ICSME 2015, Bremen Germany

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, determine a precise sequence of refactorings to apply, and perform this sequence repetitively to a laborious degree. We designed, implemented, and evaluated *Reflective Refactoring* ($\mathcal{R}^2$), 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 $\mathcal{R}^2$ scripts and explain why the remaining are inappropriate for refactoring engines. We evaluate the productivity and scalability of $\mathcal{R}^2$ with a case study of 6 real-world applications. In one case, $\mathcal{R}^2$ automatically created a Visitor with 276 visit methods by invoking 554 Eclipse refactorings in 10 minutes – an achievement that could not be done manually. $\mathcal{R}^2$ 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* ($\mathcal{R}^2$), a Java package whose goal is to encode the construction of classical design patterns as Java methods. Using Eclipse *Java Development Tools (JDT)* [21], $\mathcal{R}^2$ 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 ($\mathcal{R}^2$) 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 the Java package, $\mathcal{R}^2$, with several novel features to improve refactoring technology. $\mathcal{R}^2$ objects are Java entity declarations and $\mathcal{R}^2$ methods are JDT refactorings, primitive $\mathcal{R}^2$ transformations, $\mathcal{R}^2$ pattern scripts, and program element navigations (i.e., $\mathcal{R}^2$ object searches).
- **Generality.** We encoded 18 out of 23 Gang-of-Four design patterns [3], inverses, and variants as short Java methods in $\mathcal{R}^2$, several of which we illustrate. This shows that $\mathcal{R}^2$ can express a wide range of patterns.
- **Implementation.** $\mathcal{R}^2$ is also an Eclipse plugin that leverages existing JDT refactorings and enables programmers to script many high-level patterns elegantly.

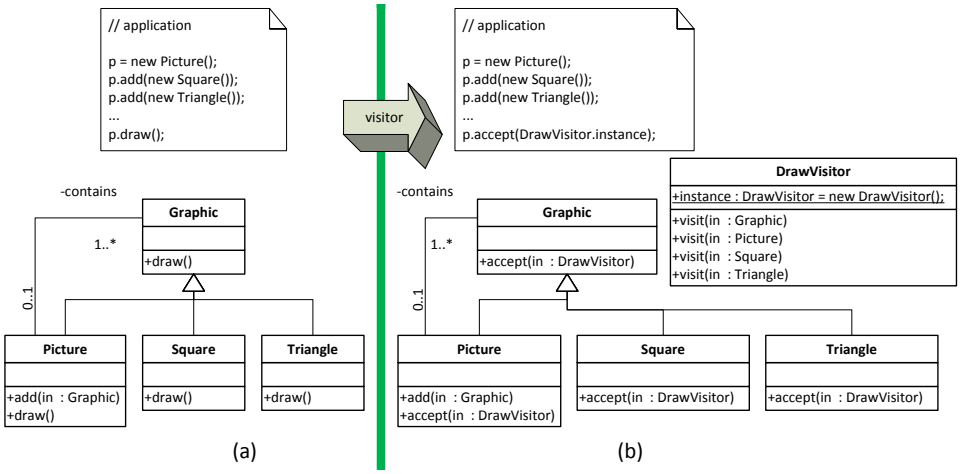

Fig. 1. A Visitor Pattern Refactoring.

- **Evaluation.** A case study shows the productivity and scalability of $\mathcal{R}^2$. We applied a 20-line $\mathcal{R}^2$ script to retrofit 52 pattern instances into 6 real-world applications. One case invoked 554 refactorings, showing that $\mathcal{R}^2$ 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 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.

Mechanics. To create a Visitor for the `draw` method (Figure 1b), 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:

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

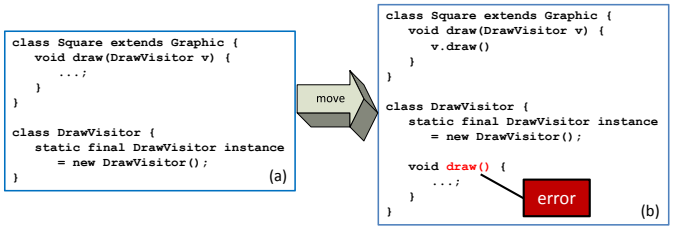

Fig. 2. A JDT Refactoring Being Too Smart.

