Scripting Refactorings in Java to Introduce Design Patterns

Jongwook Kim
University of Texas at Austin
jongwook@cs.utexas.edu

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

Abstract

Introducing design patterns into a program by hand is tedious and error-prone. Refactorings help but manual tasks still remain: you must understand available refactorings, determine a precise sequence of refactorings to invoke, and perform these tasks repetitively to a laborious degree.

We present *Reflective Refactoring ($\mathcal{R}^2$)*, a Java package to automate the introduction of classical design patterns (Visitor, Abstract Factory, *etc*), their inverses and variants. We encoded 78% of classical design patterns as $\mathcal{R}^2$ scripts. In one application, $\mathcal{R}^2$ automatically created a visitor with 276 `visit` methods by invoking 554 Eclipse refactorings in a few minutes – an achievement that could not be done manually. We demonstrate the generality and scalability of $\mathcal{R}^2$, illustrate its productivity potential, and explain why refactoring speed and correctness are critical issues for scripting in next-generation refactoring engines.

1. Introduction

Modern IDEs – Eclipse, NetBeans, JDeveloper, IntelliJ IDEA, and Visual Studio – mostly offer primitive refactorings (*eg* `rename`, `move`, `change method signature`) that constitute the basic steps to introduce design patterns, not patterns themselves. It is surprising that they do not permit the scripting of transformations (or refactorings) that introduce whole patterns, especially as it has been 20 years since design patterns were introduced [15], longer still for refactorings [17, 29, 30], and for at least 15 years it was known that many design patterns could be automated by scripting transformations [21, 39].

A Visitor pattern, for example, could be scripted and thus created automatically by identifying a method (called a "seed") that is to be moved into a visitor. (All methods in a class hierarchy that have the same return type, name, and argument signature of the seed are also moved into a newly created visitor class). While the steps to introduce this pattern are not complex and have been well documented [21], manually executing them is extraordinarily error-prone and laborious. In one of our experiments, a seed method for a Visitor entails creating a class with 276 `visit` methods by invoking 554 Eclipse refactorings. It is hard to imagine that anyone could manually execute such a change.

We teach undergraduate and graduate courses on software design. Among the best ways to learn refactorings and patterns is not only to ***use*** them, but also to ***write*** programs that sequence transformations to mechanize patterns. Doing so forces students, and programmers in general, to understand the nuances and capabilities of each refactoring. Although we are primarily motivated to improve tools for teaching refactorings and patterns, we strongly believe our work will benefit professional programmers as well.

The key question is: what language should be used to script refactorings? There are many proposals with distinguished merit [3, 5, 6, 8, 9, 18, 23, 25, 36, 42, 44], but all fall short in fundamental ways ***for our goal***. Undergraduates are novices to Java; it is unrealistic to expect them to learn yet another language (functional or otherwise) or to become proficient in sophisticated *Program Transformation Systems (PTSs)* [5–8] or utilities, such as the *Eclipse Language Toolkit (LTK)* [14], to manipulate programs. Although PTSs and LTK are monuments of engineering prowess, their learning curve is measured in weeks or months.

We present a practical way to move Java refactoring technology forward in this paper. *Reflective Refactoring ($\mathcal{R}^2$)* is a Java package whose goal is to encode the construction of classical design patterns as simple methods. Using *Eclipse Java Development Tools (JDT)* [12], $\mathcal{R}^2$ leverages reflection by presenting an Eclipse project, its packages, classes, methods, fields, *etc*, as Java objects whose methods are transformations and JDT refactorings. Automating design patterns becomes no different than importing an existing Java package ($\mathcal{R}^2$) and using it to write programs (in this case, refactoring scripts). There is no need for a DSL.

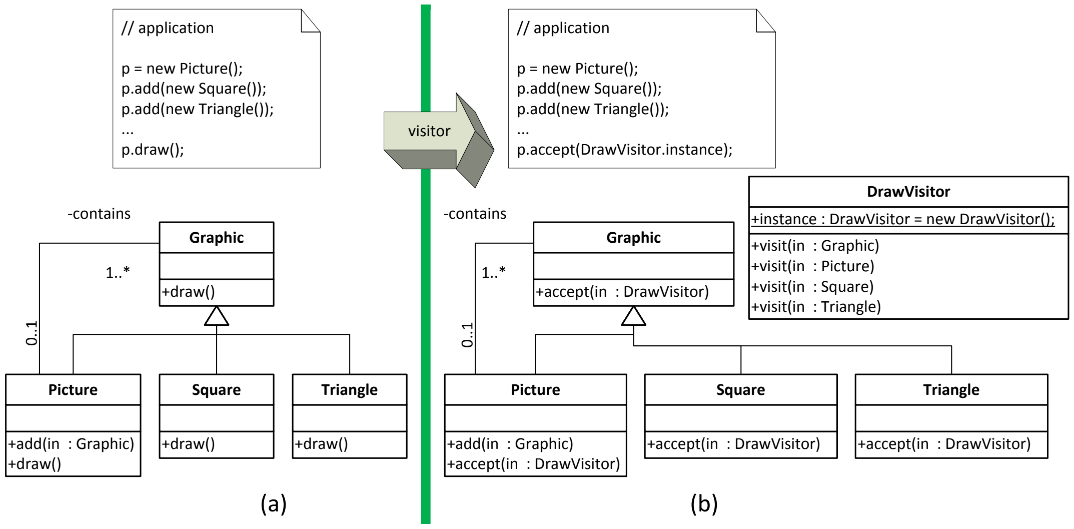


Figure 1. A Visitor Pattern Refactoring.

Our paper makes the following contributions:

- **Design.** We present a reflection-like API, $\mathcal{R}^2$, to script refactorings to construct design patterns in Java.
- **Generality.** We encoded 78% of the Gang-of-Four design patterns and their inverses as short Java methods in $\mathcal{R}^2$.
- **Productivity.** We asked 44 students (grads or 3rd-year undergrads) to introduce a small visitor of 7 methods into a program using Eclipse refactorings. $\mathcal{R}^2$ accomplished the task in 13 seconds; the average completion time for students was well over an hour and 47% of them required help to do so.
- **Evaluation.** We used $\mathcal{R}^2$ to retrofit 37 pattern instances into 6 real-world applications. To add a Visitor in one case invoked 554 Eclipse refactorings; to undo the Visitor coincidentally also used 554 Eclipse refactorings.

2. A Motivating Example

Consider the Visitor pattern. There are different ways to encode a Visitor; we use the one below. Figure 1a shows a hierarchy of graphics classes; `Graphic` is the superclass and `Picture`, `Square`, `Triangle` are its subclasses. Each class has its own distinct `draw` method.

A "mechanics" script in a refactoring text spells out the procedure to install a Visitor into a program [21]. Look at how tedious it is to create a visitor for the `draw` method (Figure 1b): First, create a singleton visitor class (`DrawVisitor`). Next, each `draw` method has to be moved into the `DrawVisitor` class, renamed to `visit`, and an extra parameter (namely the class from which the method was moved) is added. Referenced non-public fields and methods must become `public` after a method `move`. Further, a delegate (named `accept`) must be created for each moved method, taking its place in the original class. The signature of the `accept` method extends the original `draw` signature with a `DrawVisitor` parameter and whose code is (for our example):

```
void accept(DrawVisitor v) {
  v.visit(this);
}
```

And finally, all calls to the original method, `o.draw()`, are replaced with `o.accept(DrawVisitor.instance)`.

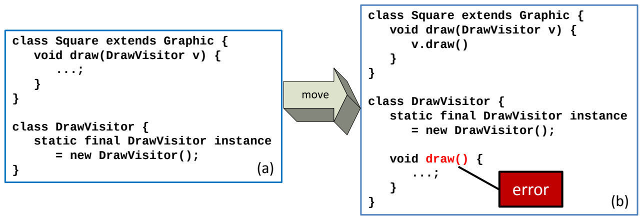
Individual steps can be performed by JDT refactorings, but this requires knowledge and familiarity with the available refactorings to know which to use and in what order.

After each step, the program is recompiled and regression tests are run to ensure that the refactored program was not corrupted. It is easy to make a mistake or forget a step. A programmer can loose track of the overall process, inadvertently skipping `draw` methods to move. One would think that creating a visitor with *only 7 methods* using JDT refactorings is simple, yet as we see in Section 6.2.1, programmers may abandon, may request help, or may not attempt this task. Refactoring scripts eliminate these problems.

