A Compositional Paradigm of Automating Refactorings

Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson

University of Illinois at Urbana-Champaign
{mvakili2, nchen, rzilouc2, snegara2, rjohnson}@illinois.edu

Abstract. Though modern IDEs have supported refactorings for more than a decade, recent studies suggest that programmers greatly underuse such tools, especially for *complex* refactorings. Complex refactorings affect several methods or classes and tend to be *tedious* and *error-prone* to perform by hand. To promote the use of refactoring tools for complex changes, we propose a new paradigm for automating refactorings called *compositional refactoring*. The key idea is to perform small, *predictable* changes using a tool and *manually* compose them into complex changes. This paradigm trades off some level of automation by higher levels of *predictability* and *control*. We show that this paradigm is *natural*, because our analysis of programmers' use of the Eclipse refactoring tool in the wild shows that they frequently batch and compose automated refactorings. We then show that programmers are *receptive* to this new paradigm through a survey of 100 respondents. Finally, we show that the compositional paradigm is *effective* through a controlled study of 13 professional programmers, comparing this paradigm to the existing wizard-based one.

1 Introduction

Refactoring is defined as changing code without affecting the observable behavior of the program [5, 7, 19]. Refactoring is not only recommended by expert practitioners [5, 9], but also commonly practiced by programmers [15, 17, 24, 31]. The first refactoring tool was invented more than a decade ago to make the refactoring process faster and more reliable [20]. Today, modern Integrated Development Environments (IDEs), such as Eclipse, IntelliJ IDEA, NetBeans, ReSharper, and Xcode, support many refactorings that rename, move, split, or join various program elements including methods, classes, and packages. In addition, researchers continue to propose new tools for automating complex changes [4, 11, 22, 26, 30].

Despite the expected benefits of automated refactorings, studies have shown that programmers greatly underuse these tools, especially for complex changes. Although complex refactorings are more *tedious* and *error-prone* than simple ones to perform manually, programmers use the refactoring tools mostly for performing simple refactorings such as Rename, Extract Local Variable, and Extract Method [17, 18, 24].

The mainstream refactoring tools follow a wizard-based paradigm. Typically, a programmer selects a piece of code in the editor and invokes an automated refactoring from a menu. The programmer may then change some of the configuration options on the wizard. These options control the outcome of the refactoring by specifying the entities that should be created, copied, or moved. The tool may also preview the change, e.g., by showing snapshots of the affected files before and after the refactoring side-by-side.

Prior studies have identified several problems with the wizard-based paradigm of refactoring [17, 24]. For instance, the long list of automated refactorings in the menu leads to higher *learning* and *invocation* costs. The context-switch from a code editor to a wizard *disrupts* the programming workflow. The wizard imposes an upfront *configuration* cost, making it difficult to *control* the outcome of the tool. The preview page of the wizard is often too cluttered, which makes the refactoring tool less *predictable*. That is, the programmer cannot easily predict how the tool is going to affect her code. Even if a programmer makes her way through all the steps of invocation, configuration, and preview, the wizard may still notify her at the end that the refactoring is impossible or unsafe to perform. These problems call for rethinking the design of refactoring tools.

The main contribution of this paper is a new paradigm, called *compositional refactoring*, for automating complex refactorings. The key idea is to have the tool automate small, predictable changes and let the programmer manually compose these changes into complex ones. For instance, rather than performing a large refactoring such as Extract Superclass in a single step, the compositional paradigm automates small steps, e.g., Create New Superclass and Move Member to Immediate Superclass, leaving it to the programmer to compose these steps.

The compositional paradigm offers a lower level of automation than the wizard-based one by automating small changes. It puts the programmer in control by letting her compose the small changes. Although it may seem counterintuitive that reducing the level of automation improves an automated tool, this phenomenon is not new. Other fields such as aviation, health-care, and manufacturing have gone through a similar process. Motivated by the perceived benefits of automation, highly automated systems were invented, often neglecting the role of the human operators. Further studies showed that often a less automated system with a human-centered design performs better, concluding that *less is (sometimes) more*, when it comes to automation [2, 10, 28].

The idea of compositional refactoring is inspired by our studies of the refactoring practices in the wild. Even though expert practitioners recommend that programmers perform refactorings as a composition of smaller ones [5, 9], little is known about how programmers compose refactorings in practice. Therefore, we mined two refactoring data sets: the *Eclipse foundation* and *Illinois* data sets. The Eclipse foundation has collected data from hundreds of thousands of programmers over the years. Our data mining of this large corpus of data shows that programmers *frequently* invoke a *variety* of multiple automated refactorings within a short period of time. Nonetheless, refactorings invoked in close time proximity may be semantically unrelated. Therefore, we consulted the Illinois data set, which we collected during a prior field study from 30 programmers over eight months. The Illinois data set is smaller but contains more contextual information about refactoring invocations. We manually inspected a sample of its refactorings that were invoked in a short time span. This analysis reveals some of the *rationales* for *systematically* composing refactorings, providing further evidence for the *naturalness* of the compositional paradigm to programmers.

We evaluated the idea of compositional refactoring in two ways. We first distributed an online survey to get early feedback from hundreds of programmers on our design (Section 5). The survey presented mockup screenshots of compositional and wizard-based refactorings and asked the participants to compare the two paradigms. The survey results showed that programmers are *receptive* to the idea of compositional paradigm and provided *improvement suggestions*. This positive response motivated us to implement the compositional paradigm.

We enhanced the design based on the feedback we received from the survey participants. Then, we implemented it as an Eclipse plug-in. Finally, we conducted a lab study with 13 professional programmers at a large software company (Section 6). We instructed the participants to perform a refactoring on an open-source project using the compositional and wizard-based refactoring tools in a random order. Like the survey participants, the majority (nine) of the lab study participants were more satisfied with the compositional paradigm than the wizard-based one. In addition, the participants were more likely to finish the task *correctly* and significantly *faster* in the compositional paradigm.

Overall, the participants of the survey and lab studies appreciated the compositional paradigm because of its perceived higher level of control, easier method of invocation, and interactivity. In addition, they suggested features like abstract views and multi-selections. These results suggest that compositional refactoring is a promising paradigm for assisting programmers in performing complex refactorings. Our work contributes to the refactoring practice in several ways:

1. We provide empirical evidence for the prevalence and nature of automated refactorings that are invoked in close time proximity (Section 2).
2. We discuss some of the rationales for composing automated refactorings based on our manual inspection of the Illinois data set (Section 3).
3. We propose a new paradigm for automating complex refactorings, namely compositional refactoring (Section 4).
4. We provide an implementation of compositional refactoring as an Eclipse plug-in (Subsection 6.1).
5. We show the effectiveness of compositional refactoring through a survey (Section 5) and a lab study (Section 6).
6. We draw implications from our analyses of refactoring usage data, survey study, and lab study for designing future tools that better support complex refactoring.

Our tool and study materials are available at `http://codingspectator.cs.illinois.edu/CompositionalRefactoring`.