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 $\mathcal{R}^2$; 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 3a 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).

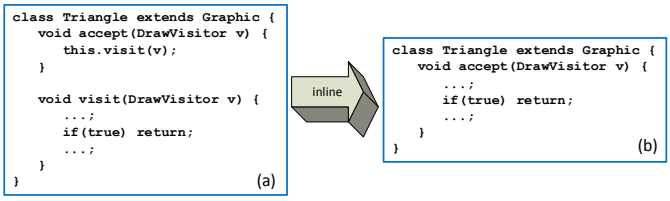

Fig. 3. Restriction of JDT `inline` Refactoring.

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_x`θ(), whose body calls `super.x()`; θ is just a random number to make the name of the manufactured method unique.[1,2]

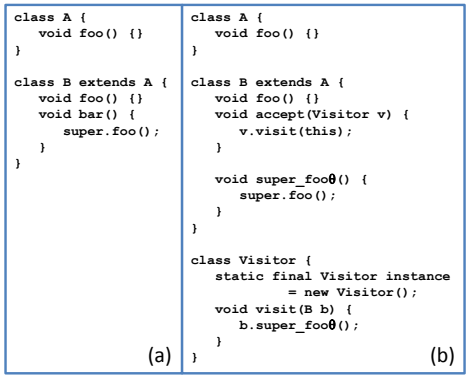

Fig. 4. Rewrite that Uses `super` Delegate.

[1]If `super.x()` returns a result of type `X`, `super_x`θ() also returns type `X`.

[2]A unique name is needed for a refactoring that "undoes" or "removes" a Visitor (Section III-B). It 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.

In Figure 4a, 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_foo`θ() 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`.

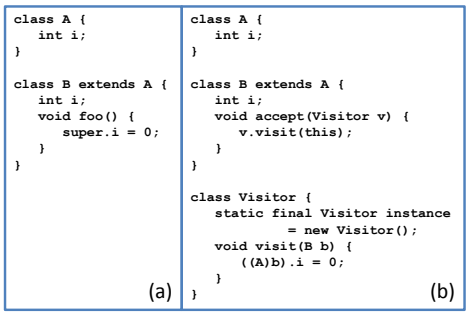

Fig. 5. `super` Field Access.

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 $\mathcal{P}$ be a JDT project. We leverage the idea of reflection; $\mathcal{R}^2$ defines class `RClass` whose instances are the class declarations in $\mathcal{P}$; instances of classes `RMethod` and `RField` are the method and field declarations of $\mathcal{P}$, and so on. When $\mathcal{P}$ is compiled, $\mathcal{R}^2$ 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 $\mathcal{P}$. *These tables are not persistent; they exist only when the JDT project for $\mathcal{P}$ is open.*

The fields of `RClass`, `RMethod`, `RField`, etc. – henceforth called $\mathcal{R}^2$ *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 $\mathcal{R}^2$ classes are JDT refactorings, simple $\mathcal{R}^2$ transformations, composite refactorings (our scripts), and ways to locate program elements (i.e., $\mathcal{R}^2$ objects).

Internally, we leveraged XML scripts which Eclipse uses only to replay refactoring histories. An $\mathcal{R}^2$ 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. $\mathcal{R}^2$ exposes every available JDT refactoring as a method and a few more (Section II). Overall, we changed `51` lines in `8` JDT internal packages; the $\mathcal{R}^2$ package consists of $\sim$ `5`K LOC.

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

A. Automating the Visitor Pattern

Visitor is fully automatable as an $\mathcal{R}^2$ 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` $\mathcal{R}^2$ 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 $\mathcal{R}^2$ script is indistinguishable from an existing JDT refactoring.[3]

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 7–8 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*.`instance` as the default extra argument. Lines 10–15 move each movable method to the Visitor class, leave behind a delegate, and rename each method to `visit`. Lines 17–18 collect delegate relatives and rename them to `accept`.[4] Line 20 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 7. 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 $\mathcal{P}$ 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`}.

[3]To add another script, its method is added to $\mathcal{R}^2$. Eclipse is then run with the updated $\mathcal{R}^2$.

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

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

Fig. 6. A `makeVisitor` Method.

