Scripting Parametric Refactorings in Java to Retrofit Design Patterns

Jongwook Kim  
University of Texas at Austin  
Austin, TX 78712, USA  
Email: jongwook@cs.utexas.edu

Don Batory  
University of Texas at Austin  
Austin, TX 78712, USA  
Email: batory@cs.utexas.edu

Danny Dig  
Oregon State University  
Corvallis, OR 97333, USA  
Email: digd@eecs.oregonstate.edu

Abstract—Retrofitting design patterns into a program by hand is tedious and error-prone. A programmer must distinguish refactorings that are provided by an Integrated Development Environment (IDE) from those that must be realized manually. A programmer must determine a precise sequence of refactorings to apply, and perform this sequence repetitively to a laborious degree. We designed, implemented, and evaluated Reflective Refactoring (R²), a Java package to automate the creation of classical design patterns (Visitor, Abstract Factory, etc.), their inverses, and variants. We encoded 18 out of 23 Gang-of-Four design patterns as R² scripts and explain why the remaining are inappropriate for refactoring engines. We evaluate the productivity and scalability of R² with a case study of 6 real-world applications. In one case, R² automatically created a Visitor with 276 visit methods by invoking 554 Eclipse refactorings in 10 minutes—an achievement that could not be done manually. R² also sheds light on why refactoring correctness, expressiveness, and speed are critical issues for scripting in next-generation refactoring engines.

I. INTRODUCTION

Most design patterns are not present in a program during the design phase, but appear later in maintenance and evolution [1]. Modern IDEs—Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio—offer primitive refactorings (e.g., rename, move, change-method-signature) that constitute basic steps to retrofit design patterns into a program [2], [3]. It has been over 20 years since design patterns were popularized [2], [3] and longer still for refactorings [4]–[6]. For at least 15 years it was known that many design patterns could be automated by scripting transformations [1], [7]. So it is both surprising and disappointing that modern IDEs automate few patterns and offer no means to script transformations or refactorings to introduce whole patterns.

Manually introducing design patterns using primitive refactorings from the IDE is error-prone. To retrofit a Visitor pattern into a program requires finding all relevant methods to move by hand and applying a sequence of refactorings in precise order. It is easy to make mistakes. Missing a single method in a class hierarchy produces an incomplete but executable Visitor. But a future extension that uses the Visitor can break the program (Section III-A).

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 primitive transformations to mechanize them. Doing so forces students, and programmers in general, to understand the nuances and capabilities of each refactoring and pattern. Although we are primarily motivated to improve tools for teaching refactorings and patterns, 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 [8]–[18], but all fall short in fundamental ways for our goal. It is unrealistic to expect that students can quickly learn sophisticated Program Transformation Systems (PTSs) [9]–[11], [19] or utilities, such as Eclipse Language Toolkits (LTKs) [20], to manipulate programs. Although PTSs and LTKs are monuments of engineering prowess, their learning curve is measured in weeks or months. Domain Specific Languages (DSLs) to write refactoring scripts still have an unneeded overhead [8]–[13], [15]–[18].

We present a practical way to move Java refactoring technology forward. We designed, implemented, and evaluated Reflective Refactoring (R²), a Java package whose goal is to encode the construction of classical design patterns as Java methods. Using Eclipse Java Development Tools (JDT) [21], R² leverages reflection by presenting a JDT project, its package, class, method and field declarations as Java objects whose methods are JDT refactorings. Automating design patterns becomes no different than importing an existing Java package (R²) and using it to write programs (in this case, refactoring scripts). There is no need for a DSL.

Our paper makes the following contributions:

• JDT Extensions. JDT refactorings, as is, were never designed to script design patterns. We describe our repairs to make JDT supportive for scripting.

• Object-Oriented (OO) Metaprogramming. We present a Java package, R², with several novel features to improve refactoring technology. R² objects are Java entity declarations and R² methods are JDT refactorings, primitive R² transformations, R² pattern scripts, and program element navigations (i.e., R² object searches).

• Generality. We encoded 18 out of 23 Gang-of-Four design patterns [3], inverses, and variants as short Java methods in R², several of which we illustrate. This shows that R² can express a wide range of patterns.