2 Frequent Refactoring Sets

In this section, we answer the following research questions:

– How *frequently* do multiple kinds of refactorings occur in a short time span?
– How *diverse* are the refactorings frequently invoked in a short time span?

Answers to these questions provide a bird's-eye view of the phenomenon of invoking several automated refactorings in a short time span.

2.1 Eclipse Foundation Data Set

Usage Data Collector (UDC) is a pre-installed plug-in in Eclipse, which records uses of Eclipse commands, views, and perspectives. UDC generates a fresh identifier for the user and persists it in the home folder of the user. For each event or action performed by the user, UDC captures the timestamp, the event identifier, the user identifier, and the bundle that generated the event. If a user agrees, UDC regularly sends the user's data to the Eclipse foundation's servers. We analyzed a subset of the UDC data that contained information about the invocations of the Eclipse refactoring tool for Java. The Eclipse foundation has released the data from a total of 195,105 programmers who used the Eclipse refactoring tool for Java during 20 months from January 2009 until August 2010.

2.2 Data Analysis

We used the large data set of Eclipse foundation to infer the *frequent refactoring sets*, the sets of automated refactorings that are frequently invoked in a short time span. Since the refactorings invoked in temporal proximity may not be semantically related, this analysis only provides a bird's-eye view of the frequency and variety of compositions of automated refactorings in the wild.

Refactoring Batches Intuitively, a *refactoring batch* is a maximal set of automated refactorings, such that the consecutive refactorings are invoked within a close time proximity. A refactoring batch is a nonempty set of refactoring kinds. For instance, the refactoring batch {Move, Rename} may stand for one or more invocations of Move and Rename in any order within a close time proximity.

We partitioned the refactoring events into refactoring batches using a heuristic. The heuristic uses the large gaps between the invocation times of consecutive refactorings as the partition boundaries. This heuristic is based on the assumption that refactorings invoked far apart in time are less likely to be semantically related. First, the partitioning algorithm sorts the refactoring events of every UDC user by invocation time. Next, the algorithm creates a new batch for each user and adds the kind of the first refactoring event of the user to the batch. If a refactoring event is invoked by the same user within δ minutes of the preceding event, the algorithm will add the kind (Rename, Move, etc.) to the batch of the preceding event. Otherwise, the algorithm will add the kind to a new batch. When the batch of every refactoring event is determined, the algorithm terminates and returns the set of refactoring batches.

Mining Frequent Refactoring Sets A *refactoring set* is a nonempty subset of a refactoring batch. The *support* of a refactoring set R is the fraction of refactoring batches that are supersets of R. We applied a frequent itemset mining algorithm [8, pp. 246–50] on the set of refactoring batches to infer the *frequent refactoring sets*—the refactoring sets with the highest supports.

We used an implementation of the frequent itemset mining algorithm in the statistical computing software R [1]. We provided the algorithm with refactoring batches ($\delta = 10$) and set the parameter `minsup` to 0.001. The output of the algorithm is a list of refactoring sets with a support of at least `minsup`.

We repeated the analysis for each $\delta \in \{5, 10, 20, 40\}$, and compared the resulting frequent refactoring sets. Due to the negligible effect of such changes of δ on the most frequent refactoring sets, we present the results for only $\delta = 10$.

refactoring set	**support**
{Rename}	0.591
{Extract Local Variable}	0.270
{Extract Method}	0.154
{Inline}	0.090
{Extract Local Variable, Rename}	0.076
{Move}	0.058
{Extract Method, Rename}	0.057
{Change Method Signature}	0.055
{Extract Constant}	0.043
{Extract Local Variable, Extract Method}	0.042
{Inline, Rename}	0.033
{Extract Local Variable, Inline}	0.031
{Extract Method, Inline}	0.027
{Convert Local Variable to Field}	0.025
{Move, Rename}	0.024
{Change Method Signature, Rename}	0.022
{Extract Local Variable, Extract Method, Rename}	0.020
{Pull Up}	0.016
{Extract Local Variable, Inline, Rename}	0.015
{Extract Constant, Rename}	0.015

Table 1: The 20 most frequent refactoring sets of UDC active users. A *refactoring batch* is the kinds of a set of automated refactorings such that the consecutive refactorings are invoked within 10 minutes. A *refactoring set* is a subset of a refactoring batch. For instance, the refactoring set {Pull Up} stands for one or more invocations of Pull Up in a refactoring batch. The support of a refactoring set R is the fraction of batches that are supersets of the R.

2.3 Results

The data mining algorithm inferred 47 frequent refactoring sets for all UDC users. However, the vast majority of UDC users use automated refactorings rarely. Most (98.6%) users invoked the refactoring tool at most 50 times, and 98.9% invoked at most five kinds of automated refactorings. We consider users who invoked at least five kinds of automated refactorings for a total of at least

50 times *active* and the rest *inactive*. This leads to 1,188 active users with a total of 112,885 refactoring events.

We hypothesized that the data of inactive users conceals some of the frequent refactoring sets of the active ones. Thus, we repeated the data mining algorithm on the active users alone. This resulted in about three times more frequent refactoring sets ($N = 150$), 44 of which were also inferred for all users. This indicates that limiting the data set to active users uncovers more frequent refactoring sets. Table 1 lists the most frequent refactoring sets of active UDC users. This table shows the following:

1. Programmers invoke a variety of automated refactorings in a short time span.
2. Some refactoring sets with multiple refactoring kinds are more frequent than those with a single kind. For instance, {Extract Local Variable, Extract Method} is about 2.5 times more frequent than {Pull Up}. In other words, a refactoring batch is more likely to contain Extract Local Variable and Extract Method than at least one Pull Up. This result reveals a limitation of prior studies [15, 17, 24], which focused only on individual refactorings.

3 Refactoring Composition Patterns

The frequency and variety of refactoring sets (Section 2.3) led us to the hypothesis that programmers *systematically compose* certain kinds of automated refactorings to apply larger changes. This section presents answers to the following reseach questions:

– Do programmers compose automated refactorings?
– What are some of the rationales for composing automated refactorings?

The analysis of the Eclipse foundation data set showed that certain kinds of automated refactorings (e.g., {Extract Local Variable, Extract Method}) are frequently invoked in a short time span. Although this data set is huge, it does not capture enough context about each event to infer the rationales for invoking several automated refactorings in a short time span. Therefore, we analyzed the smaller but more detailed Illinois data set.

3.1 Illinois Data Set

The Illinois data set comes from two of our Eclipse-based data collectors, namely *CodingSpectator* and *CodingTracker* [24,25]. CodingTracker records applications of all 33 automated refactorings of Eclipse, and CodingSpectator records more detailed data (e.g., the piece of code surrounding the refactored program element and error messages reported by the refactoring tool) for 23 automated refactorings.

