Design Pattern Refactoring by Pretty-Printing

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*—Most design patterns in the Gang-of-Four text can be written as a refactoring script – a programmatic sequence of refactorings. $\mathcal{R}^3$ is a new Java refactoring engine based on pretty-printing. It builds a main-memory, non-persistent database to encode containment relationships among Java entity declarations (e.g., packages, classes, methods), language features such as inheritance and modifiers, and precondition checks for each declaration. $\mathcal{R}^3$ design pattern scripts modify the database and do not modify *Abstract Syntax Trees (ASTs)*. While classical refactoring engines transform source code or ASTs directly, $\mathcal{R}^3$ produces refactored code by displaying the contents of the modified database when pretty-printing ASTs. $\mathcal{R}^3$ performs comparable precondition checks and code changes to that of the *Eclipse Java Development Tools (JDT)* refactorings but $\mathcal{R}^3$'s codebase is about half the size of JDT refactorings and runs an order of magnitude faster. $\mathcal{R}^3$ refactorings pass relevant regression tests in JDT and fix JDT refactoring bugs that we found.**

I. Introduction

Refactoring is a core technology in software development. All major *Integrated Development Environments (IDEs)* offer some form of refactoring support and refactoring is central to popular software design movements, such as Agile [1] and Extreme Programming [2]. In the last decade, refactoring tools have revolutionized how programmers design software. They have enabled programmers to continuously explore the design space of large codebases, while preserving existing behavior. Modern IDEs such as Eclipse, NetBeans, IntelliJ IDEA, and Visual Studio incorporate refactoring in their top menu and often compete on the basis of refactoring support.

Despite vast interest and progress, a key functionality that many researchers have recognized to be missing in IDEs is scripting [3–5]. Most design patterns in the Gang-of-Four text [6] can be expressed as a *refactoring script* – a programmatic sequence of refactorings [7], [8]. Adding and removing design patterns manually is tedious, repetitious, error prone, and often too difficult to do – try creating a Visitor with over `20` methods; the benefits of scripting become clear.

We recently added scripting to Eclipse JDT [9], exposing the core declarations of a Java program (packages, classes, methods, etc.) as objects whose methods are JDT refactorings. Refactoring scripts that add or remove design patterns are short Java methods. Our tool, called $\mathcal{R}^2$, is detailed in the next section. $\mathcal{R}^2$ is a plug-in that uses the *JDT Refactoring Engine (JDTRE)* as it represents state-of-the-practice in refactoring.

Experiments with $\mathcal{R}^2$ revealed JDTRE is ill-suited for scripting for three reasons:

- **Reliability.** JDTRE is buggy [10–12]. We found over `25` new bugs to date, but only a fraction have been fixed in the latest version of JDT. Prior to the current release, one $\mathcal{R}^2$ script executed `6` JDT refactorings; the resulting program had `27` compilation errors. Another script invoked `96` refactorings, producing a program with `100` compilation errors. These particular errors are now fixed, but we are constantly discovering more. Worse is waiting months or years for a repair [12]. We rediscovered an old bug that took `5` years to be fixed [13].

  Note. We are not in the position to repair JDTRE. There is no reason for us to believe our patches would be accepted. We report bugs as others do.
- **Expressivity.** We found the need for additional primitive refactorings and to repair existing refactorings. Examples: JDTRE refuses to move methods that include `super` keyword(s); collecting methods that reference `super` into a Visitor class is really useful. We also had to turn-off parameter optimization in order for JDT refactorings to produce correct design patterns [9].
- **Speed.** JDTRE is slow. While a single JDT refactoring has acceptable speed, executing many is not. $\mathcal{R}^2$ scripts execute a series of JDT refactorings, thus exposing JDTRE's Achilles heel. $\mathcal{R}^2$ scripts that invoke about `20` refactorings take over `10` seconds. One case invoked `554` refactorings and took `9` minutes to execute. Programmers expect refactorings to be instantaneous.

We concluded that a radically different approach to build refactoring engines for scripting was needed to remove these problems. Our novel solution, called $\mathcal{R}^3$, computes a "view" of a program. Refactorings do not modify ASTs in $\mathcal{R}^3$. We create a database of program elements (classes, methods, fields, etc.), their associations, and primitive properties for precondition checks. Precondition checks consult this database; refactorings alter the database. *Pretty-printing ASTs to display the contents of the database produces refactored code.* Doing this yields a significant increase in refactoring speed, a much smaller codebase, and comparable reliability to JDTRE.

The contributions of this paper are:

- A novel foundation using database+pretty printing for designing a new generation of refactoring engines that allows scripting,
- An efficient way to evaluate refactoring preconditions. We harvest boolean properties of ASTs during database creation and let precondition checks consult their values,
- Implementation. $\mathcal{R}^3$'s codebase is a mere 4K LOC and does not use Eclipse program transformation utilities,

- An empirical evaluation of $\mathcal{R}^3$ on 6 case studies executed 52 scripts. $\mathcal{R}^3$ runs at least 10× faster, in two cases 290× faster than JDTRE. Further, $\mathcal{R}^3$ is at least as reliable as JDTRE. $\mathcal{R}^3$ passes all relevant JDTRE regression tests and fixes problems still latent in the latest release of JDT.

II. A Recap of $\mathcal{R}^2$

It is well-known that many, but not all, classical design patterns can be expressed by a series of refactorings [7], [8]. In prior work, we leveraged the JDTRE to provide a practical means to implement such scripts [9].

$\mathcal{R}^2$ is a Java package. Its objects correspond to Java entity declarations such as packages, classes, methods, etc. in a JDT project. The program in Figure 1 has 7 $\mathcal{R}^2$ objects: 3 classes `Graphic`, `Square`, `Picture` and 4 methods `Graphic.draw`, `Square.draw`, `Picture.add`, `Picture.draw`.

```
class Graphic {
 void draw { … }
}

class Square extends Graphic {
 void draw() { … }
}

class Picture extends Graphic {
 void add(Graphic g) { … }
 void draw() { … }
}
```

Fig. 1: A Java Program

Methods of $\mathcal{R}^2$ objects are JDT refactorings or object searches. Table I lists a few methods that can be performed on $\mathcal{R}^2$ class and method objects.

$\mathcal{R}^2$ Type	Method Name	Semantics
RClass	rename	rename class
	move	move class to another package
	getSuperClass	return the $\mathcal{R}^2$ object that is the superclass of the class
	getName	return the name of the class
	getAllMethods	return a list of $\mathcal{R}^2$ objects that are all methods of the class
	newMethod	add a new method to the class
	newField	add a new field to the class
RMethod	rename	rename method
	move	move method to new class
	addParameter	add a parameter with default value
	getRelatives	return a list of $\mathcal{R}^2$ objects of methods with same signature
	getName	return the name of the method
	moveAndDelegate	move a method and leave behind a delegate

TABLE I. Methods of $\mathcal{R}^2$.