• Implementation. R² is also an Eclipse plugin that leverages existing JDT refactorings and enables programmers to script many high-level patterns elegantly.

I. INTRODUCTION

Most design patterns are not present in a program during the design phase, but appear later in maintenance and evolution [1]. Modern IDEs—Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio—offer primitive refactorings (e.g., rename, move, change-method-signature) that constitute basic steps to retrofit design patterns into a program [2], [3]. It has been over 20 years since design patterns were popularized [2], [3] and longer still for refactorings [4]–[6]. For at least 15 years it was known that many design patterns could be automated by scripting transformations [1], [7]. So it is both surprising and disappointing that modern IDEs automate few patterns and offer no means to script transformations or refactorings to introduce whole patterns.

Manually introducing design patterns using primitive refactorings from the IDE is error-prone. To retrofit a Visitor pattern into a program requires finding all relevant methods to move by hand and applying a sequence of refactorings in precise order. It is easy to make mistakes. Missing a single method in a class hierarchy produces an incomplete but executable Visitor. But a future extension that uses the Visitor can break the program (Section III-A).

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 primitive transformations to mechanize them. Doing so forces students, and programmers in general, to understand the nuances and capabilities of each refactoring and pattern. Although we are primarily motivated to improve tools for teaching refactorings and patterns, 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 [8]–[18], but all fall short in fundamental ways for our goal. It is unrealistic to expect that students can quickly learn sophisticated Program Transformation Systems (PTSs) [9]–[11], [19] or utilities, such as Eclipse Language Toolkits (LTKs) [20], to manipulate programs. Although PTSs and LTKs are monuments of engineering prowess, their learning curve is measured in weeks or months. Domain Specific Languages (DSLs) to write refactoring scripts still have an unneeded overhead [8]–[13], [15]–[18].

We present a practical way to move Java refactoring technology forward. We designed, implemented, and evaluated Reflective Refactoring (R²), a Java package whose goal is to encode the construction of classical design patterns as Java methods. Using Eclipse Java Development Tools (JDT) [21], R² leverages reflection by presenting a JDT project, its package, class, method and field declarations as Java objects whose methods are JDT refactorings. Automating design patterns becomes no different than importing an existing Java package (R²) and using it to write programs (in this case, refactoring scripts). There is no need for a DSL.

Our paper makes the following contributions:

• JDT Extensions. JDT refactorings, as is, were never designed to script design patterns. We describe our repairs to make JDT supportive for scripting.

• Object-Oriented (OO) Metaprogramming. We present a Java package, R², with several novel features to improve refactoring technology. R² objects are Java entity declarations and R² methods are JDT refactorings, primitive R² transformations, R² pattern scripts, and program element navigations (i.e., R² object searches).

• Generality. We encoded 18 out of 23 Gang-of-Four design patterns [3], inverses, and variants as short Java methods in R², several of which we illustrate. This shows that R² can express a wide range of patterns.

• Implementation. R² is also an Eclipse plugin that leverages existing JDT refactorings and enables programmers to script many high-level patterns elegantly.
• Evaluation. A case study shows the productivity and scalability of $R^2$. We applied a 20-line $R^2$ script to retrofit 52 pattern instances into 6 real-world applications. One case invoked 554 refactorings, showing that $R^2$ scales well to large programs.

II. A Motivating Example

Among the most sophisticated patterns is Visitor. There are different ways to encode a Visitor; we use the one below. Figure 1 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.

Mechanics. To create a Visitor for the draw method (Figure 1a), a programmer first creates a singleton Visitor class DrawVisitor. Next, s/he moves each draw method into the DrawVisitor class, renames it to visit, and adds an extra parameter (namely the class from which the method was moved). Referenced declarations (e.g., fields and methods) must become visible by changing their access modifiers after a method move [22]. Further, s/he creates a delegate (named accept) 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 for our example is:

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

Finally, s/he replaces all calls to the draw method with calls to accept. Note that some of these steps can be performed by JDT refactorings, but they require knowledge and familiarity with available refactorings to know which to use and in what order. Further, after each step, the programmer recompiles the program and runs regression tests to ensure that the refactored program was not corrupted.