The Illinois data set contains data from 30 programmers consisting of a total of 2,296 programming hours over eight months. Fourteen of our participants were external developers who we recruited by sending invitation messages to individual programmers, mailing lists, and IRC channels of open-source projects. We also recruited twelve graduate students and four interns from six research labs at the computer science department of the University of Illinois at Urbana-Champaign. Based on the results of our demographic survey that 28 participants took, 1, 5, 15, and 7 participants had 1–2, 2–5, 5–10, and more than 10 years of programming experience, respectively.

3.2 Data Analysis

The partitioning algorithm (Section 2.2) found 1,633 refactoring batches of 244 kinds in the Illinois data set. We selected 32 kinds of batches, which were frequent in the Illinois data set or contained the frequent refactoring sets of the Eclipse foundation data set. Then, we manually analyzed a random sample of at most ten batches of each kind, leading to a total of 139 batches.

We examined the information captured for each refactoring event in a batch, e.g., the kind, invocation time, error messages, and the piece of code surrounding the selection. Based on these data, we decided if the refactorings in the batch were semantically related, and inferred a rationale for the batch. Next, for each batch kind, we collected the rationales of the batches of that kind. Finally, we collected five *refactoring composition patterns*. A refactoring composition pattern is a recurring set of automated refactorings that programmers compose for similar rationales.

3.3 Results

We found that the majority (81%, i.e., 112 of 139) of the analyzed batches contained related refactorings. The following presents the refactoring composition patterns that we observed in our sample of refactoring batches. Each pattern reveals some of the *rationales* for composing refactorings, providing evidence that programmers *systematically* compose automated refactorings. The value of n in parentheses shows the number of refactoring batches with a particular property.

Refactoring Closely Related Entities ($n = 47$) We found that programmers frequently compose refactorings to refactor closely related entities, which are not related by binding. For instance, the participants composed several Rename refactorings on program entities with similar names ($n = 8$) or a method and the variable that gets the return value of the method ($n = 2$). As another example, our participants performed the Rename refactoring to rename a field and the constructor parameter that initialized the field ($n = 2$).

Refactoring tools only update the entities that are syntactically related. For instance, the Rename refactoring updates the declaration and all references of a name. One recommendation for future tools is to support this refactoring composition pattern by reliably detecting the names that are likely to co-evolve.

```
public static void main(String[] args) {
  int factorial = 1;
  for (int i = 1; i <= 10 ; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(a) The initial code. The programmer extracts `10` into a new local variable `n`.

```
public static void main(String[] args) {
  int factorial = 1;
  int n = 10;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(b) The programmer moves the declaration of `n` to exclude it from her future selection.

```
public static void main(String[] args) {
  int n = 10;
  int factorial = 1;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  System.out.println(factorial);
}
```

(c) The programmer extracts the computation of factorial of `n` into a new method.

```
public static void main(String[] args) {
  int n = 10;
  int factorial = getFactorial(n);
  System.out.println(factorial);
}

private static int getFactorial(int n) {
  int factorial = 1;
  for (int i = 1; i <= n; ++i) {
    factorial *= i;
  }
  return factorial;
}
```

(d) The programmer inlines local variable `n`, which is now used just once.

Fig. 1: Participants composed Extract Local Variable, Extract Method, and Inline Local Variable to extract methods with their desired signatures.

Adapting Extract Method (n = 34) We found that programmers compose three kinds of refactorings, Extract Local Variable, Extract Method, and Inline Local Variable, to adapt the outcome of Extract Method (Figure 1). This refactoring composition pattern consists of three steps: *preparation*, *method extraction*, and *simplification*. First, the programmer performs Extract Local Variable so that the upcoming Extract Method refactoring infers a method parameter corresponding to the extracted local variable. Second, she invokes Extract Method on a piece of code excluding the declarations of any variables added during the preparation step. Finally, the programmer invokes Inline Local Variable to simplify the code. A refactoring batch with method extraction may contain only the preparation step (n = 11), only the simplification step (n = 19), or both (n = 4). It is impossible to configure the refactoring wizard of Extract Method to extract the same method in one step.

Instead of composing three refactorings to include certain parameters in the signature of the extracted method, the programmer could compose just two refactorings, namely, Extract Method and Introduce Parameter. However, there were no instances of the latter in the Illinois data set. In general, the automated Introduce Parameter refactoring is used infrequently and fewer programmers know about it [17, 24]. Nonetheless, a programmer can adapt Extract Method without the need to learn the Introduce Parameter refactoring.

The following are some of the rationales of this composition pattern:

- configuring a refactoring in ways not supported by a wizard
- avoiding the need to learn additional kinds of automated refactorings

If a refactoring tool is aware of composition patterns for adapting a refactoring, it could offer the programmer the opportunity to perform the simplification actions in one step.

Backtracking Refactorings ($n = 12$) Not all refactorings in a batch contribute to the overall effect of the batch. For example, a pair of Extract Local Variable and Inline Local Variable refactorings may cancel the effects of each other ($n = 6$). A refactoring batch may also contain a refactoring that is followed up by an undo operation ($n = 6$). For example, we found that the participants extracted a piece of code into a local variable, undid the refactoring, and finally extracted the same piece of code into a constant ($n = 2$). This indicates that programmers experiment with refactorings or accidentally invoke the wrong refactoring.

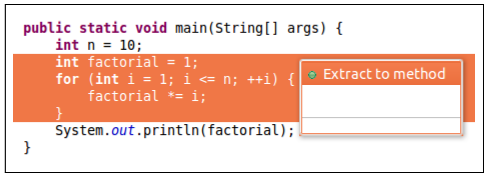
Composition-over-configuration ($n = 8$) *Composition-over-configuration* is composition pattern that we found programmers employ to avoid the upfront configuration cost of refactoring wizards. With this pattern, the programmer composes multiple automated refactorings to perform a refactoring that could have been done by configuring a refactoring wizard.

For example, it is possible to configure the Pull Up refactoring wizard to move one or more members (fields or methods) of a class to a superclass in one step. However, the participants sometimes composed two Pull Up refactorings to pull up two members of a class one at a time ($n = 2$).

As another example, the Extract Local Variable refactoring wizard allows the programmer to set the name of the new variable. Nevertheless, a participant composed Extract Local Variable by a Rename ($n = 1$).

Observing the composition-over-configuration pattern, we propose the compositional paradigm of automating refactorings (Section 4). We implemented the compositional paradigm using a feature of Eclipse called *Quick Assist* (Figure 2). Quick Assist is a popular method of invoking refactorings that supports composition-over-configuration [24]. For example, if a programmer invokes Extract Method through Quick Assist, it would apply Extract Method with a default name and then initiate a composition with Rename on the method name.

Multiple Refactorings on an Entity ($n = 6$) A program entity may undergo multiple refactorings. For instance, the participants composed Extract Method with Pull Up on the same method to refactor to the Template Method design



```
public static void main(String[] args) {
    int n = 10;
    int factorial = 1;
    for (int i = 1; i <= n; ++i) {
        factorial *= i;
    }
    System.out.println(factorial);
}
```

Fig. 2: A screenshot of Eclipse Quick Assist (`CTRL+1`). In this case, Quick Assist suggests Extract Method as a refactoring applicable to the selected piece of code.

pattern ($n = 2$) [6, p. 325]. As another example, a participant composed Push Down with Encapsulate Field on the same field ($n = 1$).