Here is another complicating issue: *Eclipse refactorings were never designed with scripting in mind*. We encountered a series of design and implementation issues in Eclipse JDT (version 4.2.2) [13] that compromises its ability to support refactoring scripts without considerable work. (We think these issues need to be addressed, regardless of our work). Here are some examples.

2.1 Separation of Concerns

Figure 2a shows method `draw` in class `Square`, after a `DrawVisitor` parameter was added. Figure 2b shows the result of Eclipse moving `Square.draw` to `DrawVisitor.draw` and leaving a delegate behind. Not only was the method moved, *its signature was also optimized*. Eclipse realizes



```
class Square extends Graphic {
   void draw(DrawVisitor v) {
      ...;
   }
}

class DrawVisitor {
   static final DrawVisitor instance
      = new DrawVisitor();
}                                        (a)
```

```
class Square extends Graphic {
   void draw(DrawVisitor v) {
      v.draw()
   }
}

class DrawVisitor {
   static final DrawVisitor instance
      = new DrawVisitor();

   void draw() {
      ...;
   }
}                                        (b)
```

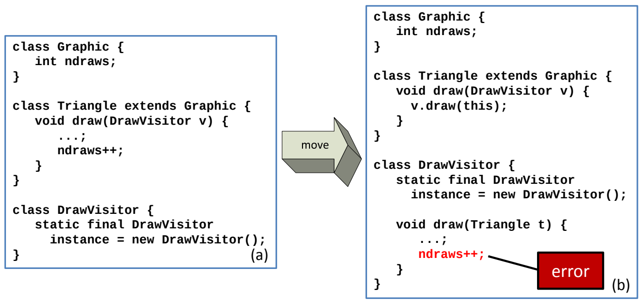
Figure 2. An Eclipse Refactoring Being Too Smart.

that the original `draw` method did not need its `Square` parameter, so Eclipse simply removes it.

As a refactoring, this optimization is *not* an error. But when an entire set of refactorings must produce a coherent result, it *is* an error. Preserving all parameters of moved methods in a Visitor pattern is essential. This is an example where two concerns – method movement and method signature optimization – were bundled into a single refactoring, instead of being separated into distinct refactorings. We were able to deactivate method signature optimizations in our Eclipse plug-in; we do not know how to disable this optimization from the Eclipse GUI.

2.2 Bugs

Figure 3a shows method `Triangle.draw` that increments field `ndraws`. When Eclipse moves `Triangle.draw` to `DrawVisitor`, the code of Figure 3b is produced, which is incorrect. The correct increment should be `t.ndraws++`.



```
class Graphic {
   int ndraws;
}

class Triangle extends Graphic {
   void draw(DrawVisitor v) {
      ...;
      ndraws++;
   }
}

class DrawVisitor {
   static final DrawVisitor
     instance = new DrawVisitor();
}                                   (a)
```

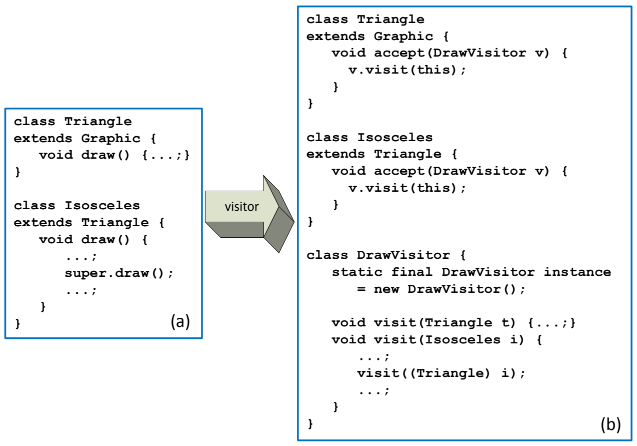
```
class Graphic {
   int ndraws;
}

class Triangle extends Graphic {
   void draw(DrawVisitor v) {
     v.draw(this);
   }
}

class DrawVisitor {
   static final DrawVisitor
     instance = new DrawVisitor();

   void draw(Triangle t) {
      ...;
      ndraws++;
   }
}                                   (b)
```

Figure 3. Bugs in Eclipse Refactorings.

We were unable to repair this refactoring in our Eclipse plug-in and reported it and other bugs that we found (see Table 5 in Section 6). For these situations, we repaired the transformed source code by hand.

2.3 Limited Scope

A benefit of Visitor is that a single visitor class enables a programmer to quickly review all variants of a method. Often, such methods invoke the corresponding method of their superclass. Our `graphic` class hierarchy could extend subclass `Triangle` with `Isosceles`, where `Isosceles.draw` invokes `Triangle.draw` (Figure 4a). A visitor implementation is shown in Figure 4b. Moving methods with `super` calls is not only possible, it is desirable.



```
class Triangle
extends Graphic {
   void draw() {...;}
}

class Isosceles
extends Triangle {
   void draw() {
      ...;
      super.draw();
      ...;
   }
}                    (a)
```

```
class Triangle
extends Graphic {
   void accept(DrawVisitor v) {
     v.visit(this);
   }
}

class Isosceles
extends Triangle {
   void accept(DrawVisitor v) {
     v.visit(this);
   }
}

class DrawVisitor {
   static final DrawVisitor instance
      = new DrawVisitor();

   void visit(Triangle t) {...;}
   void visit(Isosceles i) {
      ...;
      visit((Triangle) i);
      ...;
   }
}                                     (b)
```

Figure 4. Visitors with `super` References.

Eclipse refuses to move methods that reference `super`. It is not an error, but a strong limitation. In our Eclipse plug-in, this limitation is removed (see Section 5.1).

3. Reflective Refactoring

A key decision for us was choosing the scripting language. As refactorings are transformations, our initial inclination was to define and script refactorings in a functional or dedicated language, as others have done [3, 5, 6, 8, 9, 18, 23, 36, 42, 44]. But as we said earlier, the learning curve to become proficient in yet another language makes these approaches unappealing. The obvious answer is to script refactorings in Java.

Let $\mathcal{P}$ denote the current program or project of Eclipse. We leverage reflection — we define class `RClass` whose instances are the classes in $\mathcal{P}$; we define classes `RMethod` and `RField` whose instances are the methods and fields of $\mathcal{P}$, and so on. When $\mathcal{P}$ is compiled by Eclipse, a set of tables (one for `RClass`, `RMethod`, `RField`, *etc*) is created, where each row corresponds to a class, a method, or a field instance of $\mathcal{P}$. *These tables are not persistent; they exist only when the Eclipse project for* $\mathcal{P}$ *is open.*

The fields of `RClass`, `RMethod`, and `RField` — henceforth called $\mathcal{R}^2$ classes — also define the association and inheritance relationships among table rows (`A` is a superclass of `B`, `foo` is a method of `A`, *etc*). The methods of $\mathcal{R}^2$ classes expose primitive Eclipse refactorings, simple transformations, or composite refactorings (our scripts).

The usage scenario for all $\mathcal{R}^2$ refactorings is this: a user (student, programmer) points to a field, method, or class in the Eclipse GUI (using our plug-in) and invokes an $\mathcal{R}^2$ refactoring/script just like a native Eclipse refactoring. Let's look at a few $\mathcal{R}^2$ methods to get a feel for what is needed to write an $\mathcal{R}^2$ script.

3.1 Automating the Visitor Pattern

Figure 5 shows our `makeVisitor`, a method of class `RMethod`. The Java keyword "`this`" in Figure 5 denotes the "seed" method to which our script is applied. Again, this seed defines the set of methods (with the same name, argument signature, and return type) that are to be moved into a visitor.