$\mathcal{R}^2$ refactoring scripts are short Java methods; writing $\mathcal{R}^2$ scripts is just like writing regular Java code. Here are two examples.

Figure 2 is an $\mathcal{R}^2$ script that creates an adapter class `N`. Here is how it works: a programmer identifies an interface declaration that is to be implemented. Let `this` be its $\mathcal{R}^2$ object (which is provided by the Eclipse GUI). Let `c` be an $\mathcal{R}^2$ class object (the class to adapt), and let `N` be the name of adapter class. `makeAdapter` works by creating a class `N` in the package of class `c` that implements interface `this` (Line 5). A field named `adaptee` of type `c` is created and a constructor is added to initialize this field (Line 8). A stub is generated for each method in interface `this` (Line 11). The created class `N` is returned as the result of `makeAdapter`.

Figure 3 shows an $\mathcal{R}^2$ script for creating a Visitor design pattern. Here is how it works: A programmer identifies a method (called a *seed*) in a class hierarchy that s/he wants to

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

Fig. 2: $\mathcal{R}^2$ `makeAdapter` Method.

create a Visitor. Let `this` be its $\mathcal{R}^2$ object. A Visitor is created by invoking `this.makeVisitor(N)` where `N` is the name of the Visitor class to be created. `makeVisitor` gets the package of `this`, creates a class with name `N` in that package, and makes that class a singleton (Line 6). Next, all methods with the same signature as `this` are collected onto a list. To every method on the list, a parameter of type `N` is added whose default value is the singleton field of class `N` (Line 9), and then they are renamed to `accept`. Only movable methods (`abstract` or `interface` methods cannot be moved) are relocated to class `N`, leaving behind a delegate. All methods in the Visitor class are renamed to `visit` (Line 15). `makevisitor` returns the `RClass` object for class `N`.

```
1  // member of RMethod class
2  RClass makeVisitor(String N)
3  {
4    RPackage pkg = this.getPackage();
5    RClass vc = pkg.newClass(N);
6    RField singleton = vc.addSingleton();
7
8    RMethodList methodList = this.getRelatives();
9    int index = methodList.addparameter(singleton);
10   methodList.rename("accept");
11
12   for(RMethod m : this.methodList)
13     m.moveAndDelegate(index);
14
15   vc.getAllMethods().rename("visit");
16
17   return vc;
18 }
```

Fig. 3: $\mathcal{R}^2$ `makeVisitor` Method.

We implemented 18 of the 23 design patterns in the Gang-of-Four text [6] using $\mathcal{R}^2$. Eight patterns (including Visitor) are fully automatable. Another ten are partially automatable. This includes Adapter, where only stubs are generated. The bodies of stub methods must be provided manually as they require semantic knowledge. Some of the remaining patterns are automatable, such as State and Mediator,[1] while others, such as Façade and Iterator, seem to be so application-specific that little or nothing is reusable [9].

A good idea about $\mathcal{R}^2$ was using Java as a scripting language for refactorings. Writing $\mathcal{R}^2$ scripts is like writing regular Java code (as our examples suggest). Thus, a

[1] State is a typical MDE application [14]. Mediator is the essence of GUI builders. Neither seem appropriate to include in a refactoring engine.

programmer does not need to learn a new *Domain Specific Language (DSL)* for program transformations. In retrospect, a bad idea was using JDT as $\mathcal{R}^2$'s refactoring engine. As said in the Introduction: JDTRE has serious issues on reliability, expressivity, and speed. It is clear that JDTRE was never designed for scripting refactorings. $\mathcal{R}^3$, discussed next, keeps the good idea of $\mathcal{R}^2$ and is our replacement for JDTRE.

III. $\mathcal{R}^3$ Concepts

A. Relativistic Displays

Elementary physics inspired $\mathcal{R}^3$. A physical object appears different depending on the perspective from which it is viewed. Silhouette portraits of people are different from frontal portraits. Changing perspectives in physics is a *coordinate transformation* that *preserves object properties*.

To see the relevance to *Object-Oriented (OO)* languages, we strip away OO notation. A method implements an *absolute function* (the reason for 'absolute' is explained shortly) where all method parameters are explicit as they would be in a C-language declaration. Figure 5a is the signature of an absolute function `foo` with three parameters whose types are `B`, `C`, `D`.

If we were to 'view' `foo` as a member of class `B`, its signature would appear as in Figure 5b; the `B` parameter becomes `this` and is otherwise not shown. If `foo` were a member of class `C`, its signature would appear as Figure 5c, where the `C` parameter becomes `this` and is otherwise not shown. We say the *natural homes* of an absolute function are its parameter types. The natural homes for method `foo` are `B`, `C`, `D`. If we were to view `foo` as a member of class `E`, not a natural home, it would appear as the `static` method of Figure 5d with no implicit parameters.

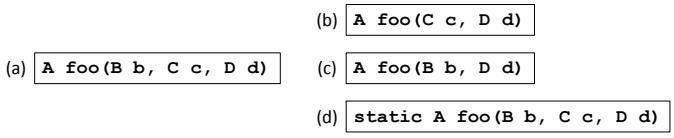

Fig. 5: An Absolute Function and its Relative Methods.

Connecting these ideas to refactoring, recall *Intentional Programming (IP)* [15]. IP is a structure editor whose ASTs could be adorned with different pretty-print methods. Figure 6a depicts an AST of an `if` statement. This AST could be pretty-printed in C-programming language syntax (Figure 6b), in Pascal syntax (Figure 6c), or even as a flowchart (Figure 6d). How an AST is displayed is under the control of a programmer [16]. For every distinct display, each AST node has a distinct rendering (pretty-print) method for it.

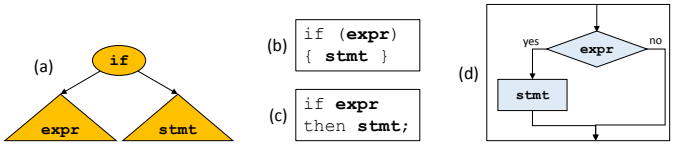

Fig. 6: IP Tree and its Different Views.

Observe that an AST is *never* altered by a display. The AST is 'absolute' or immutable; it appears different *relative to the language or modularity perspective* from which it is displayed. The move-method refactoring, which is what Figure 5 is about, is a coordinate transformation for software; it should preserve the semantic properties of a program. The same holds for other primitive refactorings in $\mathcal{R}^3$.

Similar ideas apply to comments and annotations in source code; they should be preserved and transformed too.

> **Note.** $\mathcal{R}^3$ presently does not transform comments or support annotation refactorings. $\mathcal{R}^3$ *does* preserve comments in refactored source code.