4 Design of a Compositional Refactoring Tool

Based on the lessons learned from our data analysis, literature review, and our prior research studies, we compiled a list of design goals for a new refactoring tool. These goals inspired the design and implementation of a tool for compositional refactoring. The tool currently supports Extract Superclass and Introduce Parameter, and can be easily extended to support other refactorings. In this section, we first discuss our design goals and then explain the steps involved in performing the Extract Superclass refactoring using our tool.

4.1 Design Goals

Predictability Our prior study showed that programmers rarely use the automated refactorings whose outcomes are not easily predictable [24], e.g., the refactorings that affect several files. Our goal is to make such refactorings more predictable. One strategy to achieve predictability, employed by the wizard-based refactorings, is to assist the programmer in reviewing the changes. Another strategy, which we have explored in our compositional paradigm, is to divide a large refactoring into smaller, predictable refactorings.

Control Programmers prefer to maintain control over the evolution of their code during a refactoring [24]. One way to control a refactoring is to allow the programmer to configure it upfront. However, configuration dialogs increase the cost of using the tool [17, 24], and programmers rarely configure the refactoring tool [17]. An alternative paradigm is to put the programmer in control by assisting her in performing the refactoring in smaller steps. In this paradigm, she can evaluate the refactoring at each step and intersperse it with manual edits.

Discoverability Researchers and tool vendors continue to automate more recurring code transformations [4, 11, 22, 26, 30]. However, programmers discover only a subset of the automated refactorings in their IDEs [24]. Quick Assist makes the refactorings more discoverable by proposing them based on the current context. Programmers frequently use Quick Assist to discover and invoke refactorings [24]. Thus, our tool relies on Quick Assist as its method of invocation.

Learnability The list of automated refactorings in modern IDEs is long. The cost of learning so many tools is a barrier to their adoption. Our goal is to solve this problem by allowing programmers learn a small number of *reusable* refactorings and compose them in a variety of ways to perform many kinds of larger refactorings.

Low Disruptiveness Although configuration dialogs make the refactoring tool more powerful and customizable, they distract the programmer from the code and disrupt her flow of programming [17, 24]. We aim to design refactoring tools that are highly interactive and allow the programmer to focus on the code.

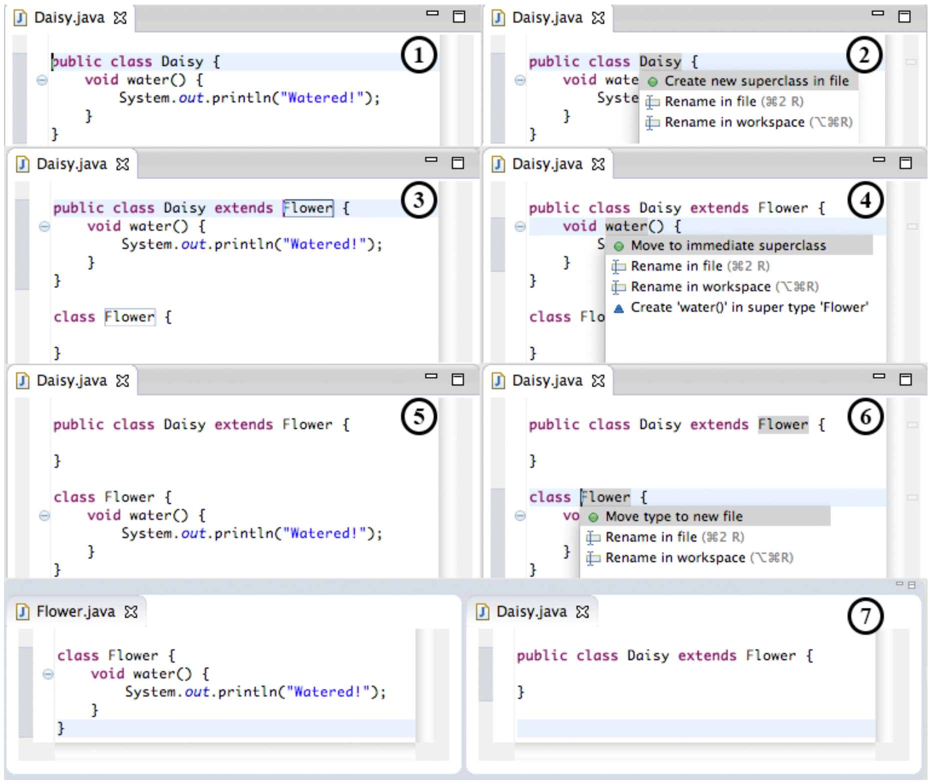
Correctness Wizard-based refactorings guarantee correctness by checking a set of *preconditions*. Similarly, compositional refactorings check the preconditions of the individual steps. Because the steps are small, we expect that programmers can verify them more easily. Moreover, the programmer can run tests after each step. This allows the programmer to identify the exact step that led to a problem.

4.2 Compositional Extract Superclass Refactoring

Our goals informed the design of a compositional refactoring tool. We use the Extract Superclass refactoring as an example to demonstrate the compositional paradigm. This refactoring lets the programmer create a superclass for one or more classes and move some of the members of the subclasses to the superclass. We chose Extract Superclass because it is one of the more complex and less frequently used automated refactorings of Eclipse [17, 24]. Figure 3 shows our mockup of compositional Extract Superclass. We later improved the mockup and implemented it as an Eclipse plug-in (Section 6.1). In the following, we briefly describe how the tool works.

1. The programmer selects the class (`Daisy`) to extract a superclass from.
2. She selects "Create new superclass in file" from the Quick Assist menu.
3. This creates a new empty superclass and prompts for a new name (`Flower`).
4. The programmer invokes "Move to immediate superclass" on method `water`.
5. This moves method `water` from `Daisy` to `Flower`.
6. The programmer invokes "Move type to new file" on class `Flower`.
7. This moves class `Flower` to a new file and completes the refactoring.

The outcome of each of the above steps is immediately visible to the programmer in the code editor. At each step, the Quick Assist menu suggests a set of actions that are applicable to the selected element. The steps are *independent* of each other. That is, Quick Assist suggests the steps regardless of what step was previously performed. The programmer does not have to perform every step using our Quick Assist actions. Rather, she can perform some steps manually.



Daisy.java
1
2
Create new superclass in file
Rename in file (⌘2 R)
Rename in workspace (⌥⌘R)
3
4
Move to immediate superclass
Rename in file (⌘2 R)
Rename in workspace (⌥⌘R)
Create 'water()' in super type 'Flower'
5
6
Move type to new file
Rename in file (⌘2 R)
Rename in workspace (⌥⌘R)
Flower.java
7

```
public class Daisy {
    void water() {
        System.out.println("Watered!");
    }
}
```

```
public class Daisy extends Flower {
    void water() {
        System.out.println("Watered!");
    }
}

class Flower {

}
```

```
public class Daisy extends Flower {

}

class Flower {
    void water() {
        System.out.println("Watered!");
    }
}
```

```
class Flower {
    void water() {
        System.out.println("Watered!");
    }
}
```

```
public class Daisy extends Flower {

}
```

Fig. 3: Mockup screenshots of compositional Extract Superclass. See Section 4.2 for a description of each screenshot. The survey used similar screenshots (Section 5). We later implemented the mockup as an actual tool (Section 6.1).