Pitfalls. It is easy to make a mistake or forget a step. A programmer can inadvertently skip draw methods to move. Suppose a missed method is Triangle.draw. Although the refactored code would compile and execute correctly in this version, it breaks when another kind of Visitor is added in a future maintenance task. Example: another programmer creates a SmallScreenVisitor that displays widgets for small screens of smartphones. When s/he passes an instance of the SmallScreenVisitor instead of the DrawVisitor, the Triangle.draw method will render the original behavior for a large screen, not the expected one for small screens.

Complicating Issues. JDT refactorings were never designed with scripting in mind. We encountered a series of design and implementation issues in the latest version of Eclipse JDT (Luna 4.4.1, Dec. 2014) [23] that compromises its ability to support refactoring scripts without considerable effort. (These issues need to be addressed, regardless of our work). Here are examples.

A. Separation of Concerns

Figure 2 shows method draw in class Square, after a DrawVisitor parameter was added. Figure 2 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 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 consistent result, it is an error. Preserving all parameters of moved methods in a Visitor pattern is essential. Two concerns – method movement and method signature optimization – were bundled into a single refactoring, instead of being separated into distinct refactorings. We programmatically deactivated method signature optimizations in $R^2$; users cannot disable such optimizations from the Eclipse GUI.
B. Need for Other (Primitive) Refactorings

Suppose that we want to “undo” an existing Visitor – eliminate the target Visitor class by moving its contents back into existing class hierarchies. Each `visit` method in the Visitor is moved back to its original class. As an example, Figure 3b shows class Triangle after such a move: Triangle has both `accept` and `visit` methods. When the `visit` method is inlined, the `accept` method absorbs the `visit` method body (Figure 3b).

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

Fig. 3. Restriction of JDT inline Refactoring.

Unfortunately, Eclipse refuses to inline the `visit` method since a `return` statement potentially interrupts execution flow. This precondition prevents automating a Visitor “undo”. We had to deactivate this precondition check to script the Inverse-Visitor described in Section III-B, in effect adding a new refactoring to JDT, to accomplish our task.

C. 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 parent class. Moving methods with `super` calls is not only possible, it is desirable. Unfortunately, JDT refuses to move methods that reference `super`. It is not an error, but a strong limitation. We removed this limitation by replacing each `super.x()` call with a call to a manufactured method `super_xθ()`, whose body calls `super.x(); θ` is just a random number to make the name of the manufactured method unique.¹²

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

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

Fig. 4. Rewrite that Uses super Delegate.

In Figure 4, the `super` keyword invokes an overridden method `A.foo()`. We remove `super` by calling a delegate method which calls the overridden method `A.foo()`. Figure 4b shows a super delegate `super.fooθ()` which replaces the `super.foo()` call in `B.bar()`, thus allowing JDT to move `B.bar()` to the Visitor class. Of course, `super-delegates` throw the same exception types as its `super invocation`.

Now consider the use of `super` to reference fields of a parent class. Again, JDT refuses to move methods with `super`-references to fields. Here is how we fixed this: fields in Java are hidden and not overridden. So we can get `super` references simply by casting to their declared type. In Figure 5 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`.

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

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

Fig. 5. super Field Access.

D. Recap

Many patterns cannot be created with off-the-shelf JDT without considerable manual effort as existing refactorings fall short of what is required. We have repairs for JDT, and now our next step is scripting, which we discuss next.

III. 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 [8]–[14], [16]–[18]. But as we said earlier, the learning curve to become proficient in yet another language or programming paradigm makes these approaches unappealing. The obvious answer is to script refactorings in Java.

Let `P` be a JDT project. We leverage the idea of reflection; `R²` defines class `RClass` whose instances are the class declarations in `P`; instances of classes `RMethod` and `RField` are the method and field declarations of `P`, and so on. When `P` is compiled, `R²` creates a set of main-memory database tables (one for `RClass`, `RMethod`, `RField`, etc.) where each row corresponds to a class, a method, or a field declaration of `P`. *These tables are not persistent; they exist only when the JDT project for `P` is open.*

The fields of `RClass`, `RMethod`, `RField`, etc. – henceforth called `R² classes` – also define association, inheritance, dependency relationships among table rows (`foo` is a method of class `A`, `A` is a superclass of `B`, `B` belongs to package `C`, etc.).
The member methods of $R^2$ classes are JDT refactorings, simple $R^2$ transformations, composite refactorings (our scripts), and ways to locate program elements (i.e., $R^2$ objects).