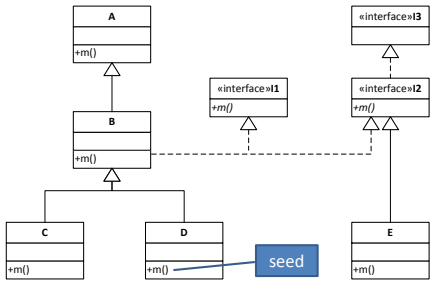

Fig. 7. Methods Altered by Change Signature.

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. $\mathcal{R}^2$ eliminates such errors by invoking a trustworthy $\mathcal{R}^2$ `getRelatives()` method.

B. Automating the Inverse Visitor

Figure 8 depicts a common scenario: An $\mathcal{R}^2$ 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.[5]

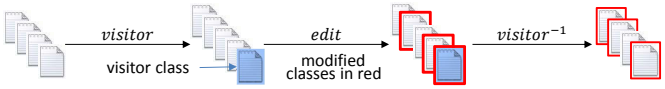

Fig. 8. A Common Programming Scenario.

[5]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.

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 13–14 inline `super`-delegates if they exist by replacing each call to `super_x`θ() 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.

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

Fig. 9. An `inverseVisitor` Method.

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`.

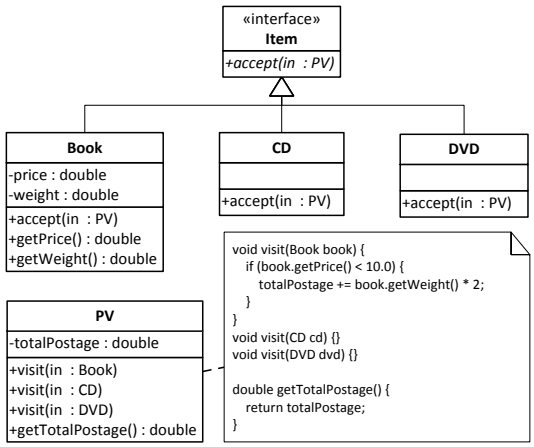

Fig. 10. Visitor with State.

The Visitor variant of Figure 10 requires a slight modification of our $\mathcal{R}^2$ `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.

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

Fig. 11. Another `inverseVisitor` Variant.

These examples illustrate the power of $\mathcal{R}^2$: (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. $\mathcal{R}^2$ 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). $\mathcal{R}^2$ *scripts for all of the 18 automatable patterns are listed in* [25]. We elaborate our key findings below.

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	✓		
Total	8	10	5

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

A. Fully Automatable Patterns

The Visitor pattern, 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 13b shows interface `AbstractFactory` that exposes factory methods for every `public` constructor of each `public` class in a given package: the package of Figure 13a contains classes `A` and `B`; the interface `AbstractFactory` is implemented by concrete factory class `ConcreteFactory` in Figure 13b. Figure 14 is the $\mathcal{R}^2$ method that produces a concrete factory for a package. A similar $\mathcal{R}^2$ script creates the `AbstractFactory` interface.

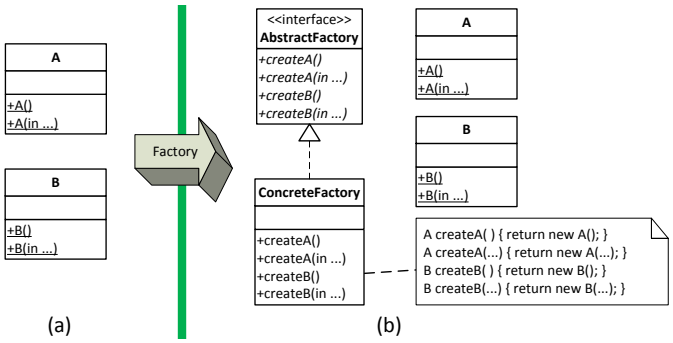

Fig. 13. Factory Pattern.

```
1   // member of RPackage class
2   RClass makeConcreteFactory(String factoryName) throws
        RException {
3       RClass factory = this.newClass(factoryName);
4
5       for(RClass c : this.getClassList()) {
6           if(c.isPublic())
7             for(RMethod m : c.getConstructorList())
8                 if(m.isPublic())
9                     factory.newFactoryMethod(m);
10      }
11
12      return factory;
13  }
```

Fig. 14. A `makeConcreteFactory` Method.

