2	0	0	3	1	1	0	0	0
Manual Changes	1	2	2	2	1	1	1	2	2
Wrong Source/Target	0	0	0	1	2	1	0	0	0
Difficulty with Selection	0	0	0	1	0	0	0	0	0
TOTAL	4	2	2	7	4	3	1	2	2

to exclude any file from the changes. During the interview, participant #10 stated that she remembers seeing one of the files being excluded seemingly by default. Upon replaying her refactoring history we concluded that the exclusion of a file was indeed the source of the bug, but also confirmed that Eclipse by default does *not* exclude any file from the change list. We conjecture that she had mistakenly or unconsciously excluded a file but because it appeared to her to be a default setting, she accepted it to be correct. While anecdotal, this case demonstrates the danger of configuration overload – it is too easy to erroneously select a wrong option. DNDRefactoring uses the default refactoring configurations and thus streamlines the refactoring process, and does not burden the programmers or provide an opportunity for accidental bugs.

2) Usability: We analyzed the video recordings to identify obstacles that the participants encountered while performing the user study using Eclipse and DNDRefactoring. We iterated through this list to code and merge similar items into categories. Table V shows the categories we identified. "Cancels" refer to when programmers cancel a refactoring during configuration, or undo an already-executed refactoring. "Manual changes" are when programmers opt to perform any refactoring by hand even though the refactoring tool in use supports it. Other categories names are self-explanatory.

"Correct Refactoring Unavailable" was an unexpected obstacle. The refactoring tool in Eclipse infers to some extent what refactorings a programmer is trying to invoke based on the current cursor position in an editor, and prompts the programmer with a subset of applicable refactorings based on the cursor position. While useful, this inference can sometimes be counter-intuitive or unexpected, which was the case with participant #3. A slight misplacement of the cursor precluded the refactoring he wanted from appearing in the menu. On the other end of the spectrum was "Cannot Choose a Refactoring" obstacle. A number of participants struggled to pinpoint a desired refactoring in the long list of refactorings.

The participants encountered the most number of obstacles when invoking Move refactoring with Eclipse. Eclipse's Move refactoring window shows a list of objects with their instance name and type, one from which a programmer can choose to move a method or field to. Many participants found the list confusing, and 6 of them canceled it up to 3 times, often spending much time studying the configuration details.

Many participants also missed a configuration opportunity to change the access modifier when invoking Convert Anonymous Class to Nested Class refactoring, and manually changed it after the refactoring was completed. In an extreme case, participant #5 opted to perform Extract Class refactoring manually while using Eclipse. Selecting the right program element to invoke refactorings was also difficult with Eclipse. For example, in order to extract a data class with a subset of fields declared in a class, 6 participants selected only the relevant fields in a Java editor and invoked the Extract Class refactoring, but Eclipse by default selects all available fields which *silently* discarded the participants' preliminary actions. At least one participant did not notice the default configuration and proceeded, eventually undoing the refactoring.

With DNDRefactoring, three participant selected wrong drop targets while invoking Extract Method, Convert Anonymous Class to Nested, and Extract Data Class refactorings. Most notably, few participants found it difficult to drag an expression out from a long method to invoke the Extract Method refactoring. Also, at least one participant struggled with selecting an expression that is nested within a line of code. Most manual changes made while using DNDRefactoring were for refactorings that DNDRefactoring currently does not support, including Rename and Change Method Signature. On average, DNDRefactoring halved the number of obstacles that participants encountered compared to Eclipse.

C. Post-Study Qualitative Survey Results

We asked each user study participant to answer a qualitative survey after they completed their tasks. Of the 10 user study participants, 9 found their interaction with DNDRefactoring to be very satisfactory, and 1 found it somewhat satisfactory.

Also, 6 participants answered that DNDRefactoring was very comfortable to use while 4 reported that it was somewhat comfortable, and 7 participants found the translation from drag-and-drop to refactorings as expected but 3 found at least one of the refactorings unexpected (refactoring for extracting a data class), or the occasional lack of immediate in-line edit support a little cumbersome. We plan to mitigate these issues in the future, as detailed in the Section VIII. All 10 would recommend DNDRefactoring to other people and some also suggested that it should be included as part of the Eclipse IDE. Some participants stated that DNDRefactoring "[is] very intuitive especially without knowing what the refactoring jargon means" and "saves me the trouble of remembering the exact refactoring to invoke", and that they "liked that several collated refactorings were invoked with a single action."