Internally, we leveraged XML scripts which Eclipse uses only to replay refactoring histories. An $R^2$ method call generates an XML script which we then feed to JDT to execute. In this way, we automate exactly the same procedures Eclipse users would follow manually. $R^2$ exposes every available JDT refactoring as a method and a few more (Section III). Overall, we changed 51 lines in 8 JDT internal packages; the $R^2$ package consists of $\sim 5$ K LOC.

In the following subsections, we give readers a feel for $R^2$ scripts by illustrating interesting examples.

A. Automating the Visitor Pattern

Visitor is fully automatable as an $R^2$ script. For a programmer to create a Visitor for some method $m$, s/he points to $m$ as a “seed” in the Eclipse editor and invokes the makeVisitor $R^2$ script via the Eclipse GUI. A parameter of makeVisitor is the name of the Visitor class. All methods related to $m$ are moved into the Visitor. So, from a programmer’s viewpoint, an $R^2$ script is indistinguishable from an existing JDT refactoring.

Figure 6 shows our makeVisitor, a method of class RMethod. The Java keyword this refers to the “seed” method to which the script is applied. Lines $3–5$ create a Visitor class (called visitorClassName) in the same package as this and add a static Singleton field instance. Lines $6–8$ find all methods (called “relatives”) with the same signature as this and add a new parameter of type visitorClassName to each of these methods. Calls to relative methods have visitorClassName as the default extra argument. Lines $9–15$ move each movable method to the Visitor class, leave behind a delegate, and rename each method to visit. Lines $16–18$ collect delegate relatives and rename them to accept. Line $19$ returns the Visitor class.

Looping through a list of methods and invoking a refactoring on each method would be the obvious way to add a parameter to relatives. But this is not how the JDT change-method-signature refactoring works (Lines $7–8$). It is applied to the “seed” method only. Consider Figure 2. 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 (relatives) will have their signature changed. The methodList variable in Line $7$ is the list of all methods in $P$ 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 $\{A, B, C, D, E\}$.

To add another script, its method is added to $R^2$. Eclipse is then run with the updated $R^2$.

Delegate relatives include generated delegate methods and methods that cannot be moved, e.g. interface and abstract methods.

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

Fig. 6. A makeVisitor Method.

Note: Although Eclipse provides ways to find methods, it is still easy to miss program methods (relatives) that are distributed over the entire program. Forgetting to move a method when creating a Visitor manually is easy, yet it is hard to detect as no compilation errors identify non-moved methods. $R^2$ eliminates such errors by invoking a trustworthy $R^2$ getRelatives() method.

B. Automating the Inverse Visitor

Figure 8 depicts a common scenario: An $R^2$ programmer creates a Visitor to provide a convenient view that allows her/him to inspect all draw methods in the graphics class hierarchy from our motivating example of Figure 1. The programmer then updates the program, including Visitor methods, as part of some debugging or functionality-enhancement process. At which point, s/he wants to remove the Visitor to return the program back to its original structure.

```
Fig. 7. Methods Altered by Change Signature.
```

Fig. 8. A Common Programming Scenario.
In this scenario, undoing a Visitor is not a roll-back, as a roll-back removes all of the programmer’s debugging edits. Instead, an Inverse-Visitor – a refactoring that removes a Visitor and preserves debugging edits – is required. Yet another practical reason is if a program already contains a hand-crafted Visitor, weaving its methods back into the class hierarchy would be an optimization. Similar scenarios apply to other design patterns, such as Builder and Factory Method.