Lines 5–7 create a visitor class (called *visitorClassName*) in the same package as `this` and add a static singleton field `instance`. Lines 9–11 find the superset of all methods with the same signature as `this` and add a new parameter of type *visitorClassName* to each of these methods. Calls to these methods have *visitorClassName*.`instance` as the default extra argument. Lines 13–19 move each movable method to the visitor class, leaving behind a delegate, and each moved method is renamed to `visit`. Lines 21–23 collect all delegate methods and methods that were not moved, and rename them to `accept`. All references to moved or renamed methods are automatically revised. Line 25 returns the visitor class.

```
 1  // member of RMethod class
 2  void RClass makeVisitor(String visitorClassName)
 3    throws RException
 4  {
 5    RPackage pkg = this.getPackage();
 6    RClass vc = pkg.newClass(visitorClassName);
 7    RField singleton = vc.addSingleton();
 8
 9    RMethodList methodList = this.getRelatives();
10    RParameter newPara =
11      methodList.addParameter(vc, singleton);
12
13    RMethod delegate = null;
14    for(RMethod m : methodList) {
15      if(!m.isMovable())
16        continue;
17      delegate = m.moveAndDelegate(newPara);
18      m.rename("visit");
19     }
20
21    RMethodList delegateList =
22      delegate.getRelatives();
23    delegateList.rename("accept");
24
25    return vc;
26  }
```

Figure 5. A `makeVisitor` Method.

Looping through a list of methods and invoking `addParameter` on each method would be the obvious way to code a Visitor pattern. But this is not how the Eclipse `change method signature` refactoring works (Lines 9–11). It is applied to a "seed" method; *all* methods with the same signature as the seed in an interconnected interface *and* class hierarchy are modified. Consider Figure 6. Suppose `D.m` is the method that "seeds" a `change method signature`. All `m` methods in `D`'s class hierarchy {`A.m`, `B.m`, `C.m`, `D.m`} *and* interconnected interface and class hierarchies {`I1.m`, `I2.m`, `E.m`} are affected by this refactoring. That is, all of these methods will have their signature changed.

The `methodList` variable in Line 9 is the list of all methods in $\mathcal{P}$ whose signature will change. This list includes methods that cannot be moved, such as interface and abstract methods. In this example, the methods moved into the visitor are from classes {`A`, `B`, `C`, `D`, `E`}.

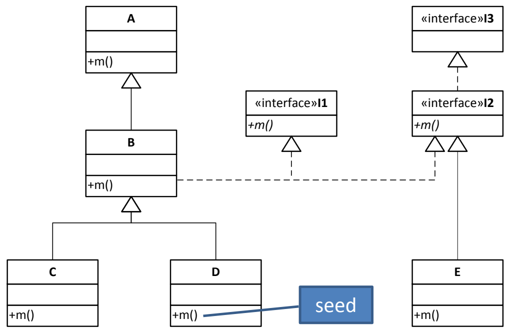


Figure 6. Methods Altered by Change Signature.

Our Eclipse plug-in allows $\mathcal{R}^2$ scripts to be invoked from the Eclipse GUI, just like native JDT refactorings. Alternatively, an $\mathcal{R}^2$ method could be invoked programmatically from another $\mathcal{R}^2$ method which is part of a larger, automated refactoring. We show below how `makeVisitor` is used in an example of Visitor pattern script where method `C.m()` in package `p` of project `R` is the seed:

```
RPackage p = RProject.getPackage("R", "p");
RClass   c = p.getClass("p.C");
RMethod  m = c.getMethod("void", "m", null);
m.makeVisitor("Visitor");
```

3.2 Automating the Inverse Visitor

Figure 7 depicts a common scenario. An $\mathcal{R}^2$ programmer creates a visitor to provide a convenient view that allows him/her to inspect all `draw` methods in the `graphic` class hierarchy. The programmer then updates the program, including visitor methods, as part of some debugging or functionality-enhancement process. At which point, he/she wants to remove the visitor to return the program back to its original structure.[1]

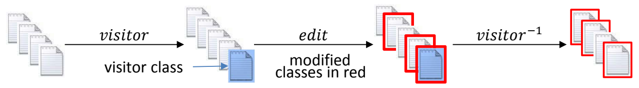


Figure 7. A Common Programming Scenario.

In this scenario, undoing a visitor *is not a roll-back, as a roll-back would undo all of the programmer's debugging edits*. Instead an inverse visitor – a refactoring that removes a visitor and preserves debugging edits – is required.

Figure 8 shows our `inverseVisitor`, a method of `RClass`, that moves `visit` methods back to their original classes and deletes the visitor class. Here's how `inverseVisitor` works: We want each `visit` method to be moved

[1] Of course for this to be possible, certain structures and naming conventions (as we use in our `makeVisitor` method) should *not* be altered. Effectively the only edits that are permitted are those that would have modified the original program. Restricting modifications can be accomplished similar to GUI-based editors, where generated code is "greyed" out and cannot be changed.

```
1   // member of RClass class
2   void inverseVisitor(String originalName)
3     throws RException
4   {
5     RMethod anyDelegate = null;
6
7     for(RMethod m : this.getMethodList()) {
8       anyDelegate = m.getDelegate();
9
10      RParameter para = m.getParameter(0);
11      RClass returnToClass = para.getClass();
12
13      m.move(returnToClass);
14
15      m.inlineSuperDelegate();
16      m.inline();
17    }
18
19    RMethodList methodList =
20      anyDelegate.getRelatives();
21    methodList.removeParameter(0);
22    methodList.rename(originalName);
23
24    this.delete();
25  }
```

Figure 8. An `inverseVisitor` Method.

back to its original class. Lines 10–11 recover the original class of a `visit` method. As we turned off method signature optimization in Section 2.1, the original class is encoded as the type of the `visit` method's first parameter. Line 13 moves the method back to its original class. Lines 15–16 inline super-delegates if they exist by replacing each call to `super_x`θ with a call to `super.x` (Section 5.1) and then restore the original method body as the body of the `accept` method by inlining.

Lines 8–16 are performed for all `visit` methods. At this point, the `accept` methods (*ie* the delegate methods) contain the body of the original methods.

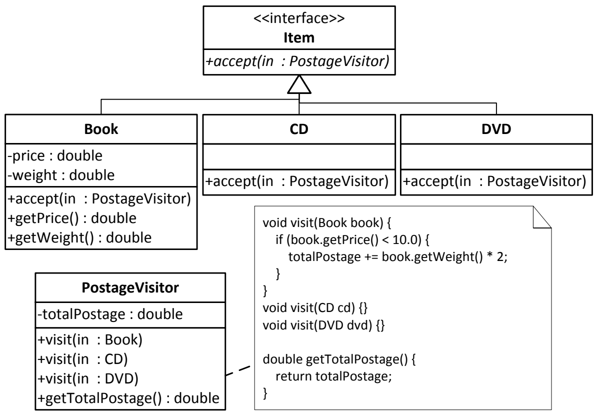
Lines 19–22 collect all of the `accept` methods, remove the first parameter (of type visitor class) from them, and restore the original name of the method. The visitor class is then deleted in Line 24.

> **Note:** Because our visitor and inverse visitor used unique names to avoid name capture, multiple instances of the Visitor refactoring can be applied to the same class and its inversion can be applied without problems.

Another practical reason to have an inverse visitor is if a program already contained a hand-crafted visitor, and the programmer wanted to weave its methods back into the class hierarchy, say for efficiency reasons [27]. We address this case in a broader setting in the next section.

3.3 More Opportunities

It is well-known that there are many variations of design patterns. Visitor is no exception. Consider the `PostageVisitor` of Figure 9 adapted from [37]. It differs from the visitor of our Motivating Example (Section 2) in several ways: `PostageVisitor` is not a singleton, it includes state `totalPostage`, it has a custom method `getTotalPostage()` – a



```
void visit(Book book) {
   if (book.getPrice() < 10.0) {
      totalPostage += book.getWeight() * 2;
   }
}
void visit(CD cd) {}
void visit(DVD dvd) {}

double getTotalPostage() {
   return totalPostage;
}
```

Figure 9. Visitor with State.

```
// member of RClass class
void inverseVisitorWithState(String originalName,
       String visitMethodName)
  throws RException
{
  RMethod anyDelegate = null;

  for(RMethod m : this.getMethodList(
       visitMethodName)) {
    anyDelegate = m.getDelegate();

    RParameter para = m.getParameter(0);
    RClass returnToClass = para.getClass();

    m.move(returnToClass);

    m.inlineSuperDelegate();
    m.inline();
  }

  RMethodList methodList = anyDelegate.
       getRelatives();
  methodList.rename(originalName);
}
```

Figure 10. Another `inverseVisitor` variant.

non-`visit` method, and at least one of its `visit` methods `visit(Book)` references `totalPostage`.