B. $\mathcal{R}^3$ Database

$\mathcal{R}^3$ maintains an internal, non-persistent database to remember changes in perspective. While $\mathcal{R}^3$ parses compilation units in a program, it creates relational database tables for all declaration types in a program. Each tuple of the `RClass` table represents a distinct class declaration in the program. Among `RClass` attributes is a pointer to the AST of that class. Each tuple of the `RMethod` table represents a distinct method (or rather, an absolute function) declaration in the program. Each `RMethod` tuple points to the AST of its method and to the `RClass` tuple in which that method is a member. Similarly, there are tables for package declarations (`RPackage`), field (`RField`), etc. There are no tables for Java executable statements or expressions; only classes, interfaces, fields, methods, and parameters, as these are the focus of design patterns and almost all classical refactorings.

Program source is compiled into ASTs which are traversed to populate $\mathcal{R}^3$ tables. Figure 4 shows the basic set-up with a few attributes per table. Three `RClass` tuples (`Graphic`, `Square`, `Picture`) and four `RMethod` tuples (`Graphic.draw`, `Square.draw`, `Picture.add`, `Picture.draw`) are created and linked to the `RClass` tuple for which it is a member. Each tuple has these attributes: an AST pointer, a database-wide unique key (e.g., `clsID` of `RClass`),[2] a simple (unqualified) name, a containment relationship, a class and interface inheritance relationship, and so on.

Basic refactorings are elementary updates to this database. Renaming a method updates the `name` field of that method's $\mathcal{R}^3$ tuple. Moving a method to another class updates the attribute of that method's $\mathcal{R}^3$ tuple to point to its new class. It is only when an AST is rendered (displayed) that the information in the $\mathcal{R}^3$ database is used. When a method's AST is displayed, the name of the method is extracted from the method's $\mathcal{R}^3$ tuple.

When a class is displayed, the tuples of the fields, methods, constructors, etc. that belong to it are extracted from the database. The ASTs of these tuples are then displayed, relative to their new class home. Figure 7 sketches the `RClass` display method: it first prints the current name of the class, its superclass, and `implements` clause, all names obtained from the database. Then each member that is assigned to that class is displayed, following by the display of the closing brace '}'. $\mathcal{R}^3$ preserves the original order in which members appeared, and reproduces this order for ease of subsequent reference by programmers.

[2] $\mathcal{R}^3$ keys are conceptually similar to *locked bindings* in [17] but are used in other ways; see Section IV-B.

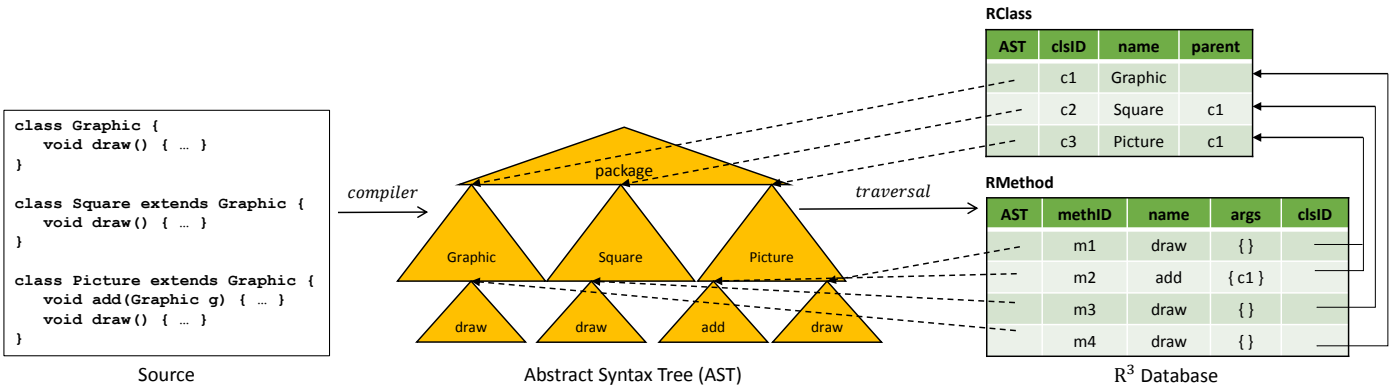

Fig. 4: $\mathcal{R}^3$ Database.

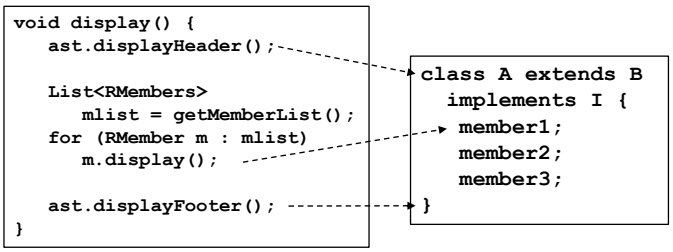

Fig. 7: `RClass` Display Method.

$\mathcal{R}^3$ is not a mere reimplementation of IP. IP displays entire trees; $\mathcal{R}^3$ integrates a database of program facts and the display of disconnected ASTs to yield a rendering that gives the appearance of a single refactored program.

Rendering displays is fast and less involved than updating ASTs and moving AST subtrees from one parent to another. Consider the changes that are needed when absolute method `foo` (Figure 5a) is moved from class `B` to `C`. All invocations of `foo`, say `b.foo(c,d)`, must be altered to `c.foo(b,d)`. A rendering simply changes the order in which arguments are displayed; it is more work to consistently update pointers when making this change to an AST.

In program transformation systems, refactorings modify ASTs. Only when all changes have been made are ASTs pretty-printed to code. $\mathcal{R}^3$ eliminates AST manipulation. $\mathcal{R}^3$ still needs to *create* trees, say when new program elements are created, but other than that, $\mathcal{R}^3$ does not manipulate ASTs. As we report later, a consequence is that the codebase for $\mathcal{R}^3$ is substantially smaller and simpler than that for program transformation systems.

C. Primitive Refactorings

We now explain some representative primitive refactorings to see how they are handled in $\mathcal{R}^3$. We partition our discussion refactorings into two segments: changes to the database (considered in this section) and checking preconditions (discussed in the next section).

1) Rename Method: Rename-method modifies the `name` field of the method's `RMethod` tuple. This refactoring, like others discussed shortly, have a database-transaction quality. Consider a class hierarchy where all classes have their own method `n`. To rename `n` to `r` can be expressed as a loop, where `getRelatives()` finds all overriding/overridden methods with the same signature as `n`:

```
for (RMethod m : n.getRelatives()) {
    m.rename("r");
}
```

Until the loop is finished, not all methods will be renamed and preserving program semantics is not guaranteed. We have not yet decided on a final interface for $\mathcal{R}^3$ (currently it looks much like $\mathcal{R}^2$), but we do have in place an alternative that performs renames on sets of methods with identical signatures, and by being a set operation, does not expose a database to users that alters program semantics:

```
RMethodList list = n.getRelatives();
list.rename("r");
```

2) Change Method Signature: Change-method-signature adds, removes, and reorders method parameters. Encoded in the $\mathcal{R}^3$ database is a list of formal parameters for every method and for each parameter there may be a default value. Adding a parameter to a method simply adds the new parameter and its default value to the database. When the method is displayed, it is shown with its new parameter; method calls are displayed with its default argument.