5 Survey Study

We distributed a survey to assess our design goals and compare our compositional prototype of the Extract Superclass refactoring (Figure 3) with the existing wizard-based user interface of this refactoring in Eclipse. The goal of the survey study is to answer the following research questions:

– How do programmers compare the compositional and wizard-based paradigms?

– Are programmers likely to adopt the compositional paradigm?
– What are some opportunities for improving the compositional paradigm?

Answering these questions shows how *receptive* programmers are to the new compositional paradigm.

5.1 Method

We recruited 100 programmers by announcing the survey[1] on `reddit.com`[2], `twitter.com`, and mailing lists of open source projects. The survey was estimated to take 20 minutes, and started with questions about the experience of the respondent with programming, IDEs, and refactoring. Then, it asked about the programmer's strategy in performing the Extract Superclass refactoring. Finally, the survey presented screenshots of the two user interfaces of the Extract Superclass refactoring, and asked the respondent to evaluate and compare them.

5.2 Thematic Coding

We employed *thematic coding* [29], a systematic qualitative method, to analyze the responses to open-ended questions. The coding was *inductive* (data-driven) as opposed to *deductive* (theory-driven). We extracted the opinions and ideas associated with each segment of the comments. Through an iterative process, we defined, merged, and split the themes to better identify the central ideas. The goal of such a coding is to identify the major ideas not to count the frequencies of keywords. This coding allowed us to reliably decide if two participants provided equivalent responses. For each statement that we report, we include the number of participants that agree with it as an indication of its overall support.

5.3 Participants

The participants were familiar enough with modern programming environments to evaluate the compositional paradigm. The majority (91%) of the survey respondents had more than five years of programming experience. Of all the participants, 76% considered themselves to be experts in at least one programming language (on a five-point Likert scale ranging from "Unfamiliar" to "Expert"), and 99% rated themselves as either very familiar with or expert at one or more languages. The respondents indicated that they were familiar with Eclipse (81%), Visual Studio (58%), NetBeans (39%), IntelliJ (36%), and Xcode (28%).

5.4 Results

The survey presented screenshots of the steps to perform the Extract Superclass refactoring using both the compositional and wizard-based paradigm on the same

[1] `https://illinois.edu/fb/sec/8454746`
[2] `http://www.reddit.com/r/programming`

page. The survey asked the participant how often she would use each interface on a five-point Likert scale ranging from "Never" to "Nearly every time". The majority (66%) of respondents said that if both compositional and wizard-based paradigms are available, they would use the compositional paradigm at least as frequently as the wizard-based one. More interestingly, those who did not use an existing tool for Extract Superclass or used the tool some of the time were more likely to prefer the compositional paradigm (Table 2). This shows that the compositional paradigm is a promising technique for increasing the utilization of automated refactorings.

Table 2: Joint distribution of respondents' frequency of using the Extract Superclass refactoring wizard and their preferred paradigm (compositional vs. wizard-based). Since three respondents did not indicate their frequency of using the wizard, the last row is slightly different from the sum of the other rows.

	Prefers composition	**Has no preference**	**Prefers wizard**
Does not use the wizard	21%	4%	6%
Sometimes uses the wizard	16%	5%	9%
Uses the wizard	15%	3%	20%
All respondents	52%	14%	34%

Finally, the survey asked the participants to compare and evaluate the paradigms. We applied a qualitative analysis method (Subsection 5.2) on the comments provided by 50 participants. The following discusses the result of this analysis, which reveals the *strengths* and *weaknesses* of each paradigm and highlights opportunities for *improvement*.

Control Three survey respondents indicated that they would prefer the compositional paradigm, because it provides more control over the evolution of code. For instance, P_5 wrote:

> I think the second [compositional] UI [...] gives me the idea of having the control over what's happening, and how further can I go with it.

This result is consistent with the findings of a prior study, which showed that programmers prefer to maintain control over their code and use automated refactorings whose outcomes are predictable [24].

Invocation Method Two respondents reported that it was difficult to invoke the wizard-based refactorings from the menu mostly because the menu was too cluttered. Five said that they would prefer keyboard shortcuts. For instance, P_{10} suggested the following as a way to improve the wizard-based interface:

> Make refactoring initiated by keyboard short-cut and not buried so deeply in a menu.

However, as one respondent said, keyboard shortcuts are hard to remember. Quick Assist is a middle-ground, because it is keyboard navigable and proposes only the refactorings that are applicable to the current context. Two participants said that Quick Assist was an easier way of invoking refactorings.

Incrementality and Testability Six respondents mentioned that they do not want refactoring tools with modal dialogs. Five said that modal dialogs are distracting, and three said that the dialogs are too complex. On the contrary, five people indicated that the compositional paradigm is more interactive and two indicated that it allows running tests after each step. For instance, P_7 said:

> The second [compositional] one provides a more stepwise view, giving me more intermediate feedback, as well as an ability to run my tests at each step. This goes a long way to making sure the refactoring is the right decision.

Nonetheless, one respondent said that the compositional paradigm had too many steps, and two preferred to perform the refactoring in a single step. For example, P_8 said:

> I don't write Hello World examples. I need control over what gets moved up and what not, what's made abstract and so forth. I want to do this in one pass, not six [four].

We made compositional paradigm incremental to achieve high control and testability. We decided that compositional refactoring fits programmers' workflow, because it mimics manual refactorings and programmers already compose refactorings (Section 3).

Abstract View Nine participants were concerned that the compositional paradigm may not be suitable for large refactorings. For example, P_2 said:

> [I'm] Not sure I'd want to use that [wizard-based] UI for any refactoring work. However, [the wizard-based UI is] probably better for very large refactoring tasks than the second [compositioanal] UI—but if you're doing that, you're doing too much in one go.

Four respondents said that a high-level view of the code would be useful for performing large refactorings. For instance, P_3 said:

> [...] However, specifying the methods to be moved one by one rather than selecting them from a list might cause methods that should be extracted into a superclass to be missed. In some sense, it is mostly about whether an abstract view of the methods is preferable to a code level view when choosing whether to extract them. Sometimes I find myself leaving the extract superclass dialog to figure out what a method actually does and whether it should be extracted.

The wizard-based paradigm lets the programmer operate at the level of classes and methods, but, makes it difficult to switch between the code and its higher-level view. On the other hand, the compositional paradigm that we demonstrated on the survey was tightly coupled with the code, which makes it easy to intersperse low-level code changes with refactorings. To offer the benefits of both, the tool could make the switch between the two views seamless, e.g., by making the refactorings available both in the code editor and graphical views.