Figure 9 shows our inverseVisitor, a method of RClass, that moves visit methods back to their original classes and deletes the Visitor class. Here is how it works: Lines 8-9 recover the original class of a visit method. As we turned off method signature optimization in Section II-A, the original class is encoded as the type of the visit method’s first parameter. Line 11 moves the method back to its original class. Lines 12-14 inline super-delegates if they exist by replacing each call to super_xθ() with super.x() (Section II-C) and then restore the original method body (which is the body of the visit method) by inlining. Lines 6-14 are performed for all visit methods. At this point, the accept methods (i.e., the delegate methods) contain the body of the original methods. Lines 17-20 collect all of the accept methods, remove their first parameter (of type Visitor class), and restore the original name of the method. The Visitor class is then deleted in Line 22.

Note: The challenge is to determine the correct order to apply move and inline refactorings. What if every visit method is moved and then inline is applied to each visit? To see the problem, let class A be the parent of class B and suppose both A and B have visit methods. Now, B.visit is inlined. B still inherits A.visit. Eclipse recognizes that inlining might alter program semantics and issues a warning: “method to be inlined overrides method from the parent class”. A similar warning arises had A.visit been inlined first. The solution is to move one method at a time, followed by an inline, as done in Figure 9 to avoid warnings.

C. More Opportunities

Design patterns have many variations; Visitor is no exception. Consider Visitor PV of Figure 10 adapted from [24]. It differs from the Visitor of our example of Section II in several ways: PV is not a Singleton, it includes state totalPostage, it has a custom non-visit method getTotalPostage(), and at least one of its visit methods visit(Book) references totalPostage.

The Visitor variant of Figure 11 requires a slight modification of our R^2 inverseVisitor method. Figure 11 shows the modified method; it differs from Figure 9 by moving only methods named visitMethodName, not removing the Visitor parameter, and not deleting the Visitor class.

These examples illustrate the power of R^2: (1) we can automate these patterns (by transforming a program without these patterns into programs with these patterns), (2) we can remove these patterns (by transforming programs with hand-crafted patterns into programs without those patterns), and (3) express common variations that arise in design patterns. R^2 offers a practical way to cover all of these possibilities.
IV. OTHER PATTERNS

Figure [12] is our review of the Gang-of-Four Design Patterns text [3]: 8 out of 23 patterns are fully automatable, 10 are partially automatable. For the remaining 5 patterns, we are unsure of their role in a refactoring tool (although some are automatable). \( R^2 \) scripts for all of the 18 automatable patterns are listed in [25]. We elaborate our key findings below.

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>Automation Possibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>✓</td>
</tr>
<tr>
<td>Adapter</td>
<td>✓</td>
</tr>
<tr>
<td>Bridge</td>
<td>✓</td>
</tr>
<tr>
<td>Builder</td>
<td>✓</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>✓</td>
</tr>
<tr>
<td>Command</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
</tr>
<tr>
<td>Decorator</td>
<td>✓</td>
</tr>
<tr>
<td>Façade</td>
<td>✓</td>
</tr>
<tr>
<td>Factory Method</td>
<td>✓</td>
</tr>
<tr>
<td>Flyweight</td>
<td>✓</td>
</tr>
<tr>
<td>Interpreter</td>
<td>✓</td>
</tr>
<tr>
<td>Iterator</td>
<td>✓</td>
</tr>
<tr>
<td>Mediator</td>
<td>✓</td>
</tr>
<tr>
<td>Memento</td>
<td>✓</td>
</tr>
<tr>
<td>Observer</td>
<td>✓</td>
</tr>
<tr>
<td>Prototype</td>
<td>✓</td>
</tr>
<tr>
<td>Proxy</td>
<td>✓</td>
</tr>
<tr>
<td>Singleton</td>
<td>✓</td>
</tr>
<tr>
<td>State</td>
<td>✓</td>
</tr>
<tr>
<td>Strategy</td>
<td>✓</td>
</tr>
<tr>
<td>Template Method</td>
<td>✓</td>
</tr>
<tr>
<td>Visitor</td>
<td>✓</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 12. Automation Potential of Gang-of-Four Design Patterns.