Prior work [18], [19] found that highly-parameterized refactorings (name, parameter add/delete/reorder, exception, delegate) discourage the use of refactorings and make it harder to understand refactoring functionality. Accordingly, $\mathcal{R}^3$ has separate methods to add, remove, and reorder parameters. The code below finds the $\mathcal{R}^3$ tuple for a field (variable) with name `f` in class `C` of package `p`. The field's name serves as the name of the new parameter and a reference to that field is the parameter's default value. The new parameter, by default, becomes the last formal parameter of method `m`. Method `setIndex` makes it the first parameter of `m`, and `remove` removes the parameter:

```
RField v = RField.find("p", "C", "f");
RParameter newParam = m.addParameter(v);
newParam.setIndex(0);
m.remove(newParam);
// m is unchanged by the above code
```

Like method `rename`, there is an `RMethodList` version of these methods.

3) Move Method via Parameter: The core move-method of $\mathcal{R}^3$ changes the home class of a method `m`. Recall a home parameter is any parameter of `m`, and a home class is the class of a home parameter. Moving `m` to a home class simply updates `m`'s $\mathcal{R}^3$ tuple to point to the tuple of its home class. The code below moves method `m` to class `c` (presuming `c` is a home class):

```
m.move(c);
```

4) Move Method via Field: The move-via-field refactoring is illustrated in Figure 8. Method `m` in class `A` is moved to class `D` via field δ. A local invocation, `m(b)`, becomes δ`.m(this,b)`.

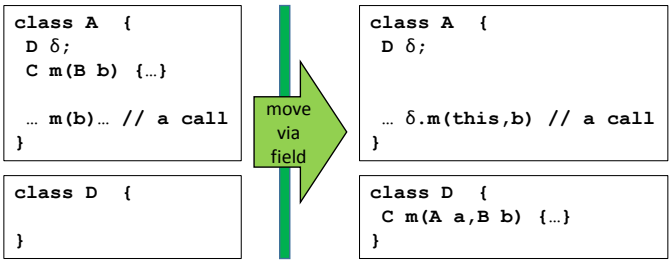

Fig. 8: Move via Field Refactoring.

Move-via-field is a composition of the $\mathcal{R}^3$ add-parameter and move-via-parameter refactorings. Let `a` be the `A`-type parameter of `m`. Parameter `d` of type `D` with default value `a`.δ is added to the absolute function of `m`, changing its signature from `m(A a, B b)` to `m(A a, B b, D d)` where the default value of parameter `d` is `a`.δ. When displayed in class `D`, the `D` argument becomes `this`; the display of a call exposes the new (default) argument. An $\mathcal{R}^3$ script that implements move-via-field is:

```
// member of RMethod class
void moveViaField(RField f) {
    RParameter newHome = addParameter(f);
    move(newHome.getType());
}
```

5) Introduce New Program Elements: $\mathcal{R}^3$ introduces complex new code declarations (classes, methods, fields, etc.) into an existing program by creating a compilation unit that has these declarations. The file is compiled and the database is updated with new declarations. These new declarations are embedded into the existing program via move refactorings or referenced by $\mathcal{R}^3$ refactorings. For example, if a new class `C` with a method `mul` is to be created in package `pkg` of JDT project `Prj`, a `String` of the class source is constructed and the method `createCU(String)` is invoked. `createCU` creates a Java file of its `String` input, the file is compiled, and its entries are added to the $\mathcal{R}^3$ database. Assuming package `pkg` exists and class `C` does not, the code below inserts two new tuples (one for class `C` and another for method `mul`) into the database:

```
String s = "package pkg;                  \n"+
           "class C {                     \n"+
           "  int mul() { return 7*57; }\n"+
           "}";
RPackage p = RProject.getPackage("Prj", "pkg");
RCompilationUnit cu = p.createCU(s);
RClass cls = p.getClass("C");
RMethod mth = cls.getMethod("mul");
```

Once the needed methods and fields are removed from a generated compilation unit, the unit can be marked deleted in the database. Its AST remains, but at pretty-printing time no text of the compilation unit is produced.

6) Other Refactorings: The current version of $\mathcal{R}^3$ supports the above refactorings, among others (e.g., pull-up and push-down), that are essential to refactoring scripts that introduce or remove design patterns from existing programs. $\mathcal{R}^3$ does not yet support all primitive refactorings in JDTRE. We see no limitation to add a full complement of refactorings.

To illustrate, nested classes generalize absolute functions in an interesting way. Figure 9a shows class `B` nested inside class `A`. Method `m` of class `B` has the absolute function:

```
void m(A a, B b) { a.i = a.i + b.j; }
```

Although `m()` displays without parameters inside `B`, it really has two implicit parameters: `this` (of type `B`) and `A.this` (of outer type `A`). In general, a method in a class that is nested `n` deep will have `n + 1` implicit parameters: `this` and one for each of its `n` outer classes. We see that `m` can be displayed as a member of class `A` using our standard pretty-print techniques by making the `B` parameter explicit. See Figure 9b.

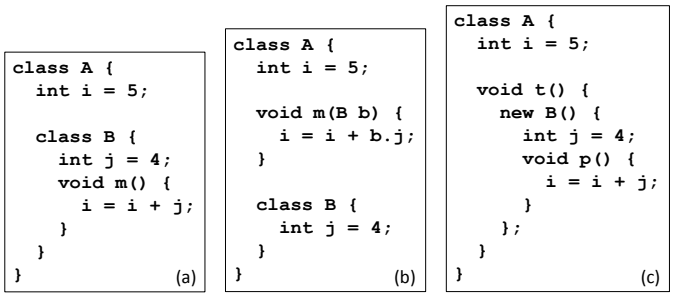

Fig. 9: Nested Classes.

A 'coordinate transformation' interpretation is consistent with why refactoring engines typically cannot move methods of anonymous classes. Consider Figure 9c. The absolute function of method `p` has signature `p(A a, ? b)`, where ? denotes an anonymous subclass of `B`. Since ? has no name to display, refactoring engines refuse to move `p`. (Of course, looking at the problem this way, creating a convert-anonymous-to-nested-class refactoring can remove this restriction).

D. Scripting Refactorings

$\mathcal{R}^3$'s interface is compatible with $\mathcal{R}^2$. That is, $\mathcal{R}^2$ scripts port to $\mathcal{R}^3$. This gives us the ability to script refactorings

to retrofit design patterns into Java programs and we can build compound refactorings as compositions of primitive refactorings. We already saw scripts for `makeAdapter` (Figure 2), `makeVisitor` (Figure 3), and `moveViaField` in Section III-C4.