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 $\mathcal{R}^2$ `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 $\mathcal{R}^2$.

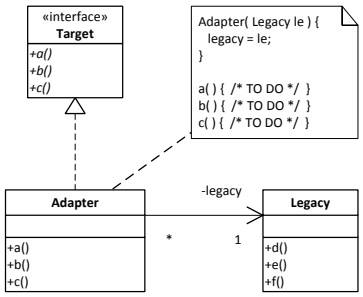

Fig. 15. Adapter Pattern.

```
1   // member of RInterface class
2   RClass makeAdapter(RClass adaptee, String adapterName)
        throws RException {
3     RClass adapter = getPackage().newClass(adapterName);
4
5     RField f = adapter.newField(adaptee, "legacy");
6     adapter.newConstructor(f);
7
8     for(RMethod m : this.getMethodList())
9       adapter.newMethod(m);
10
11    adapter.setInterface(this);
12
13    return adapter;
14  }
```

Fig. 16. A `makeAdapter` Method.

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 Delegate* 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`.

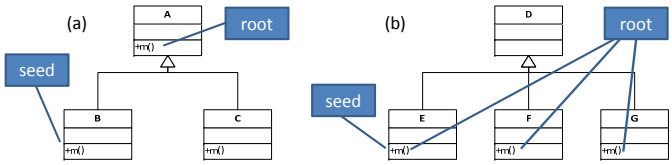

Fig. 17. Method Seeds and Method Roots.

In general, there can be multiple roots.[6] Figure 17b shows

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

TABLE I
APPLICATIONS AND VISITOR PATTERN RESULTS.

Seed ID	Subject (Ver#, LOC, #Tests)	Seed Method Name	Super Delegate	Change Signature	Move	Rename	# of Refactorings	Time	# of Errors (now fixed by JDT patches)
A1		getAST_Exp	0	26	26	52	104	72s	26
A2	**A**HEAD	getExpression	0	17	17	34	68	54s	17
A3	jak2java [26]	printorder	0	1	276	277	554	604s	0
A4	(130320, 26K, 75)	reduce2Ast	1	1	29	30	60	46s	23
A5		reduce2Java	7	1	47	48	96	84s	100
C1		encode	0	1	2	3	6	5s	27
C2	Commons	getCharset	0	4	4	8	16	13s	0
C3	**C**odec [28]	getDefaultCharset	0	4	4	8	16	12s	0
C4	(1.8, 16K, 6103)	getEncoding	0	2	4	6	12	10s	3
C5		isInAlphabet	0	1	2	3	6	5s	2
I1		getDefaultEncoding	0	1	1	2	4	4s	0
I2	Commons	getEncoding	0	1	1	2	4	5s	0
I3	**IO** [29]	getFileFilters	0	1	2	3	6	5s	0
I4	(2.4, 24K, 810)	getSize	0	1	1	2	4	5s	0
I5		setFileFilters	0	1	2	3	6	5s	0
J1		countTestCases	1	1	7	8	16	13s	1
J2		failedTest	0	1	1	2	4	3s	0
J3	**J**Unit [30]	getName	0	4	5	9	18	40s	2
J4	(4.11, 23K, 2807)	run	2	1	9	10	20	20s	4
J5		testCount	0	1	1	2	4	4s	2
Q	**Q**uark [26] (1.0, 575, 9)	apply	0	1	7	8	16	13s	0
W1		computeLikeliness	0	1	13	14	28	24s	14
W2	Refactoring	extractFullyQualifiedParentName	0	1	1	2	4	6s	0
W3	Cra**w**ler [31]	isRename	0	1	12	13	26	26s	10
W4	(1.0.0, 7K, 15)	pruneFalsePositives	1	1	4	5	10	10s	1
W5		pruneOriginalCandidates	7	1	13	14	28	25s	4

a seed whose relatives are not descendant from a single root. This means that the $\mathcal{R}^2$ `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 **A**3 in Table I. Our tool created a Visitor for the `printorder` method in **A**HEAD. $\mathcal{R}^2$ 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 **J**3. $\mathcal{R}^2$ created a Visitor for the `getName` method in **J**Unit. $\mathcal{R}^2$ 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 17b). Thus, the number of `renames` performed is `9`; `5` methods were moved, and `4` (root) delegates were renamed.