A. Fully Automatable Patterns

The Visitor pattern is its inverse and variants are fully automatable as they produce no "TO DOs" for a user. Another is Abstract Factory which provides an interface to concrete factories. Figure [13] shows interface AbstractFactory that exposes factory methods for every public constructor of each public class in a given package: the package of Figure [13] contains classes A and B; the interface AbstractFactory is implemented by concrete factory class ConcreteFactory in Figure [13]. Figure [14] is the \( R^2 \) method that produces a concrete factory for a package. A similar \( R^2 \) script creates the AbstractFactory interface.

B. Partially Automatable Patterns

10 out of 23 patterns are partially automatable, i.e., these patterns produce "TO DOs" that must be completed by a user. The Adapter pattern is typical. It resolves incompatibilities between a client interface and a legacy class. Given interface Target and class Legacy in Figure [15] an intermediate class (called Adapter) adapts Target to Legacy. The \( R^2 \) makeAdapter method in Figure [16] creates the Adapter class that implements interface Target and references class Legacy. Programmers must provide bodies for the generated method stubs; these are the user "TO DOs". Although partially automated – method bodies are still needed – tedious and error-prone work is done by \( R^2 \).

C. Remaining Patterns

We are unsure of the role for the remaining patterns in a refactoring tool (some of which are automatable):

- 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, will have. An $\mathcal{R}^2$ script can be written to produce a particular façade, but it will be application-specific and unlikely to be reusable.

- Interpreter is common in compiler-compiler tools [26, 27]; given a language’s grammar, a class hierarchy for creating language ASTs can be generated. Providing a grammar to a refactoring engine to generate a class hierarchy is possible, but seems inappropriate.

- State is a common application of Model Driven Engineering (MDE). Given a statechart of a finite state machine, MDE tools can generate the class hierarchies and method stubs that implement the State pattern. Again, providing a statechart to a refactoring engine to generate the code of a State pattern is possible, but also seems inappropriate.

- Mediator is the basis for GUI builders; the drag-and-drop of class instances from a palette of classes is the essence of a Mediator. Again, it is unclear that this functionality belongs in a refactoring engine.

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

V. CASE STUDIES, EVALUATION, AND PERSPECTIVE

We evaluated $\mathcal{R}^2$ by answering two research questions:

- RQ1: Does $\mathcal{R}^2$ improve productivity?
- RQ2: Can $\mathcal{R}^2$ be applied to large programs?

Both questions address the higher level question “Is $\mathcal{R}^2$ useful?” from different angles: Productivity measures whether $\mathcal{R}^2$ methods save programmer time. Scalability measures whether $\mathcal{R}^2$ can work with large programs.

A. Experiment

Some design patterns (e.g., Adapter) are relatively simple: create a few program elements, change class relationships, or make minor code changes. Others are different. All patterns are tedious and error-prone to create manually when the target program is non-trivial. There are $\mathcal{R}^2$ scripts for all 18 automatable patterns. We evaluate $\mathcal{R}^2$ using patterns that (a) exercise most $\mathcal{R}^2$ methods and capabilities and (b) are difficult to create manually. These are the Make-Visitor and Inverse-Visitor patterns, which we have already presented.

We used six real-world Java applications that satisfied the following criteria: (1) they were publicly available, (2) they had non-trivial class hierarchies, (3) regression tests were available for us to determine if our refactorings altered application behavior, and (4) 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 $\mathcal{R}^2$. The Subject column of Table I lists these applications, their versions, application size in LOC, and the number of regression tests. We used an Intel CPU i7-2600 3.40GHz, 16 GB main memory, Windows 7 64-bit OS, and Eclipse JDT 4.4.1 (Luna).

B. Results

We have two sets of results: creating a Visitor and removing a Visitor. First consider creating a Visitor. Table I lists results of Make-Visitor applied 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 Method Name is the number of super-delegates created (Section II-C).
- Change Signature is the number of change-method-signatures applied.
- Move is the number of methods moved into the Visitor.
- Rename is the number of methods renamed.
- # of Refactorings is the total number of JDT refactorings invoked by the makeVisitor call.
- Time is average clock time (in seconds) to perform makeVisitor.
- # of Errors is the total number of errors created by JDT bugs in the old version of Eclipse (Juno 4.2.2 [32]) that we started with.