Multi-selections Five respondents preferred to be able to select multiple program entities and manipulate them at the same time. For example, P_4 said:

> Usually, I'm extracting a common superclass to remove duplication from more than one similar class, so I'd need to be able to select multiple classes.

Extending the compositional paradigm to support an abstract view will make it possible to select multiple program elements from the abstract view in one step.

Coding Conventions The mockups of the compositional paradigm showed how to first create an empty superclass in the same file and move it to a new file later (Figure 3). Although one could move the superclass to a new file right after creating the superclass, nine preferred the tool to adhere to the coding conventions strictly and never introduce two classes in the same file.

6 Lab Study

The survey study showed the overall preference of programmers towards compositional refactoring based on participants' evaluations of the mockup screenshots. The goal of the lab study was to answer the following research questions based on programmers' experience with real tools that support refactoring in the two compositional and wizard-based paradigms.

- Which paradigm of refactoring do programmers prefer?
- Which paradigm is faster?
- Which paradigm is less error-prone?

6.1 Tool

We implemented an Eclipse plug-in to support Extract Superclass and Introduce Parameter in the compositional paradigm. Based on the survey study, we improved the design of our tool in several ways. First, we replaced the "Create new superclass in file" action by "Create a new superclass for T in a new file". We made this change to adhere to the coding conventions of Java more strictly. Second, we added an action to the menu called "Create New Superclass" to support multi-selections. When the user selects multiple classes, e.g., in the Package Explorer view, this action would create an empty superclass for the selected classes. Finally, we implemented additional actions such as "Move type T to a new file", where T is a type name, and "Add parameter to method m for expression", where m is a method name (Introduce Parameter in Quick Assist). However, the participants did not use these three actions as they were not applicable to the refactoring task of the lab study.

6.2 Participants

All of our participants were experienced programmers who used Eclipse for Java development at a large software company. We first ran a pilot study on three programmers. Then, we conducted the main study on 13 programmers. Of all the participants, two reported having five to ten years of programming experience and 11 reported more than ten years. One participant reported that he was familiar with Java, nine participants considered themselves as being very familiar with Java, and three indicated that they were experts. One participant reported that he was somewhat familiar with Eclipse, four participants rated themselves as familiar, seven said they were very familiar, and one rated himself as expert.

6.3 Study Design

We instructed each participant to finish the task in both compositional and wizard-based paradigms (*within-subject*). Each participant tried the paradigms in a random order (*counterbalancing*) to overcome the potential *carryover effect.* We did not ask each participant to try only one paradigm (*between-subject*) for several reasons. First, *individual differences* would affect the results of such a study. Second, a between-subject study requires more participants to draw meaningful conclusions. Third, such a study would only allow a quantitative comparison (e.g., efficiency and correctness) of the two paradigms. However, we felt that such measures were not enough to reliably compare the two paradigms. A participant can offer a qualitative comparison only if she tries both paradigms.

At the beginning of the study, we asked the participants to complete a *pre-questionnaire.* Then, we asked them to perform some introductory tasks to familiarize themselves with the code. We then instructed each participant to perform the task twice in a *random order*, once using the compositional paradigm and another time using the wizard-based paradigm. Finally, we asked the participants to rate the two paradigms of refactoring in a *postquestionnaire* and answer our follow-up questions during a semi-structured interview. The study took about an hour for each participant, and we offered a \$25 gift card to each participant.

Refactoring Task We used a refactoring that had occurred in the open-source project `HTMLParser` as our refactoring task. Kerievsky used this refactoring as an example of the Extract Composite refactoring [9, p. 214] in his book. Several classes of an old revision of the code base exhibit some code duplication. These classes contain a list of elements and a method that iterates over the elements and computes their string representation. The fields had different names in different classes and the methods accessed the elements in slightly different ways. We asked our participants to remove this code duplication between two classes by extracting the common field and method into a new common superclass. We limited the refactoring to two classes to make it feasible to finish the refactoring in about 20 minutes.

Pilot Study During the pilot study, we noticed that some participants accidentally introduced subtle errors while refactoring the code. So, we asked the participants of the main study to check that the unit tests passed at the beginning and end of the study. In addition, we instructed the participants to ensure that the new superclass is only referenced by its subclasses. We decided that the existing uses of the subclasses should not be replaced by the new superclass, because of the dynamic type checks and casts that were used to check for the subtype.

6.4 Data Analysis

We measured the *task completion time* and checked the *correctness* of the performed task to compare the two paradigms quantitatively. If a participant finished the refactoring task, we compared his resulting code with the expected code in the instructions. If the participant missed some expected changes or introduced unexpected ones, we considered it an incorrect refactoring.

Similar to the analysis of survey comments (Subsection 5.2), we employed *thematic coding* [29] to *systematically* analyze the retrospective interviews.

6.5 Results

Task Completion Time The medians of the task completion times using the wizard-based and compositional refactorings were 16.5 and 10.5 minutes, respectively. A Wilcoxon signed-rank test shows that there is a significant effect of refactoring tool on the task completion time ($W = 41$, $Z = 2.25, p = 0.02 < 0.05$, $r = 0.50$, two-tailed). Two participants did not finish the refactoring task using either tools during the allotted time. Another participant could not finish the task using the wizards. We excluded these three participants from our significance test. One participant finished the compositional refactoring faster using the wizards. The other nine participants finished the task faster using the compositional paradigm.

Correctness Participants were more likely to complete the task correctly in the compositional paradigm than the wizard-based one. Seven participants introduced accidental changes to the code base using the wizard-based refactorings, while only one participant left the refactoring incomplete using the compositional paradigm.

The Extract Superclass refactoring wizard has an option called "Use the extracted class where possible", which is checked by default. This option causes the tool to replace all occurrences of the selected classes by the superclass whenever this replacement does not introduce any compilation problems. The Pull Up refactoring wizard has a similar option. Only three participants unchecked this option on the wizard and just one participant noticed the unexpected changes in the preview and deselected them. The other seven participants were surprised when they discovered unexpected references to the new superclass at the end. At that point, it was difficult to revert the unexpected changes because the participants had already changed the code too much since the application of the wizard-based refactoring. Two participants failed to finish the task using wizards because reverting the unwanted changes was too time-consuming for them.

Qualitative The majority of our participants were more satisfied with the compositional paradigm, found it easier to learn and use, felt more control and confidence over the refactoring, and expected more opportunities for using it in their code (Table 3).

Table 3: Number of participants of the lab study who preferred each paradigm of refactoring (the first two rows) with respect to each quality (columns). The last row lists the number of participants with no preference.

	control	**correctness**	**ease of learning**	**ease of use**	**opportunity to use**	**satisfaction**
compositional	9	6	7	7	7	9
wizard-based	3	5	4	2	1	3
no preference	1	2	2	4	5	1

During the interviews, we asked about the advantages and disadvantages of the two paradigms of refactoring. The following presents the themes that we extracted from the participants' responses.