Finally, consider row/experiment **W**5. Our tool created a Visitor for the `pruneOriginalCandidates` method in RefactoringCra**w**ler. $\mathcal{R}^2$ 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 **A**5. Our tool removed a Visitor of the `reduce2Java` in **A**HEAD. `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 **A**4, **A**5, and **C**1. Note the difference in execution time between creating and removing a Visitor is due to different numbers and types of refactorings.

Seed ID	Change Signature	Move	Inline	Rename	# of Refactorings	Time
A1	26	26	26	26	104	97s
A2	17	17	17	17	68	61s
A3	1	276	276	1	554	395s
A4	1	29	30	1	61	42s
A5	1	47	54	1	103	70s
C1	1	2	2	1	6	5s
C2	4	4	4	4	16	15s
C3	4	4	4	4	16	15s
C4	2	4	4	2	12	10s
C5	1	2	2	1	6	5s
I1	1	1	1	1	4	4s
I2	1	1	1	1	4	4s
I3	1	2	2	1	6	6s
I4	1	1	1	1	4	5s
I5	1	2	2	1	6	5s
J1	1	7	8	1	17	13s
J2	1	1	1	1	4	4s
J3	4	5	5	4	18	22s
J4	1	9	11	1	22	18s
J5	1	1	1	1	4	5s
Q	1	7	7	1	16	11s
W1	1	13	13	1	28	22s
W2	1	1	1	1	4	8s
W3	1	12	12	1	26	37s
W4	1	4	5	1	11	14s
W5	1	13	20	1	35	34s

Fig. 18. Inverse Visitor Results.

C. Perspective and Future Work

Our experiments demonstrate that $\mathcal{R}^2$ scripts (a) improve productivity and (b) are scalable to large programs. The $\mathcal{R}^2$ 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 18. Many executions are over 20 seconds; the largest is 10 minutes. We are building a new refactoring engine that executes $\mathcal{R}^2$ scripts almost instantaneously [36].

4) We are also in the process of writing up a user study using $\mathcal{R}^2$ 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 $\mathcal{R}^2$ and prior work. First, $\mathcal{R}^2$ 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 refactorings 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 $\mathcal{R}^2$) 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 $\mathcal{R}^2$, Rascal [13] also uses 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 be automated.

SOUL [15] uses 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 $\mathcal{R}^2$.

Moreover, $\mathcal{R}^2$ deals with scripting high-level refactorings, *not* with recommending when and which refactorings to apply or detecting existing refactorings. There are excellent papers [43]–[62] on this, but all are orthogonal to the use and goals of $\mathcal{R}^2$.

Finally, refactoring research has grown enormously in the last decade. Traditional refactorings improve design, like $\mathcal{R}^2$. 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 $\mathcal{R}^2$ uses (1) Java as a scripting language, (2) $\mathcal{R}^2$ objects are class, method, and field declarations of a Java program, and (3) $\mathcal{R}^2$ methods are native JDT refactorings, primitive transformations, or our scripts. We used $\mathcal{R}^2$ to automate 18 out of 23 classical design patterns, where each $\mathcal{R}^2$ script is a compact Java method.

Our case study shows that $\mathcal{R}^2$ refactoring scripts:

- save significant time for even relatively small refactorings (reducing to 24 seconds to run an $\mathcal{R}^2$ 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, $\mathcal{R}^2$ 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

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

[2] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP*, 1993.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] W. G. Griswold, "Program Restructuring as an Aid to Software Maintenance," Ph.D. dissertation, University of Washington, 1991.

[5] B. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

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

[7] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," in *ASE*, 1999.

[8] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom: Piggybacking Rewriting on Java," in *RTA*, 2007.

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution," in *ICSE*, 2004.

[10] M. Boshernitsan and S. L. Graham, "iXj: Interactive Source-to-Source Transformations for Java," in *OOPSLA Companion*, 2004.

[11] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A Language and Toolset for Program Transformation," *Science of Computer Programming*, Jun. 2008.

[12] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, Aug. 2006.

[13] M. Hills, P. Klint, and J. J. Vinju, "Scripting a Refactoring with Rascal and Eclipse," in *WRT*, 2012.

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

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

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

[17] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser, "The ASF+SDF Meta-environment: A Component-Based Language Development Environment," in *CC*, 2001.