E. Preconditions

It is well-known that precondition checks are *the* major performance drain in refactoring engines. JDTRE may be typical of other refactoring engines: it checks preconditions as needed.

We take a different approach in $\mathcal{R}^3$. We know the refactorings that $\mathcal{R}^3$ is to support and thus we know all the precondition checks that are required. Some refactorings share checks; other checks are specific to a refactoring. Here is the key: *many (not all) checks can be determined at database build time: they are simple properties of an AST*. For them, we add a boolean attribute to $\mathcal{R}^3$ tables to indicate whether a tuple's AST satisfies that check. The checks for a refactoring then become a conjunction of these boolean attributes. The $\mathcal{R}^3$ database is created by AST traversal and semantic analysis: the first traversal populates the $\mathcal{R}^3$ database with tuples; the second assigns boolean values to these checks. We will see later how this approach improves performance.

The JDTRE *move-instance-method* refactoring is typical. `19` distinct checks for this refactoring are shared by JDTRE and $\mathcal{R}^3$; if any one is satisfied, the move is disallowed.

1) Boolean Checks Made by a Single Tuple Lookup: In $\mathcal{R}^3$, `15` preconditions are AST-harvestable at database build time as boolean values. Here is a sample:

- Abstract – is the method `abstract`?
- Native – is the method `native`?
- Constructor – is the method a `constructor`?
- Interface Declaring Type – is the enclosing type of the method an `interface`?
- Non-Local Type Reference – if the method references a non-local type parameter (e.g., a type parameter of a generic class), it cannot be moved. Figure 10a illustrates a non-local type parameter which prevents method `m` to be moved. In contrast, method `m` in Figure 10b can be moved as its parameter is local.

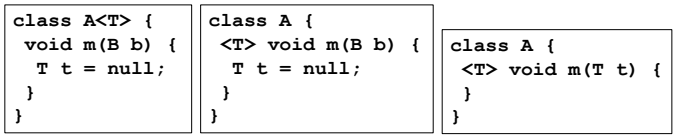

Fig. 10: Generic Constraints.

- Generic Entity Target – moving a method via a type parameter is disallowed (Figure 10c).
- Unqualified Target – a natural home of a method cannot be an `interface`. A natural home is disqualified if its argument is assigned a value as in Figure 11a.
- Null Home Value – if a method call has a `null` home parameter as in Figure 11b, a move to that home is disallowed as it would dereference `null`. More on this in Section VI.

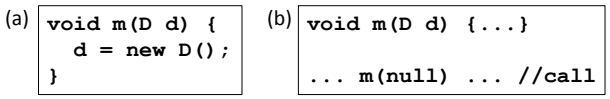

Fig. 11: Target Constraints.

- Polymorphic Method - when the target method is polymorphic, it cannot be moved unless a delegate is left behind. Our `makeVisitor` script satisfies this constraint.
- Synchronized – is the method `synchronized`?

  Note. We know how to move `synchronized` methods correctly; this will in the next $\mathcal{R}^3$ release.
- Recursion – if method `m` invokes itself, the move is disallowed by JDTRE.

  Note. We are unaware of a good reason to prevent such moves; we plan to eliminate this constraint in the next $\mathcal{R}^3$ release.
- Super Reference – JDTRE refuses to move any method that uses the `super` keyword. To write general purpose refactoring scripts, we removed this precondition in $\mathcal{R}^2$ and $\mathcal{R}^3$ by replacing each `super.x()` call with a call to a manufactured method `super_x()` called *super delegate* [9]. Other IDEs, such as IntelliJ IDEA [20] and NetBeans [21], do move such methods, but do so erroneously (Figure 12).

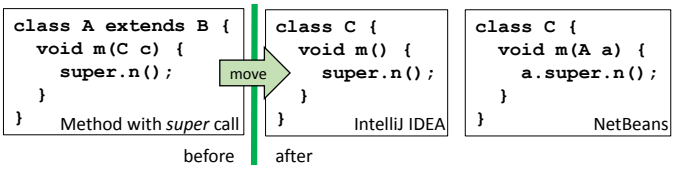

Fig. 12: Super Call Bugs.

The remaining boolean checks are more of the same [22].

2) Checks that Require Database Search: There are several precondition checks in the JDTRE *move-instance-method* that require a database search. Here is a sample:

- Accessibility – after a method is moved, it must still be visible to all of its references. Symmetrically, every declaration that is referenced inside the method's body should be accessible after the move. JDTRE promotes access modifiers of the moved method and/or referenced declarations to satisfy all visibility requirements but this can result in changes to program semantics [23], [24]. Associated with each `RMethod` object `m` is a list of its references (this list is collected at database creation time). $\mathcal{R}^3$ traverses this list to ensure that `m` is still visible to each reference. Similarly, $\mathcal{R}^3$ maintains a second list of tuples (again collected at database creation time) that are referenced in `m`'s body. $\mathcal{R}^3$ traverses this list to ensure that all referenced declarations remain visible to `m`. $\mathcal{R}^3$ now makes the same adjustments in modifiers as JDTRE.

  Note. Prior work [17], [23] identified important issues on setting access modifiers in Java which

JDTRE handles incorrectly. $\mathcal{R}^3$ supports these improved algorithms in its next release.

- Conflicting Method – a method can be moved only when it does not change bindings of existing method references. Consider the 3-class program of Figure 13. A method call `m(null)` inside `B.n()` invokes `C.m(A)`. When JDTRE moves method `A.m(B)` to class `B`, the method call changes its binding to the moved method `B.m(A)`.

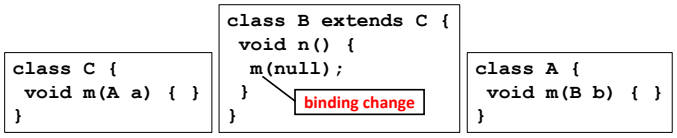

Fig. 13: Method Binding Change.

Clearly this is wrong. JDTRE determines if a conflict exists in the destination class *but not its superclasses*, an error that we have reported [25]. $\mathcal{R}^3$ does better by traversing the class hierarchy and evaluating access modifiers to find conflicts.

Note. Schäfer et al's work on visibility refactoring [17] can be used to find conflicting methods in Java. We have adapted it for the next release of $\mathcal{R}^3$.

- Duplicate Type Parameter – JDTRE moves method `m` in Figure 14 to class `B` only when type parameter `T` is removed from `m` since `T` already exists in class `B`. After the move, however, `T` inside method `m` changes binding to the existing `T` in class `B`.
$\mathcal{R}^3$ harvests type parameter names and stores them in the database tuple where they are declared. $\mathcal{R}^3$ searches the type parameter collections to find a match.