To create this visitor (from a program that does not have a visitor) requires a slight modification of our $\mathcal{R}^2$ `makeVisitor` method. To 'undo' this visitor – *ie* to move its `visit` methods back into the class hierarchy – requires a corresponding change to `inverseVisitor`. Figure 10 shows this customized $\mathcal{R}^2$ `inverseVisitorWithState` method. It differs from Figure 8 by moving only `visit` methods, not removing the visitor parameter, and not deleting the visitor class.

This illustrates how $\mathcal{R}^2$ can handle design pattern variants – either we can generate these patterns (by transforming a program without these patterns into programs with these patterns) or we can remove these patterns (by transforming programs with hand-crafted patterns into programs without those patterns). $\mathcal{R}^2$ offers a practical way to cover all of these possibilities.

4. Other Patterns

Table 1 summarizes our review of the Gang-of-Four Design Patterns text [15]. We found 35% of its patterns are fully automatable, 43% are partially automatable, and for the remaining 22%, we are unsure of their role in a refactoring tool (although some *are* automatable). We elaborate these findings in the next sections.

Design Pattern	Automation Possibility		
	Full	Some	Unsure
Abstract Factory	✓		
Adapter		✓	
Bridge		✓	
Builder	✓		
Chain of Responsibility		✓	
Command	✓		
Composite		✓	
Decorator		✓	
Façade			✓
Factory Method	✓		
Flyweight	✓		
Interpreter			✓
Iterator			✓
Mediator			✓
Memento	✓		
Observer		✓	
Prototype		✓	
Proxy		✓	
Singleton	✓		
State			✓
Strategy		✓	
Template Method		✓	
Visitor	✓		
Percentage	35%	43%	22%

Table 1. Automation Potential of Design Patterns.

4.1 Fully Automatable Patterns

The Visitor pattern, its inverse and variants are fully automatable as they produce no "`TO DO`s" for a user. Appendix A sketches other fully automatable patterns as $\mathcal{R}^2$ methods: abstract factory, command, and memento. 35% of patterns are in this category.

4.2 Partially Automatable Patterns

43% of patterns are partially automatable – the creation of a pattern produces "`TO DO`s" that must be completed by a user. The Adapter pattern, below, is typical. Appendix B sketches another partially automatable pattern: Strategy.

The Adapter pattern resolves incompatibilities between a client interface and a legacy class. For example, given interface `Target` and class `Legacy` in Figure 11, an intermediate class (called `Adapter`) adapts `Target` to `Client`.

The `makeAdapter` $\mathcal{R}^2$ method in Figure 12 creates an `Adapter` class that implements interface `Target` and references class `Legacy`. Programmers must provide bodies for the generated method stubs – these are the user "`TO DO`s". Although partially automated – method bodies are still needed – tedious and error-prone work is done by $\mathcal{R}^2$.[2]

[2] For readers who are unconvinced that creating an adapter is not tedious and error prone, try the following example: access Java `String` through the `collection` $<$ `Character` $>$ interface.

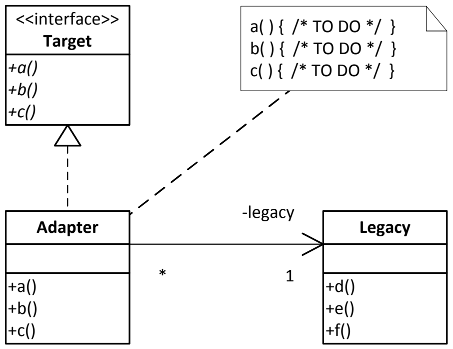


Figure 11. Adapter Pattern.

```
// member of RInterface class
RClass makeAdapter(String adapterName,
                   RClass adaptee) {
  RClass c = getRPackage().newClass(adapterName);

  RField f = c.newField(adaptee);
  c.newConstructor(f);

  for(RMethod m : getAllMethod()) {
    c.newMethod(m);
  }

  c.setInterface(this);

  return c;
}
```

Figure 12. A `makeAdapter` Method.

4.3 Remaining Patterns

We are unsure of the role for the remaining 22% in a refactoring tool (some of which we note below *are* automatable). Consider:

- Façade is a convenient class abstraction for a package. Creating a façade requires deep knowledge of an application that only an expert, not a refactoring tool, would have. An $\mathcal{R}^2$ script *could* be written to produce a particular façade, but it would be so application-specific it would not be reusable.
- Interpreter is common in compiler-compiler tools [4, 31]; given the grammar of a language, a class hierarchy for the language's AST can be generated. Providing a grammar to a refactoring engine to generate a class hierarchy is possible, but we are unsure that it is consistent with current refactoring tools.
- State is a simple application of *Model Driven Engineering (MDE)*. Given a statechart of a finite state machine, MDE tools can generate the class hierarchies and method bodies that implement the State pattern. Again, providing a statechart to a refactoring engine to generate the code of a State pattern is possible, but we are unsure that it is appropriate.
- Mediator is the basis for GUI builders; the drag-and-drop of class instances from a pallete of classes is the essence

of a mediator. Again, it is unclear that this functionality belongs to a refactoring engine.

- Iterator is already part of the Java language. It is unclear what a refactoring engine should do.

5. Current Implementation

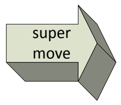
Implementing $\mathcal{R}^2$ was very challenging. We found Eclipse source to be highly tangled, there was little or no useful documentation, and it was particularly difficult for us to understand how to invoke refactorings programmatically [22].

In the end, we used existing refactoring GUI dialogs to trigger refactorings. Doing so, we could automate exactly the same procedures Eclipse users would follow manually, allowing us to measure the actual time spent on overall refactoring process *and* accurately estimate the overhead of JDT refactorings due to GUI operations. Overall, we changed 51 lines in 8 Eclipse internal packages; the $\mathcal{R}^2$ package consists of $\sim$ 5K LOC.

5.1 Generalizing Eclipse `move`

We said in Section 2.3 that Eclipse does not move methods that reference `super`. Our `makeVisitor` of Figure 5 is generic in that it does not impose `super` constraints on the methods that it moves; rather it is the Eclipse `move instance method` refactoring that imposes the `super` constraint. The fix is shown in Figure 13, where method `B.foo()` is moved to class `C`. Prior to invoking `move`, we replace each method call `super.x()` with a call to a manufactured method `super_x`θ`()`, whose body calls `super.x()`; θ is simply a random number to make the name of the manufactured method unique.[3,4] The revised method no longer invokes `super`, and thus *can* be moved by Eclipse.

```
class B extends A {
   void foo() {
      ...;
      super.bar();
      ...;
   }
}

class C {}                    (a)
```



```
class B extends A {
   void super_barθ() {
      super.bar();
   }
}

class C {
   void foo(B b) {
      ...;
      b.super_barθ();
      ...;
   }
 }                            (b)
```

Figure 13. Move Method With `super` Call.

Here is our case analysis for this revision. The first case is when `super` keywords are present but are not needed. Figure 14a shows a program with superclass `A` and its subclass `B`. The body of method `B.bar` has the `super` keyword. Suppose we want to move `B.bar` to `Visitor.bar` and make the Visitor pattern in Figure 14b. Because subclass `B` does not override `A.foo`, the expression `super.foo()` can be replaced by `foo()`, at which point the body of `B.bar` has no reference to `super` and can be moved directly by Eclipse.

```
class A {
   void foo() {}
}

class B extends A {
   void bar() {
      super.foo();
   }
}
                              (a)
```

```
class A {
   void foo() {}
}

class B extends A {
   void accept(Visitor v) {
      v.visit(this);
   }
}

class Visitor {
   static final Visitor instance
         = new Visitor();
   void visit(B b) {
      b.foo();
   }
}                             (b)
```

Figure 14. No Need For `super`.

The next case is when `super` keywords are needed due to method overriding. In Figure 15a, the `super` keyword is used to invoke an overridden method `A.foo()`. We remove `super` by calling a delegate method which calls the overridden method `A.foo()`. Figure 15b shows a super delegate `super_foo`θ which replaces the `super.foo()` call in `B.bar`, thus allowing Eclipse to move `B.bar()` to the visitor class. Of course, `super`-delegates throw the same exception types as its `super` invocation.

```
class A {
   void foo() {}
}

class B extends A {
   void foo() {}
   void bar() {
      super.foo();
   }
}
                              (a)
```

```
class A {
   void foo() {}
}

class B extends A {
   void foo() {}
   void accept(Visitor v) {
      v.visit(this);
   }

   void super_fooθ() {
      super.foo();
   }
}

class Visitor {
   static final Visitor instance
         = new Visitor();
   void visit(B b) {
      b.super_fooθ();
   }
}                             (b)
```

Figure 15. Rewrite that Uses super Delegate.