RQ1: Does $\mathcal{R}^2$ Improve Productivity? Table I shows that $\mathcal{R}^2$ performs tasks that are unachievable manually. Our largest experiment, A3, invoked 554 JDT refactorings took 10 minutes. Had programmers attempted A3 by hand, we believe that most would have given up at its sheer scale.

$\mathcal{R}^2$ offers a huge improvement in productivity even for programmers who are experts in JDT refactorings. An $\mathcal{R}^2$ script takes a fraction of the time (with no user intervention): the order in which refactorings should be sequenced, their parameters, and which refactorings to use has already been determined, in addition to choosing the “correct” options for refactorings (should there be options). The hard work has been done; $\mathcal{R}^2$ eliminates the errors and tedium of the process.

RQ2: Can $\mathcal{R}^2$ be applied to large programs? Table I clearly demonstrates that $\mathcal{R}^2$ can be applied to non-trivial programs. A number of these programs are more complicated than they appear as we explain below.

Recall makeVisitor invokes addParameter to the list of methods that are relatives of the method seed. Ideally, these relatives are descendant from a single root method (A.m in Figure 17a). This means that the $\mathcal{R}^2$ addParameter invokes the JDT change-method-signature refactoring once on A.m to add an extra parameter to all of its relatives B.m and C.m.

![Fig. 17. Method Seeds and Method Roots.](image)

In general, there can be multiple roots. Some may argue that using multiple roots is too general; only one root should ever be used. This is programmatically adjustable within $\mathcal{R}^2$. 

6 Some may argue that using multiple roots is too general; only one root should ever be used. This is programmatically adjustable within $\mathcal{R}^2$. 


a seed whose relatives are not descendant from a single root. This means that the $R^2$ addParameter invokes change-method-signature refactoring three times, once for each root E.m., F.m., G.m., to add an extra parameter to all relatives. Programmers who apply JDT refactorings manually would have to realize this situation and make these extra renames.

Now look at row/experiment A3 in Table I. Our tool created a Visitor for the printorder method in AHEAD. $R^2$ moved 276 methods into a Visitor, created no super-delegates, and applied one change-method-signature. The number of renames (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. $R^2$ created a Visitor for the getName method in JUnit. $R^2$ moved 5 methods into a Visitor, created no super-delegates, and applied 4 change-method-signatures. The reason for 4 is that there were 4 method roots for the given seed (Figure 17(b)). Thus, the number of renames performed is 9; 5 methods were moved, and 4 (root) delegates were renamed.

Finally, consider row/experiment W5. Our tool created a Visitor for the pruneOriginalCandidates method in RefactoringCrawler. $R^2$ moved 13 methods into a Visitor, where these methods had 7 “super” references and thus required a super-delegate for each to be created.

Removing a Visitor. Figure 18 lists the results of inverting (removing) the Visitors created in Table I.

Consider row/experiment A5. Our tool removed a Visitor of the reduce2Java in AHEAD. 47 visit methods were moved back to original classes. The number of inlines (54) was determined in this way: each visit method that is moved is inlined (47) and 7 super-delegates are also inlined. Only one had to be renamed to its original name (reduce2Java) and removed a Visitor-type parameter. That is because, by changing a root method’s signature, all of its descendants were updated. In addition, we turned off an inline precondition described in Section II-B for A4, A5, and C1. Note the difference in execution time between creating and removing a Visitor is due to different numbers and types of refactorings.

### C. Perspective and Future Work

Our experiments demonstrate that $R^2$ scripts (a) improve productivity and (b) are scalable to large programs. The $R^2$ idea is portable to other Java IDEs such as IntelliJ IDEA,
NetBeans, and Visual Studio; it is not limited to Eclipse (or Java, for that matter). Practical issues still remain.

1) Correctness of IDE-supplied refactorings remains a serious problem. Look at column \# of Errors in Table I. It shows A5 executed 96 JDT refactorings and introduced 100 errors (in Juno 4.2.2) that we had to fix manually. It took two years for the current version of JDT (Luna 4.4.1) to resolve these bugs (our bug reports are available at [33]).

2) IDE-supplied refactorings should be expressive and easy to understand. Odd or limited refactorings (as discussed in Section II) preclude or otherwise distort elegant scripts. An expressive basis set of primitive refactorings to be supported by IDEs remains an open problem [34], [35].