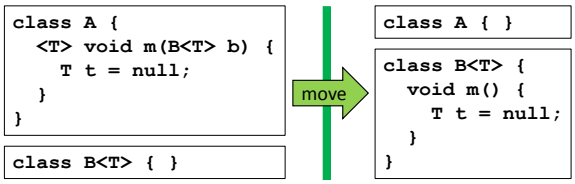

Fig. 14: Duplicate Type Parameter.

- Conflicting Target Name – the *move-instance-method* refactoring introduces an extra parameter to reference members of the original class. JDTRE goes further by allowing a user-supplied name for the parameter to be created, and this name may conflict with existing names. In $\mathcal{R}^3$, existing names are found via a database search of name collection in each tuple.

Note. By adding underscore+numbers to names, $\mathcal{R}^3$ guarantees that user-supplied names are (made) unique, and thus will not conflict with any visible names in the destination. This is called $\mathcal{R}^3$'s *name resolution rule*. It eliminates what would otherwise be a complicating problem; users can perform an explicit rename refactoring if this is inadequate.

IV. Current $\mathcal{R}^3$ Implementation

A. $\mathcal{R}^3$ Pipeline

JDTRE does not use a standard pretty-print AST method. To minimize $\mathcal{R}^3$ coding, we used a pipeline of tools, relying on Eclipse minimally and using AHEAD [26], which has pretty-print methods ideal for $\mathcal{R}^3$. Figure 15 shows the $\mathcal{R}^3$ pipeline: it is a series of stages (A)-(G) that map a target Java program (JDT project) on the left to a refactored program on the right.

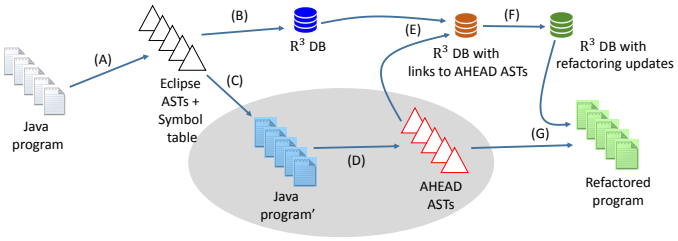

Fig. 15: R3 Pipeline.

(A) Eclipse parses a Java program into ASTs. Below is a target program with a generic method that prints the array argument of different types:

```
package p;
class C {
  // generic method
  static <E> void print(E[] array) {
    for(E e : array)
      System.out.printf("\%s ", e);
  }
}
```

(B) JDT ASTs are traversed to harvest a major part of the $\mathcal{R}^3$ database. Later, step (E) completes the database.

(C) A limitation of AHEAD is that it requires a context-free parser. To satisfy this constraint, a version of the original program is output (shown below) where all identifiers are replaced with manufactured and unique identifiers ID_#; symbols "<" and ">" that indicate generics are replaced with unambiguous symbols "<:" and ":>"; and all white space and comments are preserved. AHEAD can parse this revised file, and with the database of (B) can reconstruct the text of the original program:

```
package ID_0;
class ID_1 {
  // generic method
  static <:ID_2:> void ID_3(ID_4[] ID_5) {
    for(ID_6 ID_7 : ID_8)
      ID_9.ID_10.ID_11("\%s ", ID_12);
  }
}
```

(D) AHEAD parses the manufactured-identifier program.

(E) $\mathcal{R}^3$ database tuples are doubly-linked to their AHEAD AST nodes so each pretty-printer of an AST node can reference the corresponding $\mathcal{R}^3$ tuple and vice versa.

(F) **Refactor**. $\mathcal{R}^3$ refactorings are executed. They modify only the $\mathcal{R}^3$ database, not AHEAD parse trees.

(G) **Pretty-Print.** The source of the refactored program is pretty-printed as described earlier.

Application (Ver#, LOC, #Tests)	Seed ID	# of Refacs	$\mathcal{R}^2$ time (seconds)			$\mathcal{R}^3$ time (seconds)						Speed Up
			Precon Check	Perform Change	Total	Build DB	Link AST	Precon Check	DB Update	Proj	Total	
AHEAD jak2java [26] (130320, 26K, 75)	**A**1	104	16.58	2.31	18.89	0.39	0.16	0.000	0.028	0.21	0.24	79
	A2	68	18.49	2.67	21.16			0.010	0.010	0.11	0.13	163
	A3	554	260.85	37.48	298.33			0.017	0.230	1.87	2.12	141
	A4	60	14.69	3.70	18.39			0.001	0.032	0.54	0.57	32
	A5	96	35.46	7.19	42.64			0.003	0.047	0.96	1.01	42
Commons **C**odec [27] (1.8, 16K, 6103)	**C**1	6	1.80	1.39	3.19	0.29	0.14	0.000	0.007	0.41	0.42	8
	C2	16	4.26	0.70	4.96			0.000	0.007	0.30	0.31	16
	C3	16	3.60	0.30	3.90			0.000	0.007	0.24	0.24	16
	C4	12	3.91	0.68	4.59			0.000	0.007	0.21	0.22	21
	C5	6	1.51	0.50	2.00			0.000	0.005	0.37	0.37	5
Commons **I**O [28] (2.4, 24K, 810)	**I**1	4	1.20	0.19	1.40	0.44	0.17	0.000	0.000	0.05	0.05	28
	I2	4	2.21	0.20	2.40			0.000	0.002	0.08	0.08	31
	I3	6	1.80	0.50	2.31			0.000	0.004	0.35	0.35	7
	I4	4	2.70	0.30	3.00			0.000	0.002	0.07	0.07	42
	I5	6	1.68	0.20	1.88			0.000	0.004	0.32	0.32	6
JUnit [29] (4.11, 23K, 2807)	**J**1	16	4.49	0.70	5.20	0.39	0.15	0.000	0.011	0.17	0.18	29
	J2	4	0.31	0.09	0.40			0.000	0.004	0.05	0.05	8
	J3	18	30.22	3.37	33.60			0.000	0.008	0.32	0.33	103
	J4	20	8.10	1.40	9.49			0.000	0.011	0.44	0.45	21
	J5	4	1.41	0.20	1.61			0.000	0.003	0.09	0.10	17
Quark [26] (1.0, 575, 9)	**Q**	16	3.40	0.40	3.80	0.10	0.01	0.000	0.009	0.09	0.10	40
Refactoring Cra**w**ler [30] (1.0.0, 7K, 15)	**W**1	28	6.99	0.90	7.90	0.24	0.08	0.000	0.016	0.33	0.35	23
	W2	4	1.80	0.30	2.10			0.000	0.004	0.12	0.12	17
	W3	26	11.82	1.01	12.82			0.000	0.013	0.32	0.34	38
	W4	10	4.11	1.10	5.21			0.000	0.007	0.19	0.20	26
	W5	28	9.69	1.40	11.08			0.000	0.015	0.33	0.34	33

TABLE II. Applications and Comparison with $\mathcal{R}^2$ and $\mathcal{R}^3$; Time is in Seconds.