Control Participants felt they had more control in the compositional paradigm, because the steps are small, predictable, and mimic their manual refactorings. One participant said:

> The wizard gives this illusion of just doing everything for you. [...] The downside is that there were a number of options that I read and didn't quite make sense of, and said I guess I don't have to care about that. And, of course, I found my sorrow that that wasn't true. It did things that I completely didn't expect. [...] And, it doesn't give control.

On the other hand, one participant attributed his feeling of control in the wizard-based paradigm to the previews.

Correctness A participant said:

> The thing that I like about it [compositional paradigm] is that you're taking actions yourself. So, when you see an error, you usually have an idea of which action that you took caused the error.

Another participant explained why he did not trust the correctness of compositional refactorings as much as the wizard-based ones as follows:

> I was not sure if it [the compositional refactoring tool] was seeing the full picture of the changes. Since it was stepwise [and] I'm doing [each step] one by one, I'm not sure if each of the steps is going to be integrated correctly.

In practice, participants were more likely to refactor incorrectly using wizards.

Change Review Seven participants preferred the tool to inform them about the effects of an automated refactoring on their code. However, we observed that most participants did not inspect the previews of wizard-based refactorings. As a result, seven participants did not catch unintended changes of the wizard-based refactorings in their previews. Four participants mentioned that the previews were too cluttered. One participant said that previews are good for beginners who want to learn a new refactoring wizard. Our prior study showed that programmers rarely preview their refactorings in practice [24].

Multi-selections Four liked the wizard's ability to refactor multiple entities at the same time. During the study, five participants tried to use the Extract Superclass wizard to extract a common superclass from two classes at once. The rest either found it easier to refactor in smaller steps or did not notice the configuration option to extract from multiple classes.

Configuration Options and Error Messages Although the wizard-based refactorings provide many options to customize the refactorings, the participants only used a subset of these options. Because one of the methods that the participants had to move to the superclass referenced other members of the subclass, the refactoring wizard reported an error message. The refactoring wizard has an "Add Required" button that when pressed selects all members that are referenced by the currently selected members. However, none of the participants used this configuration option. Instead, they performed the refactoring and fixed the resulting compilation problem manually. One of the participants said that he ignored the error message of the wizard because it was not *actionable*:

> It [The refactoring wizard] came up with something like: "Sorry, this method is referring to this other variable that we can't change". I didn't know what I could do about that in the window. I was like: "OK. Thanks for the information!" [*laughter*]

These observations are consistent with the results of prior studies that showed programmers rarely configure the refactoring wizards [17] and usually apply an automated refactoring that has reported problems [24].

Composition Order Three participants mentioned that one has to be careful with the order in which she composes the refactorings. On the other hand, one participant indicated that sometimes significant work is required to transform the code to a state that is amenable to the application of a wizard-based refactoring.

6.6 Design Suggestions

The participants suggested improvements to the compositional and wizard-based paradigms. For the compositional paradigm, two participants proposed that the tool suggests the entities that the programmer might want to refactor next.

For the wizard-based paradigm, two participants suggested the ability to match up similar entities. For instance, the Extract Superclass refactoring could detect similar members in multiple classes, or let the programmer match up the related members and pull them up to the superclass in one step. In addition, one participant proposed that the refactoring wizard provides an *incremental preview*. An incremental preview gets updated as the programmer manipulates the configuration options. Finally, one participant suggested that the tool presents the previews graphically.

7 Limitations

Like any study, each of our prior field study [24], inference of refactoring sets (Section 2) and composition patterns (Section 3), survey (Section 5) and lab study (Section 6) has its own limitations. However, their results with respect to the effectiveness of compositional refactoring corroborate one another. The rest of this section discusses some of the limitations of our work.

Eclipse Foundation Data Set The Eclipse foundation data we used for mining frequent refactoring sets, while huge, lacks precision. For instance, it does not differentiate certain refactorings, e.g., Inline Local Variable and Inline Method. In addition, this data set does not include the project and workspace in which the refactoring is invoked. Moreover, it misses refactorings invoked through Quick Assist. Despite these limitations, the Eclipse foundation data serves as a good starting point to quantify the prevalence of frequent refactoring sets (Section 2).

Participants The Illinois data set, while more precise, comes from a smaller pool of participants. We found it challenging to recruit a larger group of experienced programmers due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. Nonetheless, our demographic survey shows that our pool of participants come from diverse backgrounds, have various levels of experiences, and work on a variety of nontrivial projects. Thus, we believe that our participants are representative of real-world programmers.

The majority of the lab study participants were very familiar with Eclipse. Further studies are needed to understand the effect of experience on the preferred paradigm of refactoring.

Qualitative Analysis While it is easy to draw conclusions from the qualitative responses of the participants, one must be cautious not to over-interpret. Therefore, we analyzed the survey responses in light of prior studies, analysis of refactoring usage data, and our design goals, and implemented the tool that we used in the lab study based on this holistic view.

Generalizability Due the constraints of the survey and lab studies, we evaluated the compositional paradigm using two refactorings, i.e., Extract Superclass and Extract Composite. We expect the compositional paradigm to generalize to other complex refactorings such as Pull Up, Push Down, Extract Interface, Extract Class, and Encapsulate Field. This is because these refactorings affect multiple files or change the class hierarchy in similar ways and can be represented as compositions of several refactorings.

Our data sets were limited to the use of the Eclipse refactoring tool for Java. However, we expect our results to hold for similar refactoring tools, because they follow a similar user interaction model, i.e., wizard-based refactoring.

Refactoring Tasks The survey participants did not try the tools and their evaluations were merely based on the screenshots of the wizard-based and compositional refactorings. However, the wizard-based and compositional refactorings are based on familiar features of Eclipse, i.e., wizards and Quick Assist. The insightful comments of the survey participants indicate that they understood the two paradigms well.

The survey study demonstrated the two paradigms of refactoring using a small piece of code. Several features of the wizard-based refactoring such as extracting from multiple classes, computing the required dependencies, and using the new superclass wherever possible were not applicable to that refactoring task. We intentionally kept the survey simple to make it understandable for programmers who may not be familiar with the intricacies of the wizard. Moreover, most configuration options of the wizard could be simulated by refactoring compositions. We consider some loss of *functionality* in the compositional design an acceptable trade-off for gaining *simplicity* and *naturalness*.

For the lab study, we selected a single realistic refactoring task that Kerievsky used in his book to introduce the Extract Composite refactoring [9, p. 214]. As some of our participants speculated, the wizard-based refactoring might be more appropriate when a refactoring is going to affect hundreds of files. Further studies are needed to compare the two paradigms of refactoring on a variety of refactoring tasks.

Participant Response Bias A common limitation of user studies is that participants may favor the interface that they think the researcher has developed. However, we think that our results are less affected by this bias, because most of our participants could not tell which interface was ours. At the end of the lab study, most participants asked us which interface was ours. This is because the

Extract Superclass wizard is rarely used [17,24], and few programmers remember all Quick Assist actions to identify the actions contributed by our plug-in.