Lastly, suppose an overriding method `B.foo()` uses `super` to reference its overridden method `A.foo()` (see Figure 16a). As a Visitor pattern moves both methods, a `super` delegate is not needed since the same effect of calling a `super` method can be achieved by type casting. However, `super`-delegates are required in the general case.

[3] If `super.x` returns a result of type `X`, `super_x`θ returns type `X`.

[4] A unique name is needed for a refactoring that "undoes" or "removes" a visitor (Section 3.2). It also guarantees the correct `super`-delegate is called, as the meaning of `this` and `super` depends on the position in a class hierarchy from which it is invoked.

```
class A {
   void foo() {}
}

class B extends A {
   void foo() {
      super.foo();
   }
}
```

(a)

```
class A {
   void accept(Visitor v) {
      v.visit(this);
   }
}

class B extends A {
   void accept(Visitor v) {
      v.visit(this);
   }
}

class Visitor {
   static final Visitor instance
            = new Visitor();
   void visit(A a) {}
   void visit(B b) {
      visit((A)b);
   }
}
```

(b)

Figure 16. Rewrite That Uses Type Casting.

5.2 Field Access using `super`

Now consider the use of `super` to reference field variables. Unlike method overriding, fields in Java are hidden and not overridden. So we can get `super` references simply by casting to their declared type. In Figure 17, method `B.foo` references field `A.i` with the expression `super.i`. When `B.foo` is moved to class `Visitor`, expression `super.i` is replaced with `((A)b).i`.

```
class A {
   int i;
}

class B extends A {
   int i;
   void foo() {
      super.i = 0;
   }
}
```

(a)

```
class A {
   int i;
}

class B extends A {
   int i;
   void accept(Visitor v) {
      v.visit(this);
   }
}

class Visitor {
   static final Visitor instance
            = new Visitor();
   void visit(B b) {
      ((A)b).i = 0;
   }
}
```

(b)

Figure 17. `super` Field Access.

6. Evaluation

To evaluate the usefulness of $\mathcal{R}^2$ we answer three research questions:

- **RQ1:** Does $\mathcal{R}^2$ improve productivity?
- **RQ2:** Is $\mathcal{R}^2$ scalable? Can it be applied to large programs?
- **RQ3:** Is $\mathcal{R}^2$ safer than manual changes?

These questions address the higher level question "Is $\mathcal{R}^2$ useful?" from different angles. Productivity measures whether automation saves programmer time. Scalability measures whether $\mathcal{R}^2$ can work with large programs. Safety ensures that runtime behavior is not modified by $\mathcal{R}^2$.

6.1 Experimental setup

We combine two methods, a user study and a case study, that complement each other. The user study allows us to quantify programmer time and programmer errors, while the case study gives us confidence that $\mathcal{R}^2$ is generalizable to real-world situations.

User Study. We conducted a user study with graduate and undergraduate classes in software design at UTCS in Fall 2013. Both classes had over 20 students each. Overall, a total of 44 students participated. Our participants had on average 5.4 years of programming experience, and 4.1 years of programming in Java. 64% of them were already very familiar with Eclipse. Most students self-organized in teams of 2-3 but 8 students worked individually.

We explained the Visitor pattern to both classes (pointing to standard web site videos and web explanations, too). We gave the code base of `Q` (from Table 2), a Java GUI application that computed module expressions for different program generation approaches, along with its regression tests, and told each team to work for up to one hour to create a visitor. If after an hour they did not finish, or if they gave up, we would give them a "mechanics script" for them to follow to produce the visitor. As the visitor required only 7 methods to be moved (those methods have simple bodies with few references (11)), we felt confident that most would not request help.

To carry out the experiment, among our 44 participants, 33 used Eclipse JDT refactorings to introduce the Visitor pattern, and 11 manually introduced the pattern outside of the Eclipse IDE.

Case Study. The greatest challenge for us was implementing the Visitor and Inverse Visitor patterns: these $\mathcal{R}^2$ methods stressed Eclipse, $\mathcal{R}^2$ capabilities, and exposed the core challenges for future work. The other patterns (discussed in Section 4 and the Appendixes) did not require complex JDT refactorings – they mostly added new program elements or class relationships.

We used six real-world, publicly available Java applications. We selected applications with three criteria in mind: (1) they had non-trivial class hierarchies, (2) regression tests were available for us to determine if our refactorings altered application behavior, and (3) there were numerous method candidates that could "seed" a visitor. We randomly selected methods among these candidates. We believe this selection process presents both a representative set of applications and a fair test for our work.

The Subject column of Table 2 lists these applications, their versions, application size in lines of code, and the

Seed ID	Subject (Ver#, LOC, #Tests)	Seed Method Name	Super Del	Chg Sig	Move	Ren	# of Refact	Elap Time	Esti Time	# of Errors	Bug IDs
A1		getAST_Exp	0	26	26	52	104	70s	23s	26	B2
A2	**A**HEAD	getExpression	0	17	17	34	68	52s	22s	17	B2
A3	jak2java [4]	printorder	0	1	276	277	554	547s	298s	0	-
A4	(130320, 26K, 75)	reduce2Ast	1	1	29	30	60	45s	18s	23	B2
A5		reduce2Java	7	1	47	48	96	83s	40s	100	B2
C1		encode	0	1	2	3	6	5s	2s	27	B2, B4
C2	Commons	getCharset	0	4	4	8	16	13s	6s	0	-
C3	**C**odec [1]	getDefaultCharset	0	4	4	8	16	12s	6s	0	-
C4	(1.8, 16K, 6103)	getEncoding	0	2	4	6	12	11s	6s	3	B4
C5		isInAlphabet	0	1	2	3	6	4s	2s	2	B2
I1		getDefaultEncoding	0	1	1	2	4	4s	2s	0	-
I2	Commons	getEncoding	0	1	1	2	4	5s	3s	0	-
I3	**I**O [2]	getFileFilters	0	1	2	3	6	5s	3s	0	-
I4	(2.4, 24K, 810)	getSize	0	1	1	2	4	8s	6s	0	-
I5		setFileFilters	0	1	2	3	6	5s	2s	0	-
J1		countTestCases	1	1	7	8	16	13s	6s	1	B4
J2	**J**Unit [19]	failedTest	0	1	1	2	4	6s	1s	0	-
J3	(4.11, 23K, 2807)	getName	0	4	5	9	18	109s	101s	2	B2, B4
J4		run	2	1	9	10	20	22s	10s	4	B4, B6
J5		testCount	0	1	1	2	4	4s	2s	2	B1
Q	**Q**uark [4] (1.0, 575, 9)	apply	0	1	7	8	16	13s	6s	0	-
R1		computeLikeliness	0	1	13	14	28	21s	10s	14	B2, B4
R2	**R**efactoring	extractFullyQualifie...	0	1	1	2	4	4s	1s	0	-
R3	Crawler [10]	isRename	0	1	12	13	26	25s	15s	10	B4
R4	(1.0.0, 7K, 15)	pruneFalsePositives	1	1	4	5	10	8s	4s	1	B4
R5		pruneOriginalCandid...	7	1	13	14	28	23s	12s	4	B4

Table 2. Applications and Visitor Pattern Results.

number of regression tests. We used an Intel CPU i7-2600 3.40GHz, 16 GB main memory, Windows 7 64-bit OS.

To determine whether the subjects were unchanged in terms of runtime behavior after applying an $\mathcal{R}^2$ refactoring, we ran all available regression tests included with each application before and after refactoring.

6.2 Results

6.2.1 Does $\mathcal{R}^2$ Improve Productivity?

The results of the user study are shown in Table 3. Again, we asked teams of students to create a visitor using JDT refactorings. The two rows of Table 3 partition these teams into two groups – those that followed directions (the top row) and those that didn't and created the requested visitor by hand.