B. Database Relocation and Rebinding

It is common for there to be many references to each declaration (Figure 16a). An expensive operation is to rebind/update all references to one declaration to those of another (Figure 16b). The move-and-delegate refactoring is an example.

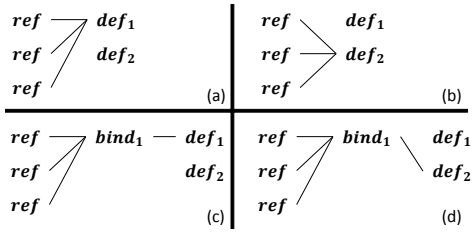

Fig. 16: Rebinding Definitions.

Following the 'one-fact-in-one-place' mantra of database normalization, we introduced an `RBinding` table where declaration bindings are represented once (Figure 16c) and with one update, all references can be rebound (Figure 16d).

V. Evaluation

$\mathcal{R}^2$ calls JDTRE refactorings; $\mathcal{R}^3$ is our replacement for JDTRE. Henceforth we use "Eclipse" to reference $\mathcal{R}^2$+JDTRE. Comparing $\mathcal{R}^3$ with Eclipse (now read: $\mathcal{R}^2$+JDTRE) is comparing $\mathcal{R}^3$ to the state of the practice.

We evaluate $\mathcal{R}^3$ in four ways:

- **Performance**: How fast is $\mathcal{R}^3$ compared to Eclipse?
- **Reliability**: Is $\mathcal{R}^3$ as reliable as Eclipse?
- **Generality**: How general is $\mathcal{R}^3$ compared to Eclipse?
- **Simplicity**: Is $\mathcal{R}^3$ simpler than Eclipse?

We answer each question in the following subsections.

A. Performance

We evaluated $\mathcal{R}^2$ by demonstrating that its scripts could retrofit design patterns into existing programs [9]. Here we compare the performance of $\mathcal{R}^2$ and $\mathcal{R}^3$ on the same programs and scripts. The first column of Table II lists these programs, along with their version, size, and number of regression tests.

The first set of experiments introduces a Visitor pattern into six Java applications. A second set removes a Visitor by executing an inverse Visitor script that exercises a different set of refactorings. Inverse Visitor is not simply performing an undo of existing changes, but is a new script which replaces an instance of a Visitor design pattern in existing code by distributing the visit methods among the constituent visited classes.[3] These experiments engage the core refactorings used in virtually all design patterns. We ran the regression tests on each application after refactoring to confirm there was no difference in their behavior. The experiment environment that we used is an Intel CPU i7-2600 3.40GHz, 16 GB main memory, Windows 7 64-bit OS, and Eclipse JDT 4.4.2 (Luna).

Table II shows the performance results of the first set of experiments. Each program (with the exception of **Q**uark) has five methods that serve as a Visitor seed. To give an idea of the complexity of the refactoring task, the number of JDT refactorings that are executed is given in the `# of Refacs`

[3]Imagine the scenario that a programmer creates a Visitor to view all declarations of a method `m` in class hierarchy. S/he then edits the methods of this Visitor. Simply "undoing" this Visitor rolls back *both* the Visitor *and* her/his changes. An inverse Visitor refactoring is needed to remove the Visitor *and* preserve her/his changes [9].

column.[4] The `Total` column lists the CPU time[5] for the $\mathcal{R}^2$ and $\mathcal{R}^3$ `makevisitor` scripts to execute.

$\mathcal{R}^2$ execution time has two parts, precondition checks and code changes, whose sum equals column `Total`. Column `Precon Check` is the time for all precondition checks discussed in Section III-E *and* a check/parse to see if the compilation units (Java files) involved in the refactoring are broken. ('Broken' means the file has syntax errors). Code change (column `Perform Change`) is the sum of times for calculating the code changes to make, updating the Eclipse workspace, and writing updated files to disk. Precondition checks in JDTRE consume about 87% of refactoring execution time.

$\mathcal{R}^3$ execution time covers six steps (B)-(G) in Figure 17. Steps (C)-(D) are due to our use of AHEAD for coding convenience (and would be zero if JDTRE had pretty-print methods), and thus have nothing to do with $\mathcal{R}^3$ performance.

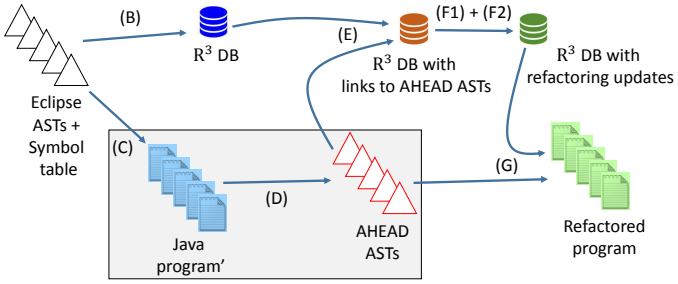

Fig. 17: Performance Pipeline of $\mathcal{R}^3$.

A cost of $\mathcal{R}^3$ is (B) creating the database and (E) linking database tuples to AST nodes, shown as columns in `Build DB` and `Link AST` in Table II. These are tiny execution times. During the brief interval that a programmer selects a refactoring to execute via the Eclipse GUI, a database could be created+linked and the delay would be unnoticeable.

The true execution time for $\mathcal{R}^3$ is (F1) running the script, (F2) checking preconditions, and (G) at the end of the script execution pretty-printing the compilation units that have changed. The sum of these numbers, the $\mathcal{R}^3$ `Total` column, is $\mathcal{R}^3$'s run-time.

To compare the performance of $\mathcal{R}^2$ and $\mathcal{R}^3$, we compute the ratio of their `Total` columns, listed in the `Speed Up` column. $\mathcal{R}^3$ ranges from `5×` to `163×` faster than $\mathcal{R}^2$. The longest $\mathcal{R}^2$ execution time was seed `A3` to create a Visitor of `276` methods, taking `298` seconds of CPU time. In contrast, $\mathcal{R}^3$'s execution time was only `2.2` seconds. Interestingly, even if the number of refactorings executed in a script are small ($4-6$), $\mathcal{R}^3$ was over `18×` faster on average; for larger numbers of refactorings ($>$ `50`), the speed-up was `89×` faster. On average for these experiments, $\mathcal{R}^3$ was `38×` faster than Eclipse. Had we included database build time for steps (B) and (E) in our calculations, the average speed-up ratio drops to `15×`.

Table III shows the corresponding run-times for our second set of experiments that remove a Visitor. Although a different set of refactorings are used, we reach similar conclusions. $\mathcal{R}^3$ ranges from `5×` to `291×` faster than $\mathcal{R}^2$. On average, $\mathcal{R}^3$ was `55×` faster than Eclipse. Had we included database build time for steps (B) and (E) in our calculations, the average speed-up ratio drops to `17×`.