8 Related Work

Composite Refactorings One paradigm of automating composite refactorings is to build new tools that execute a sequence of smaller refactorings atomically. Several researchers [3, 12, 21] have proposed methods for checking the behavior-preservation of a composite refactoring based on the pre and post conditions of its individual refactorings. Others [14,27] have introduced scripting languages for automating composite refactorings. We introduce a radically different paradigm for automating composite refactorings. Rather than building a monolithic tool from several refactorings, we propose that a large refactoring be decomposed into smaller refactorings. These two paradigms suit different needs. The monolithic paradigm is suited for toolsmiths who are in charge of applying a refactoring on a large code base in batch mode. The compositional paradigm is designed for interactive refactoring in an IDE. In addition, the monolithic paradigm aims to provide correctness guarantees by inferring preconditions. The compositional paradigm makes it easy for the programmers to verify the correctness of the refactoring by making each step easy to predict and verify.

Murphy-Hill et al. [17] showed that developers frequently invoke automated refactorings in batches, i.e., within 60 seconds of one another. Their work was limited to batches of repeated invocations of the same kinds of refactoring. Negara et al. [18] found that more than one third of manual and automated refactorings are performed in batches. Our study goes beyond reporting the frequencies of refactoring sets (Section 2) and sheds light on the rationales of composing automated refactorings (Section 3).

Schäfer et al. [23] argued that a very fine-grained decomposition of a refactoring into a composition of micro-refactorings over an extended language makes the implementation of the refactoring tool more reliable. They used Extract Method as an example to demonstrate their technique. While their focus was on reliability, ours was on usability. Generalizing their results, the compositional paradigm should lead to more reliable implementations of large refactorings.

Usability of Refactoring Tools Murphy et al. [15] reported the first results about the frequencies of invocations of Eclipse automated refactorings. Subsequent empirical studies [17, 24] showed that refactoring tools are underused. These results prompted research on improving the usability of refactoring tools.

Murphy-Hill and Black [16] developed a prototype tool that visualizes code selections and error messages. They evaluated their tool on a single refactoring, namely Extract Method. Similarly, we evaluated compositional refactoring on two refactorings, Extract Superclass and Extract Composite.

Lee et al. [13] showed that invoking refactorings through drag-and-drop gestures is more intuitive than menus and wizards. Alternative methods of invocations are complementary to our work, because they can streamline the invocation of the individual steps of a composite refactoring.

9 Conclusions

We feel a rush to more automation in the software engineering community, often through wishful thinking or superficial claims about the impact of additional automation on the productivity of programmers. Despite the push to automate more refactorings and other recurring program transformations, studies have shown that programmers greatly underuse such tools [16, 18, 24].

Rather than offering more automation, we took the opposite direction, and proposed the *compositional paradigm* for refactoring. In this paradigm, the tool automates the individual steps, and puts the programmer in control by letting her manually compose the steps into a complex change.

The compositional paradigm was inspired by our analysis of the refactoring practices of programmers in the wild. Our data mining and manual examination of two refactoring usage data sets provided evidence for the *prevalence*, *diversity*, *rationales*, and *naturalness* of composing automated refactorings. In addition, our survey and lab studies showed that the compositional paradigm is more *effective* than the existing wizard-based paradigm of refactoring.

The compositional paradigm outperforms the wizard-based one by reducing the automation level. Although this result may seem counterintuitive, it is not unique to software engineering. Designers of other fields, e.g., aviation, health-care, and manufacturing, struggle with similar problems. What is an appropriate level of automation? What should the role of the human operator be? Often, researchers find that *less is more*. That is, a modest design, which provides clear, immediate feedback, outperforms a design with a high level of automation that does not integrate the human operator well [2, 10, 28].

References

1. The R Project for Statistical Computing, `http://www.r-project.org/`
2. Bainbridge, L.: Ironies of Automation. Automatica (1983)
3. Cinnéide, M.Ó.: Automated Application of Design Patterns: A Refactoring Approach. Ph.D. thesis, Univ. of Dublin, Trinity College (2000)
4. Dig, D., Marrero, J., Ernst, M.D.: Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In: ICSE (2009)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
7. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, Univ. of Washington (1991)
8. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann, third edn. (2011)
9. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)

10. Kirlik, A.: Modeling Strategic Behavior in Human-Automation Interaction: Why an “Aid” Can (and Should) Go Unused. J. Human Factors and Ergonomics Soc. (1993)
11. Kjolstad, F., Dig, D., Acevedo, G., Snir, M.: Transformation for Class Immutability. In: ICSE (2011)
12. Kniesel, G., Koch, H.: Static Composition of Refactorings. Sci. Comput. Program. (2004)
13. Lee, Y.Y., Chen, N., Johnson, R.E.: Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In: ICSE (To Appear) (2013), `http://hdl.handle.net/2142/30011`
14. Li, H., Thompson, S.J.: A Domain-Specific Language for Scripting Refactorings in Erlang. In: FASE (2012)
15. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software (2006)
16. Murphy-Hill, E., Black, A.P.: Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In: ICSE (2008)
17. Murphy-Hill, E., Parnin, C., Black, A.P.: How We Refactor, and How We Know It. IEEE Trans. Software Eng. (2011)
18. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A Comparative Study of Manual and Automated Refactorings. In: Submission to ECOOP (2013)
19. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, Univ. of Illinois at Urbana-Champaign (1992)
20. Roberts, D., Brant, J., Johnson, R.: A Refactoring Tool for Smalltalk. Theor. Pract. Object Syst. (1997)
21. Roberts, D.B.: Practical Analysis for Refactoring. Ph.D. thesis, Univ. of Illinois at Urbana-Champaign (1999)
22. Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Refactoring Java Programs for Flexible Locking. In: ICSE (2011)
23. Schäfer, M., Verbaere, M., Ekman, T., Moor, O.: Stepping Stones over the Refactoring Rubicon. In: ECOOP (2009)
24. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, Disuse, and Misuse of Automated Refactorings. In: ICSE (2012)
25. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Zilouchian Moghaddam, R., Johnson, R.E.: The Need for Richer Refactoring Usage Data. In: PLATEAU (2011)
26. Vakilian, M., Dig, D., Bocchino, Jr., R.L., Overbey, J.L., Adve, V., Johnson, R.: Inferring Method Effect Summaries for Nested Heap Regions. In: ASE (2009)
27. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a Scripting Language for Refactoring. In: ICSE (2006)
28. Vicente, K.J.: Less is (sometimes) more in cognitive engineering: the role of automation technology in improving patient safety. Qual Saf Health Care (2003)
29. W. Ryan, G., Bernard, H.R.: Handbook of Qualitative Research, chap. Data Management and Analysis Methods. SAGE Publications, second edn. (2011)
30. Wloka, J., Sridharan, M., Tip, F.: Refactoring for Reentrancy. In: ESEC/FSE (2009)
31. Xing, Z., Stroulia, E.: Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study. In: ICSM (2006)