Individuals who used JDT refactorings took 107 minutes to complete the refactoring; 60% of them requested help. Teams of two or three took 71 minutes and 43% of these teams requested help. Interestingly, individuals who implemented the visitor without using JDT refactorings completed the assignment in 76 minutes, and teams of two or more took 54 minutes, none requesting help.

To put these numbers into perspective, $\mathcal{R}^2$ took 13 seconds (6 seconds estimated) to correctly perform the task. Even relatively simple refactorings – in this case creating a visitor with 7 methods – is harder than it seems. It is easy to get lost and make mistakes. We have concluded that any help is beneficial.

The data coming from the case study shows too that $\mathcal{R}^2$'s automation can not be beaten by manual transformations. Each Eclipse refactoring invoked in `makeVisitor` took roughly 1 second. Our largest experiment, A3, which invoked 554 Eclipse refactorings took 547 seconds (over 9 minutes). Had programmers done this experiment by hand, we believe that most of them would have given up at the sheer scale of change.

		Individual	Team of 2-3
Use Eclipse JDT	**Time to Finish (min)**	107	71
	Reference Help Pages (%)	60	43
Manual change	**Time to Finish (min)**	76	54
	Reference Help Pages (%)	0	0

Table 3. User Study to Introduce a Visitor.

Using $\mathcal{R}^2$, there can be a huge improvement in programmer productivity even for programmers who are experts on working with Eclipse's refactoring tools. A single Eclipse refactoring requires several mouse clicks (or keyboard shortcuts) and text inputs. When multiple refactorings are applied in concert, the number of clicks and inputs become burdensome. Manual interactions with dialogs require additional decisions (*eg* when refactorings violate preconditions, but refactorings can repair these exceptions), all of which is hidden by $\mathcal{R}^2$.

6.2.2 Is $\mathcal{R}^2$ Scalable?

Visitor Pattern: Using the data from the case study, Table 2 lists results of applying our `makeVisitor` refactoring to different methods in multiple applications. Each row represents data from a subject program. The columns are:

- **Seed ID** identifies the experiment.
- **Subject** is the Java subject program.
- **Seed Method Name** is the seed of the visitor.

- **Super Del** is the number of super delegates created (Section 5.1).
- **Chg Sig** is the number of `change method signatures` applied.
- **Move** is the number of methods moved into the visitor.
- **Ren** is the number of methods renamed, which is the sum of the **Chg Sig** and **Move** columns.
- **# of Refact** is the total number of Eclipse refactorings invoked by the `makeVisitor` call.
- **Elap-Time** is the clock time to perform `makeVisitor`. This includes the overhead for dialog box displays.
- **Esti-Time** is our estimate of the time specific to LTK core execution.
- **# of Errors** is the number of compilation errors that JDT refactorings generated, all which had to be manually corrected.
- **Bug IDs** are identifiers of the reported Eclipse bugs (see the Safety subsection 6.2.3 and Table 5).

About Table 2: recall that the $\mathcal{R}^2$ `makeVisitor` method of Figure 5 invokes `addParameter` to the list of methods that are relatives of the method seed. Ideally, these relatives are all descendant from a single root method (`A.m` in Figure 18a). This means that the $\mathcal{R}^2$ `addParameter` method invokes the Eclipse `change method signature` refactoring once on `A.m` to add an extra parameter to all of its relatives `B.m`, `C.m`.

In general there can be multiple roots. Figure 18b shows a seed whose relatives are not descendant from a single root. This means that the $\mathcal{R}^2$ `addParameter` method invokes the Eclipse `change method signature` refactoring three times, once for each root `E.m`, `F.m`, `G.m`, to add an extra parameter to all relatives.

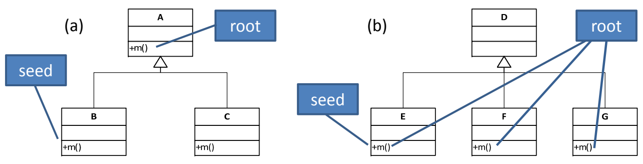


Figure 18. Method Seeds and Method Roots.

With the above in mind, now look at row/experiment `A3` in Table 2. Our tool created a visitor for the `printorder` method in `AHEAD`. `276` methods were moved into a visitor, no super delegates were created, and one `change method signature` was applied. The number of `rename methods` (`277`) was determined in this way: each method that is moved is renamed to `visit` (`276`). Although `276` method delegates were created, only one had to be renamed to `accept`. By renaming a root method, all of its descendants were renamed. Thus the total number of renames is $276 + 1 = 277$.

Now consider row/experiment `J3`. Our tool created a visitor for the `getName` method in `JUnit`. `5` methods were moved into a visitor, no super delegates were created, and `4` `change method signatures` were applied. The reason for `4` is that there were `4` method roots for the given seed (Figure 18b). Thus, the number of `renames` performed is `9`; `5` methods were moved, and `4` delegates (roots) were renamed.

Finally, consider row/experiment `R5`. Our tool created a visitor for the `pruneOriginalCandidates` method in `RefactoringCrawler`. `13` methods were moved into the visitor class, where `7` of these methods referenced "`super`" and thus required a `super` delegate for each to be created.

As $\mathcal{R}^2$ invokes refactorings automatically via dialog boxes, there is unnecessary overhead. Every Eclipse refactoring incurs three basic computations: `checkInitialConditions` confirms that the refactoring is applicable, `checkFinalConditions` performs the remaining precondition checks, and `createChange` changes the code [14, 45]. We profiled our executions to determine the amount of time spent in them and in dialogs. Let τ be the clock time that we measured to execute an $\mathcal{R}^2$ refactoring and γ be our estimate of run-time without GUI overhead, where `Esti-Time` equals γ, the time to execute `checkInitialConditions`, `checkFinalConditions`, and `createChange` APIs. To a good approximation, $\tau = 2\gamma$; that is, 50% of our clock time measurements estimates the LTK core execution-time of an $\mathcal{R}^2$ refactoring. We repeated each experiment five times and found no significant variance.

Inverse Visitor: As discussed in Section 3.2, an inverse Visitor can remove a manually-created visitor from a legacy program, or (as we concentrate here) can undo an $\mathcal{R}^2$-created visitor; it is definitely *not* just a "roll-back" of edits.

Seed ID	Chg Sig	Move/ Inline	Ren	# of Refact	Elap Time	Esti Time
A3	1	276	1	554	407s	308s
C2	4	4	4	16	14s	10s
C3	4	4	4	16	17s	10s
I1	1	1	1	4	4s	3s
I2	1	1	1	4	3s	2s
I3	1	2	1	6	5s	4s
I4	1	1	1	4	3s	2s
I5	1	2	1	6	4s	3s
J2	1	1	1	4	4s	2s
Q	1	7	1	16	10s	8s
R2	1	1	1	4	6s	4s

Table 4. Inverse Visitor Results.

Only experiments `A3`, `C2-C3`, `I1-I5`, `J2`, `Q`, `R2` produced no errors. It was for these cases that we could apply our inverse Visitor refactoring. Table 4 lists the results. Each Eclipse refactoring took roughly `1` second. To invert the `A3` visitor took `554` Eclipse refactorings for a total of `407` seconds (over `6` minutes). The `Esti-Time` column of Table 4 gives our estimated run-time for LTK core APIs. When fewer than 10 Eclipse refactorings are needed, run-times of a few seconds are acceptable. For more than 10 refactorings, the run-times seem too long to be "interactive".

Can $\mathcal{R}^2$ refactorings be applied to large programs? A qualified "yes" is the answer. Experiment `A3` invoked 554 transformations on a code base of over 26K LOC, and did so in 9 minutes (which we estimate could be reduced to 5 minutes).

6.2.3 Is $\mathcal{R}^2$ Safer than Manual Changes?

From our user study, we found that 60% of solo teams and 43% of multi-member teams that used JDT refactorings requested help because they introduced all sorts of errors and could not get the tests to pass.

Using the data from the case study, notice that numerous compilation errors were introduced by JDT refactoring bugs (*ie* column **# of Errors** in Table 2), all of which we manually repaired. These bugs are documented in Table 5. We reported these bugs to Eclipse bugzilla by August 2013 except one; `B1` was reported in February 2008. As of 3/6/2014 bugs with a ✓ have been resolved; those with a $\star$ are now in the current version of Eclipse (Kepler 4.3.1). We have since discovered more bugs and they will be explained in future papers.