[18] M. Verbaere, R. Ettinger, and O. de Moor, "JunGL: a Scripting Language for Refactoring," in *ICSE*, 2006.

[19] 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.

[20] L. Frenzel, "The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs," https://eclipse.org/articles/Article-LTK/ltk.html.

[21] "Eclipse Java development tools (JDT)," http://www.eclipse.org/jdt/.

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

[23] "Eclipse Luna," https://eclipse.org/luna/.

[24] J. Sugrue, "Design Patterns Uncovered: The Visitor Pattern," https://dzone.com/articles/design-patterns-visitor, 2010.

[25] "R2 Design Pattern Scripts," http://www.cs.utexas.edu/~jongwook/r2designpatternscripts.html/.

[26] D. Batory, "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite," in *GTTSE*, 2005.

[27] "Dialect user's guide," 1990.

[28] "Apache Commons Codec," https://commons.apache.org/proper/commons-codec/.

[29] "Apache Commons IO," https://commons.apache.org/proper/commons-io/.

[30] "JUnit," http://junit.org/.

[31] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *ECOOP*, 2006.

[32] "Eclipse Juno," https://eclipse.org/juno/.

[33] "JDT Refactoring Bugs," http://www.cs.utexas.edu/~jongwook/jdtrefactoringbugs.html/.

[34] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," in *ICSE*, 2012.

[35] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," in *ICSE*, 2009.

[36] J. Kim, D. Batory, and D. Dig, "Design Pattern Refactoring by Pretty-Printing," in *submitted for publication*, 2015.

[37] M. Schaefer and O. de Moor, "Specifying and Implementing Refactorings," in *OOPSLA*, 2010.

[38] D. Roberts, "Practical Analysis for Refactoring," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1999.

[39] A. Garrido, "Program Refactoring in the Presence of Preprocessor Directives," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

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

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

[42] M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju, "Using The Meta-Environment for Maintenance and Renovation," in *CSMR*, 2007.

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

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

[45] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, May 2009.

[46] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "On the Existence of High-Impact Refactoring Opportunities in Programs," in *ACSC*, 2012.

[47] H. Melton and E. Tempero, "Identifying Refactoring Opportunities by Identifying Dependency Cycles," in *ACSC*, 2006.

[48] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling Manual and Automatic Refactoring," in *ICSE*, 2012.

[49] D. Silva, R. Terra, and M. T. Valente, "Recommending Automated Extract Method Refactorings," in *ICPC*, 2014.

[50] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "Recommending Refactorings to Reverse Software Architecture Erosion," in *CSMR*, 2012.

[51] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending Move Method Refactorings Using Dependency Sets," in *WCRE*, 2013.

[52] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, Oct. 2014.

[53] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering*, Jul. 2014.

[54] N. A. Milea, L. Jiang, and S.-C. Khoo, "Scalable Detection of Missed Cross-function Refactorings," in *ISSTA*, 2014.

[55] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. RSSE, 2014.

[56] Q. D. Soetens, J. Pérez, and S. Demeyer, "An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings," in *ICSM*, 2013.

[57] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, "Comparison of Similarity Metrics for Refactoring Detection," in *MSR*, 2011.

[58] M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental Assessment of Software Metrics Using Automated Refactoring," in *ESEM*, 2012.

[59] N. A. Milea, L. Jiang, and S. Khoo, "Vector Abstraction and Concretization for Scalable Detection of Refactorings," in *FSE*, 2014.

[60] G. Bavota, S. Panichella, N. Tsantalis, M. D. Penta, R. Oliveto, and G. Canfora, "Recommending Refactorings based on Team Co-Maintenance Patterns," in *ASE*, 2014.

[61] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y. Guéhéneuc, "Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory," in *ICSM*, 2010.

[62] G. Bavota, A. D. Lucia, and R. Oliveto, "Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures," *Journal of Systems and Software*, Apr. 2011.

[63] C. Sahin, L. Pollock, and J. Clause, "How Do Code Refactorings Affect Energy Usage?" in *ESEM*, 2014.

[64] K. T. Stolee and S. Elbaum, "Refactoring Pipe-like Mashups for End-User Programmers," in *ICSE*, 2011.

[65] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the Impact of Refactoring Operations on Code Quality Metrics," in *ICSME*, 2014.