VI. Limitations

A. Threats to Validity

1) Internal Validity: We allowed participants to use their own machines for familiarity. These machines varied greatly in terms of specifications and operating systems. Such differences could have affected the configuration time, e.g., using a trackpad instead of a mouse for drag-and-drop, and having a larger screen requires dragging more. Also, while we minimize intervention with participants during the controlled user study, the presence of an external viewer (to ensure that we could successfully video capture their session) might subconsciously affect the participants' performance. Lastly, because participants were aware that DNDRefactoring is a new addition to Eclipse that we have developed, it might have biased them toward/against the approach.

2) External Validity: Our survey and user study participants were advanced undergraduate and graduate students in Computer Science at the University of Illinois. Although collectively the participants have diverse experiences with Java, refactoring, and Eclipse, they might not be representative of all software developers who use refactoring tools. Perhaps within a larger group, different gestures might be suggested for each refactoring. Also, while the refactoring exercise from LORE that we used in our user study is well-known and often used in software engineering classes, it involved only a subset of the refactorings supported by DNDRefactoring. We prioritized keeping the exercise short to enable participants to finish within an hour. Therefore we don't have data on the performance of DNDRefactoring for the untested refactorings. Lastly, we implemented DNDRefactoring only in Eclipse and compared it to the default refactoring tools in Eclipse. While most refactoring tools in different IDEs follow a similar dialog-based approach, subtle difference between each IDE could still affect the comparison with a drag-and-drop implementation.

3) Reliability: All experimental materials and collected data are available online. This allows an interested reader to replicate our results.

B. Limitations of DNDRefactoring

One limitation of DNDRefactoring is the difficulty of translating some refactorings into drag-and-drop gestures. Currently DNDRefactoring only supports move and extract based refactorings. It is difficult, for example, to translate Rename refactorings in drag-and-drop gestures. Secondly, perhaps mirroring the first limitation, is that some drag-and-drop gestures can be translated into multiple refactorings. For example, drag-and-dropping an expression from within a method to its declaring class can easily translate into both Extract Method and Extract Constant refactorings. In an effort to follow our initial design goal of not interrupting programmers during the execution of refactorings, we default to the Extract Method refactoring. We plan to support multiple refactorings in the future by, for example, prompting programmers with a set of refactoring previews in small tooltips that they can choose from when they drop their drag source.

While useful, drag-and-drop also has some shortcomings. One of the major concerns with drag-and-drop is that the entire gesture has to be completed in a single motion. This can be problematic when the drag source and drop target are obscured in the user interface, e.g., when the users operate on a smaller screens. Suspendable drag-and-drop techniques such as *Boomerang* alleviate this by allowing the user to first select the drag source, interact with other program elements and resume the drop gesture later [18]. Drag-and-drop can also be problematic on larger screens where the mouse has to travel further distances. *Pick-and-drop* alleviates this by dynamically clustering and displaying the potential drop targets close to the mouse cursor *after* the source target has been selected [19]. Many other extensions are possible. Collomb and Hascoët provide a good introduction to other possible extensions and show how they can be unified to support different use cases [20]. Future work on DNDRefactoring could incorporate some of these extensions to make it easier to use on smaller or larger screens.

VII. Related Work

Drag-and-drop interfaces have traditionally been used in visual programming environments such as Alice [21], EToys [22] and Scratch [23]. In such environments, novice programmers write programs using visual blocks instead of text. Programmers use drag-and-drop as the primary means for organizing and restructuring those visual blocks.

Because visual blocks can be clunky to navigate in large programs, we eschew this approach in DNDRefactoring and implemented it directly in the textual Java editor, Package Explorer, and Outline View. Moreover, simple restructuring of visual blocks merely moves blocks to different locations in the program without considering behavior preservation. DNDRefactoring, on the other hand, intuitively maps each drag-and-drop operation to a corresponding refactoring operation that, when performed, preserves program behavior.

The typical modal window-based approach to invoking and configuring refactorings was introduced in the first refactoring tool, i.e. the Refactoring Browser [11]. For more than a

decade, little has changed in the interface of refactoring tools. Recently, Murphy-Hill et al. introduced new approaches to invocation with selection assists [9] and gesture-to-refactoring mappings [8]. Eclipse and IntelliJ have also introduced in-place refactoring features [24] that allow widely-used refactorings to be configured directly in the editor without the need for a modal window. Commercial tools such as CodeRush with Refactor! Pro [25] also aid programmers' refactoring tasks with suggestions and visual hints within the code, without modal windows. Nonetheless, these new approaches still rely exclusively on keyboard shortcuts and mouse menus. Our work investigates and demonstrates the potential of new methods of invocation for refactoring tools.