Bug ID	Eclipse Bugzilla #	Description	Status
B1	217753	When a method with reference to `static import` is moved, the reference type is qualified incorrectly.	✓
B2	385550	When a method with reference to inherited fields is moved, the field access is not updated.	✓ $\star$
B3	385989	When a method with reference to `import` type is moved, the reference type is not qualified as the `import` type.	✓ $\star$
B4	404471	When a method with `@Override` annotation is moved, the annotation is also moved with the method.	✓
B5	404477	When a method is moved, wrong detection of duplicate methods occurs.	✓
B6	411529	When a method with reference to `protected` methods is moved to other package, `protected` is not modified to `public`.	✓
B7	416198	When a method is inlined, its reference in `enum` constructor call is not updated.	

Table 5. Eclipse Bug Reports.

Manually repairing Eclipse-introduced errors is tedious and error-prone. If many errors are introduced (*eg* experiments `A5` and `C1`), it is unclear whether programmers would want to fix the errors or just abandon the refactoring altogether.

Using Eclipse JDT, we implicitly adopted its strengths and frailties. We built our tool on a foundation that we thought was rock-solid; to our chagrin, it resembled quicksand. $\mathcal{R}^2$ *should* be safer than applying changes manually. Trust is the real issue: correctness of refactorings is a significant scientific problem and lack of trust (correctness) is a serious practical problem for future refactoring tools.

7. Related Work

Writing program transformations is a non-trivial exercise as research has shown [3, 5–9, 14, 16, 18, 23–26, 29, 32, 33, 36, 40, 42–44]. Prior work introduced a number of impressive metaprogramming languages such as `ASF + DSF` [42], `iXj` [6], `JunGL` [44], `Parlanse` [5], `Rascal` [18], `Refacola` [36], `SOUL` [25], `Stratego` [8], `Tom` [3], and `TXL` [9]. None match our requirements.

There are two primary distinctions between $\mathcal{R}^2$ and prior work. First is using the base language (*ie* the language in which programs to be refactored are written) as the refactoring script language. Second, writing primitive refactorings (ie rename method, move method, change method signature, etc.) is non-trivial and adds further burden on the potential user of program transformation systems. It is important to distinguish approaches that leverage existing refactoring engines from those where primitives have to be written by users. Table 6 categorizes these distinctions to the best of our knowledge.

At the top of Table 6 are tools, $\mathcal{R}^2$ and Wrangler, whose refactoring scripting language is the same as the base language. The remaining rows of tools use a different scripting language (possibly even a different scripting paradigm) than the base language and/or requires writing primitive refactorings. Tools that have ● for a base language can be applied to any language, given its language specification in that tool's formalism(s). Below we highlight representative works in Table 6.

`JunGL` and `Refacola` are DSLs specialized for scripting refactorings. `JunGL` is the `ML`-style functional language implemented on the `.NET` platform and targeted the `C#` language. `JunGL` facilitates AST manipulation with higher order functions and tree pattern matching, and also supports querying facilities for semantic and data flow information look-up. `Refacola` is a constraint language where refactorings are specified by constraint rules. The `Refacola` framework supports implementation of program element queries and constraint generation.

Program transformation systems are monuments of engineering prowess. Among them are `Codelink` [40], `DMS` [5], `SmaCC` [7], `SYDIT` [24], `Wrangler` [23], and `XT` [8]. `Wrangler`, in particular, is a refactoring framework implemented in functional language `Erlang` which is also the target language. `Wrangler` supports refactoring commands for locating program elements to be refactored, and provides a custom DSL to execute the commands.

Hills et al. [18] also use JDT refactorings, which are available as APIs in the `Rascal JDTRefactoring` library. They too target Java, but their scripting language (`Rascal`) is not an OO language. Further, manual code changes are required in their transformation process to fix incorrect access modifiers, clean up unnecessary codes, *etc*, which we would have preferred to have a tool to automatically.

Mens et al. [25] use declarative metaprogramming to define design patterns and their constraints in a language-independent manner. Their use of a variant of Prolog is elegant, as they tackle problems similar to $\mathcal{R}^2$.

Finally, $\mathcal{R}^2$ deals with refactoring scripts, *not* when refactorings should be applied. There are excellent papers on this

Tool or DSL	Scripting Language	Base Language	Paradigm	Primitives Available
$\mathcal{R}^2$	Java	Java	Imperative	✓
Wrangler	Erlang	Erlang	Functional	
ASF+SDF	ASF, SDF	●	Term Rewrite	
iXj	iXj	Java	Imperative	
JunGL	JunGL	C#	Functional and Logic Query	
DMS	Parlanse	●	Lisp-like	
Rascal	Rascal	Java	Imperative, Non-OO	✓
Refacola	Refacola	Eiffel, Java	Constraint	
SOUL	SOUL	Java, C, Cobol, Smalltalk	Logic Programming	✓
XT	Stratego	●	Term Rewrite	
Tom	Tom	C, Java, Python, C++, C#, *etc*	Term Rewrite	
TXL	TXL	●	Functional, Term Rewrite	
Codelink	(GUI-based)	●	(N/A)	
SmaCC	SmallTalk	Java, C#, Delphi	Imperative	
SYDIT	(GUI-based)	Java	(N/A)	

● indicates arbitrary languages that can be defined by users.

Table 6. Tools and Languages to Script Refactorings

topic [11, 20, 28, 34, 35, 38, 41] but all are orthogonal to the use and goals of $\mathcal{R}^2$.

8. Conclusions

Introducing design patterns into a program using refactorings is tedious and error-prone. The burden can be alleviated, either partially or fully, by refactoring scripts. Today's IDEs offer poor support for such scripts, or require a background and understanding of IDE internals that students and most programmers will never have. Proposed DSLs that can be used for scripting typically require knowledge of yet another programming language and maybe even require the need to code primitive refactorings.

Our solution $\mathcal{R}^2$ uses (1) Java as a metaprogramming language, (2) metaobjects are classes, methods, and fields of a Java program, and (3) metaobject methods are native JDT refactorings or our scripts. We have used $\mathcal{R}^2$ to automate classical design patterns, where each $\mathcal{R}^2$ script is a compact Java method. We fail to see how any DSL could make these scripts significantly shorter. And we fail to see how any DSL (exposing JDT refactorings as primitives) would be any easier to learn than Java.

Our experiments show that large refactoring scripts can be successfully applied to non-trivial programs (554 refactorings applied to a code base of 26K), that refactoring scripts can accomplish even small objectives and save significant time (reducing an hour of work to 13 seconds to create a visitor with 7 methods), and *in principle* can be safer than a manual process.

Our work reveals an Achilles heel in contemporary refactoring technology. Despite countless man-years of JDT development, we repeatedly discovered errors not of our making were introduced by JDT refactoring bugs. There is a practical, not theoretical, limit to extend Eclipse with scripting. If too many errors are introduced, it is unclear whether programmers would want to fix the errors or just abandon the refactoring altogether. Correctness of refactorings remains a significant scientific problem and lack of trust (correctness) remains a serious practical problem.

Further, the speed at which refactoring scripts execute is a concern. To execute 554 refactorings took 9 minutes (5 minutes estimated). While the magnitude of this task would virtually preclude it from being done manually, and although this run-time is miniscule to the time a manual attempt would require, it is too long to be truly "interactive".

It remains a conjecture on our part that students can script refactorings using $\mathcal{R}^2$. This will take more time. Never-the-less, we are convinced that IDE support for refactoring scripts is an important problem that next-generation refactoring engines must solve. Our work takes us a step closer to this goal.

Acknowledgments

We thank referees for their insightful comments. We also thank C. Brindescu, M. Codoban, A. Gyori, M. Hilton, Y. Lin, S. Shmarkatiuk, and F. Steimann, for valuable suggestions.We gratefully acknowledge support for this work by NSF grants CCF-1212683 and CCF-1213091.

References

[1] Apache Commons Codec. `commons.apache.org/proper/commons-codec/`.

[2] Apache Commons IO. `commons.apache.org/proper/commons-io/`.

[3] E. Balland and et al. Tom: piggybacking rewriting on java. In *RTA*, 2007.

[4] D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *GTTSE*, 2005.