3) Refactoring speed is important as programmers expect instantaneous results. Look at the Time columns of Table I and Figure [13]. Many executions are over 20 seconds; the largest is 10 minutes. We are building a new refactoring engine that executes \( R^2 \) scripts almost instantaneously [36].

4) We are also in the process of writing up a user study using \( R^2 \) that shows students find it easy to use.

VI. RELATED WORK

Writing program transformations is a non-trivial exercise as research has shown [5], [8], [20], [37]–[42]. Prior work introduced a number of impressive metaprogramming languages such as ASF+DSF [17], iXJ [10], JunGL [18], Parlanse [9], Rascal [13], Refacola [16], SOUL [15], Stratego [11], Tom [8], and TXL [12]. None match our requirements.

There are two primary distinctions between \( R^2 \) and prior work. First, \( R^2 \) uses the base language – the language in which programs to be refactored are written – as the scripting language. Interestingly, the base and scripting language are identical only in Wrangler [14]; all others use a different scripting language (possibly even a different programming paradigm) than the base. The second is whether a user has to implement primitive refactoring in order to script them. Since writing primitive refactorings (e.g., rename, move, change-method-signature) is non-trivial, it is important to distinguish approaches that can leverage existing refactoring engines from those where primitives need to be written by users. To the best of our knowledge, only SOUL and Rascal (besides \( R^2 \)) satisfy the second criterion.

JunGL and Refacola are DSLs specialized for scripting refactorings. JunGL is an ML-style functional language implemented on the .NET platform and targets C#. JunGL facilitates AST manipulation with higher order functions and tree pattern matching. It also has 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 [41], DMS [9], SmaCC [19], Wrangler [14], and XT [11]. Wrangler, mentioned earlier, is a tool (refactoring framework) implemented in Erlang which is also the base language. Wrangler supports refactoring commands for locating program elements and provides a custom DSL to execute the commands.

Like \( R^2 \), Rascal [13] also uses JDT refactorings, which are available as APIs in the Rascal JDT Refactoring 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 be automated.

SOUL [15] uses declarative metaprogramming to define pattern and their constraints in a language-independent manner. Their use of a variant of Prolog is elegant, as they tackle problems similar to \( R^2 \).

Moreover, \( R^2 \) deals with scripting high-level refactorings, not with recommending when and which refactorings to apply or detecting existing refactorings. There are excellent papers [45]–[62] on this, but all are orthogonal to the use and goals of \( R^2 \).

Finally, refactoring research has grown enormously in the last decade. Traditional refactorings improve design, like \( R^2 \). More recent refactorings improve non-functional qualities (e.g., energy consumption [63]), address more challenging languages (e.g., Yahoo! Pipes [64]), or use novel paradigms to check refactoring safety [65].

VII. CONCLUSIONS

Retrofitting 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 or no support for 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 may require knowledge of yet another programming language and the need to code primitive refactorings.

Our solution \( R^2 \) uses (1) Java as a scripting language, (2) \( R^2 \) objects are class, method, and field declarations of a Java program, and (3) \( R^2 \) methods are native JDT refactorings, primitive transformations, or our scripts. We used \( R^2 \) to automate 18 out of 23 classical design patterns, where each \( R^2 \) script is a compact Java method.

Our case study shows that \( R^2 \) refactoring scripts:

- save significant time for even relatively small refactorings (reducing to 24 seconds to run an \( R^2 \) script that introduces a Visitor with 13 methods), and
- can be applied to non-trivial programs (554 refactorings applied to a code base of 26K).

Next-generation refactoring engines should support refactoring scripts. We found that such scripts place a heavy demand on the correctness, expressiveness, and speed of IDE-provided refactorings. Whether off-the-shelf JDT (or other IDE refactoring engines) will meet these challenges remains to be seen. Nevertheless, \( R^2 \) takes us a step closer to this goal.

Acknowledgements. We thank Friedrich Steimann for his valuable comments on an early draft of this paper. We gratefully acknowledge support for this work by NSF grants CCF-1212683 and CCF-1439957.
REFERENCES