While drag-and-drag infrastructure has always been available in modern IDEs, none have truly exploited its capabilities. Existing IDEs such as Eclipse, NetBeans and IntelliJ provide *minimal* support for drag-and-drop refactoring. Currently, the only refactoring supported is Move refactoring, which can be invoked by drag-and-dropping a class into a package in the Outline View. All other drag-and-drop operations are interpreted as plain textual moves. Existing products dedicated to restructuring code only target organizational refactorings between different packages. For instance, Restructurer101 [26] provides a graphical view of all the classes and packages in the system and allows a developer to perform Move refactorings on them via drag-and-drop. To the best of our knowledge, our tool is the *first* to leverage the drag-and-drop as an intuitive way to invoke a variety of refactorings beyond Move refactorings.

VIII. Future Work

The current implementation of DNDRefactoring assumes that programmers can accurately distinguish between different program elements. We believe selection assist tools such as [9] will be an effective complement to DNDRefactoring. Also, visual cues such as highlights or tooltips indicating the specific refactoring that will be invoked may help narrow down programmers' selection of drop targets.

Lastly, we plan to conduct a long-term study to analyze and evaluate the utility of DNDRefactoring in assisting programmers with floss refactorings. Would programmers using DNDRefactoring use the refactoring tool in IDE more often? If so, what kind of refactorings would they use DNDRefactoring for? We plan to collect refactoring data from programmers using DNDRefactoring in the wild, using such tools as [4], and study the impact of DNDRefactoring on floss refactoring.

IX. Conclusions

We presented DNDRefactoring, a novel approach and tool that streamlines refactoring invocation and configuration processes by allowing programmers to refactor their code via drag-and-drop of program elements. DNDRefactoring is a practical and functional demonstration of a radically different user interface for automated refactoring tools. DNDRefactoring eliminates the need for menus and dialogs that programmers have to understand and interact with. Eliminating the menus and dialogs not only makes the process more intuitive, but also increases invocation efficiency in terms of decreased configuration time and error rates compared to traditional menu and dialog approach approach. Our results make a case for the design of next generation refactoring tools that depart from the traditional menu and dialog approach.

Acknowledgment

We thank Darko Marinov, John Brant, Mohsen Vakilian, Milos Gligoric, and Jeff Overbey for their valuable reviews. We also thank all the user study and survey participants. This work is dedicated to the loving memory of Brett Daniel.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1999.
[2] W. Opdyke and R. E. Johnson, "Refactoring, An Aid in Designing Application Frameworks and Evolving Object-oriented Systems," in *SOOPA*, 1990.
[3] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," in *ICSE*, 2009.
[4] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," in *ICSE*, 2012.
[5] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving Usability of Software Refactoring Tools," in *ASWEC*, 2007.
[6] A. O'Connor, M. Shonle, and W. Griswold, "Star Diagram with Automated Refactorings for Eclipse," in *Eclipse*, 2005.
[7] C. Parnin, C. Görg, and O. Nnadi, "A Catalogue of Lightweight Visualizations to Support Code Smell Inspection," in *SoftVis*, 2008.
[8] E. R. Murphy-Hill, M. Ayazifar, and A. P. Black, "Restructuring Software With Gestures," in *VL/HCC*, 2011.
[9] E. Murphy-Hill and A. P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method," in *ICSE*, 2008.
[10] "DNDRefactoring," https://wiki.engr.illinois.edu/display/cs599yyl/DNDRefactoring.
[11] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, 1997.
[12] "Quick Assist," http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickassist.htm.
[13] E. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, 2008.
[14] J. M. Corbin and A. L. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 2008.
[15] S. Demeyer, M. Rieger, B. Van Rompaey, and B. Du Bois, "Refactoring Lab Session," http://lore.ua.ac.be/Research/Artefacts/refactoringLabSession/.
[16] "Refactor Actions," http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm.
[17] "Tips and Tricks (JDT)," http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/tips/jdt%_tips.html.
[18] M. Kobayashi and T. Igarashi, "Boomerang: Suspendable Drag-and-Drop Interactions Based on a Throw-and-Catch Metaphor," in *UIST*, 2007.
[19] M. Collomb, M. Hascoët, P. Baudisch, and B. Lee, "Improving Drag-and-Drop on Wall-size Displays," in *GI*, 2005.
[20] M. Collomb and M. Hascoët, "Extending Drag-and-Drop to New Interactive Environments: A Multi-display, Multi-instrument and Multi-user Approach," *Interact. Comput.*, vol. 20, no. 6, pp. 562–573, Dec. 2008.
[21] M. J. Conway, "Alice: Easy-to-Learn 3D Scripting for Novices," Ph.D. dissertation, University of Virginia, 1997.
[22] "Squeakland: Home of Squeak Etoys," http://www.squeakland.org/about/.
[23] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.
[24] "In-place Refactorings," http://www.jetbrains.com/idea/webhelp/editor.html.
[25] "Refactor! Pro," http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/refactor_pro.xml.
[26] "Restructurer101," http://www.headwaysoftware.com/products/.