[5] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE*, 2004.

[6] M. Boshernitsan and S. L. Graham. iXj: interactive source-to-source transformations for java. In *OOPSLA Companion*, 2004.

[7] J. Brant and D. Roberts. The SmaCC transformation engine: how to convert your entire code base into a different programming language. In *OOPSLA Companion*, 2009.

[8] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, June 2008.

[9] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, Aug. 2006.

[10] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[11] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring-Improving Coupling and Cohesion of Existing Code. In *WCRE*, 2004.

[12] Eclipse Java development tools (JDT). `eclipse.org/jdt/`.

[13] Eclipse Juno. `eclipse.org/juno/`.

[14] L. Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. `www.eclipse.org/articles/Article-LTK/ltk.html/`.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman, 1995.

[16] A. Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, U. of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

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

[18] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with Rascal and Eclipse. In *WRT*, 2012.

[19] JUnit. `junit.org/`.

[20] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In *ICSM*, 2002.

[21] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[22] J. Kim, D. Batory, and D. Dig. Can Undergraduates Script Their Own Refactorings? In *WRT*, 2013.

[23] H. Li and S. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In *FASE*, 2012.

[24] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, 2011.

[25] T. Mens and T. Tourwe. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *ICSM*, 2001.

[26] R. C. Miller and B. A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX*, 2001.

[27] V. K. Nandivada. Private conversation, 2014.

[28] M. O'Keeffe and M. O'Cinneide. Search-Based Software Maintenance. In *CSMR*, 2006.

[29] B. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, U. of Illinois at Urbana-Champaign, 1992.

[30] W. F. Opdyke and R. E. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *SOOPA*, 1990.

[31] Dialect user's guide, 1990.

[32] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, U. of Illinois at Urbana-Champaign, 1999.

[33] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *OOPSLA*, 2010.

[34] O. Seng, J. Stammel, and D. Burkhart. Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems. In *GECCO*, 2006.

[35] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics Based Refactoring. In *CSMR*, 2001.

[36] F. Steimann, C. Kollee, and J. von Pilgrim. A Refactoring Constraint Language and its Application to Eiffel. In *ECOOP*, 2011.

[37] J. Sugrue. Design Patterns Uncovered: The Visitor Pattern. `http://java.dzone.com/articles/design-patterns-visitor`, 2010.

[38] L. Tahvildari and K. Kontogiannis. A Metric-Based Approach to Enhance Design Quality Through Meta-pattern Transformations. In *CSMR*, 2003.

[39] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *ASE*, 1999.

[40] M. Toomim, A. Begel, and S. L. Graham. Managing Duplicated Code with Linked Editing. In *VLHCC*, 2004.

[41] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Trans. Softw. Eng.*, May 2009.

[42] M. van den Brand and et al. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *CC*, 2001.

[43] M. G. van den Brand and et al. Using The Meta-Environment for Maintenance and Renovation. In *CSMR*, 2007.

[44] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE*, 2006.

[45] T. Widmer. Unleashing the Power of Refactoring. `www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/`.

Appendices

A. Examples of Fully Automatable Patterns

A.1 Abstract Factory

The Abstract Factory pattern provides a general interface for concrete factories. Figure 19 shows interface `AbstractFactory` that is implemented by concrete factory class `ConcreteFactory`. `ConcreteFactory` consists of factory methods produced for every public/protected constructor in a given package. (The package of Figure 19a contains classes `A` and `B`; Figure 19b contains classes `A`, `B`, `AbstractFactory` and `ConcreteFactory`). Figures 20 and 21 are $\mathcal{R}^2$ methods that produce these concrete and abstract factories.

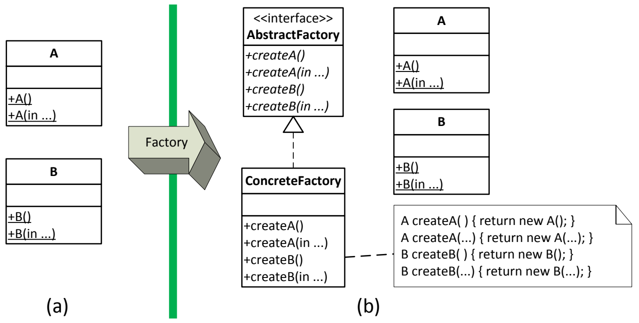


Figure 19. Factory Pattern.

```
// member of RPackage class
RClass makeConcreteFactory(String factoryName)
{
  RClass f = this.newClass(factoryName);

  for(RClass c : this.getClass(RModifier.Public |
        RModifier.Protected)) {
    for(RMethod m : c.getConstructor(RModifier.
         Public | RModifier.Protected)) {
      f.newFactoryMethod(m);
    }
  }

  return f;
}
```

Figure 20. A `makeConcreteFactory` Method.

```
// member of RClass class
RInterface makeAbstractFactory(String absFacName)
{
  RInterface i = getRPackage().newInterface(
       absFacName);

  for(RMethod m : getMethod()) {
    i.newMethod(m);
  }

  this.setInterface(i);

  return i;
}
```

Figure 21. A `makeAbstractFactory` Method.

A.2 Command and Memento

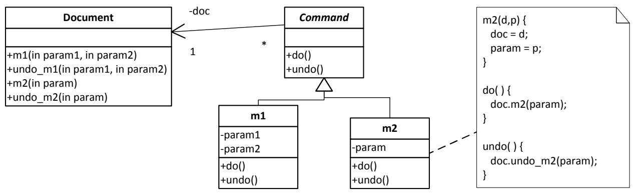


Figure 22. Command and Memento Patterns.

The Command pattern encapsulates a method invocation as an object, typically for journaling. The Memento pattern adds the ability to undo method calls, and completes the usual abilities of logs to undo and redo calls. Figure 22 presents an example of Memento where an undo method is

```
// member of RMethod class
RClass makeCommand(String commandName)
{
  RClass c = getRPackage().newClass(commandName);

  ArrayList<RField> l = new ArrayList<RField>();

  RField f = c.newField(getRClass());
  l.add(f);

  for(RParameter p : getParameters()) {
    f = c.newField(p);
    l.add(f);
  }

  c.newConstructor(l);
  c.newCommandMethod(this);

  return c;
}
```

Figure 23. A `makeCommand` Method.

```
// member of RMethod class
RClass makeMemento(RClass commandClass) {
  commandClass.newCommandMethod(this);
  return commandClass;
}
```

Figure 24. A `makeMemento` Method.

given for every do method. (Another way is to remember the state of an object prior to a do method call; undo restores the previous object state. This would be another $\mathcal{R}^2$ method variant). The `makeCommand` $\mathcal{R}^2$ method in Figure 23 is applied to the target method and creates a concrete command class. The `makeMemento` $\mathcal{R}^2$ method in Figure 24 adds the target undo method to the command class, which becomes a memento class.

B. Partially Automatable Patterns

B.1 Strategy

The Strategy pattern encapsulates each concrete algorithm independently and selects one dynamically at runtime. The participants in Strategy are a common strategy interface, concrete strategies, and a `Context` class that references the strategy interface (Figure 25).

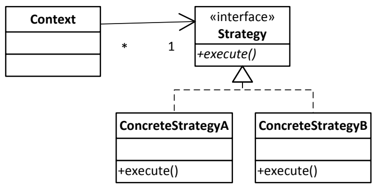


Figure 25. Strategy Pattern.

The `makeStrategy` $\mathcal{R}^2$ method in Figure 26 is applied to a method (that becomes a strategy) in class `Context`. Then, an interface (called `Strategy`) is created along with a concrete class where the strategy method is moved from `Context`. A field of the concrete class type is added to

Context and the strategy method is moved via the field. Our $\mathcal{R}^2$ method can be extended by adding different concrete strategy classes.

```
// member of RMethod class
RClass makeStrategy(String strategyName)
{
  RPackage p = getRPackage();
  RInterface i = p.newInterface(strategyName);
  i.newMethod(this);

  RClass c = p.newClass(getName() + "Strategy");
  c.newMethod(this);
  c.setInterface(i);

  RField f = getRClass().newField(c, String.
      format("new %s()", c.getFullName()));
  this.moveAndDelegateVia(f);

  f.setType(i);

  return c;
}
```

Figure 26. A makeStrategy Method.