Seed ID	# of Refacs	$\mathcal{R}^2$ time (seconds)			$\mathcal{R}^3$ time (seconds)				Speed Up
		Precon Check	Perform Change	Total	Precon Check	DB Update	Proj	Total	
A1	104	50.80	8.47	59.27	0.003	0.005	0.20	0.21	286
A2	68	27.19	5.10	32.29	0.001	0.006	0.10	0.11	291
A3	554	167.27	46.59	213.86	0.023	0.021	1.75	1.79	119
A4	60	9.98	5.78	15.76	0.008	0.006	0.53	0.55	29
A5	96	19.23	8.97	28.21	0.010	0.008	0.99	1.01	28
C1	6	1.59	0.70	2.29	0.001	0.001	0.43	0.43	5
C2	16	6.61	0.68	7.28	0.000	0.001	0.28	0.28	26
C3	16	7.10	0.40	7.50	0.000	0.001	0.23	0.23	33
C4	12	4.61	0.59	5.20	0.000	0.001	0.20	0.20	26
C5	6	1.70	0.59	2.29	0.000	0.001	0.35	0.35	6
I1	4	2.20	0.21	2.40	0.000	0.000	0.05	0.05	51
I2	4	2.22	0.30	2.52	0.000	0.000	0.07	0.07	35
I3	6	2.21	0.50	2.71	0.000	0.001	0.33	0.33	8
I4	4	1.99	0.20	2.19	0.000	0.000	0.06	0.06	34
I5	6	1.51	0.49	2.00	0.000	0.001	0.30	0.30	7
J1	16	4.75	0.99	5.74	0.000	0.002	0.26	0.27	22
J2	4	1.90	0.20	2.10	0.000	0.000	0.04	0.04	51
J3	18	11.60	0.69	12.28	0.001	0.001	0.31	0.31	39
J4	20	5.81	1.10	6.91	0.001	0.002	0.45	0.46	15
J5	4	2.78	0.21	2.98	0.000	0.000	0.09	0.09	34
Q	16	2.58	0.80	3.38	0.000	0.001	0.08	0.08	41
W1	28	6.28	1.79	8.07	0.002	0.002	0.33	0.33	25
W2	4	5.01	0.40	5.41	0.000	0.001	0.11	0.11	49
W3	26	21.19	1.52	22.71	0.000	0.002	0.31	0.31	74
W4	10	7.92	0.87	8.79	0.000	0.001	0.20	0.20	44
W5	28	15.74	1.68	17.42	0.001	0.002	0.33	0.33	53

TABLE III. Inverse Visitor Results; Time is in Seconds.

There are two reasons for the huge difference in performance. (1) Preconditions are computed with little overhead at $\mathcal{R}^3$ database creation time, in contrast to JDTRE which computes them on demand. Most $\mathcal{R}^3$ preconditions checks are trivial `boolean` field lookups as is evidenced by the almost zero values in the `Precon Check` column. (2) JDTRE writes out changed files after each refactoring and parses all files involved in a refactoring. In contrast, $\mathcal{R}^3$ refactorings are virtually instantaneous database updates. Projection (writing out changed files) is performed only *once* after the script execution is finished. In short, JDTRE was not designed for efficient scripting.

B. Reliability

$\mathcal{R}^3$ uses the same (or improved) precondition definitions as JDTRE; these definitions are well-documented in the JDTRE code base. We extracted from the JDTRE regression suite (`org.eclipse.jdt.ui.tests.refactoring`) tests that are relevant to $\mathcal{R}^3$ refactorings. We excluded tests on Java 8 features (e.g., lambda expressions), as $\mathcal{R}^3$ presently works on JRE 7. There were `122` tests for change-method-signature, `72` for move-method, `73` for pull-up, `59` for push-down, and `138` for rename. $\mathcal{R}^3$ satisfies all `464` extracted tests; they are now part of the $\mathcal{R}^3$ regression suite.[6] Further, in building $\mathcal{R}^2$ and $\mathcal{R}^3$, we discovered and reported `6` errors in the JDTRE, which have now been corrected [9]. We have found others that are not yet fixed and are documented in [12]. In short, $\mathcal{R}^3$ is at least as reliable as JDTRE.

[4]Our `makeVisitor` and `undoVisitor` scripts create and delete program elements but these operations are not counted as JDT refactorings.

[5]We used profiling tool `VisualVM` (ver. `1.3.8`) [31] to measure CPU times in running $\mathcal{R}^2$ and $\mathcal{R}^3$.

[6]$\mathcal{R}^3$ does not produce exactly the same refactored source as JDTRE. For example, $\mathcal{R}^3$ keeps track of moved methods. All type declarations in these methods are displayed with fully qualified names so that additional `import` declarations do not need to be added.

C. Generality

$\mathcal{R}^2$ leveraged existing JDTRE refactorings to script 18 of 23 design patterns in the Gang-of-Four text [6]. We reimplemented all 18 of these patterns in $\mathcal{R}^3$. There are refactorings (inline, extract constant, encapsulate field) that JDTRE supports that $\mathcal{R}^3$ presently does not. These refactorings were not needed to implement design patterns. This paper focuses on $\mathcal{R}^3$ as a replacement of $\mathcal{R}^2$. Future work will extend $\mathcal{R}^3$ to a full complement of standard IDE refactorings.

D. Simplicity

Comparing the size of $\mathcal{R}^3$ to JDTRE in *Lines of Code (LOC)* is difficult, as $\mathcal{R}^3$ implements a subset of JDTRE refactorings. To level the playing field, we used the EcLEmma code coverage tool [32] to see what volume of code was executed by $\mathcal{R}^2$ and $\mathcal{R}^3$ when the `makeVisitor` script was run – this gives us an estimate of the number of *Unique LOC (ULOC)* executed for equivalent functionalities.

$\mathcal{R}^3$ executes 1,782 ULOC for `makeVisitor`. But these ULOC are self-contained, meaning that print, file open and close methods are its only external calls. In contrast, $\mathcal{R}^2$ executes 1,050 ULOC, which in turn calls 1,691 ULOC in `ltk.core.refactoring` (the primary package for JDTRE) and 975 ULOC in `ltk.ui.refactoring` where other core refactoring functionality resides.[7] We conservatively estimate $\mathcal{R}^3$ to be 2× simpler than $\mathcal{R}^2$. Simplicity matters because $\mathcal{R}^3$ does not rely on layers of Eclipse infrastructure, and thus can be studied, debugged, and maintained in isolation.

VI. Related Work

In developing $\mathcal{R}^2$, we found 13 prior works [3], [5], [33–43] that could be used to implement refactoring scripts [9]. They could be classified as program transformation systems, domain specific languages, and refactoring engines built atop of IDEs. Notably none reported performance of refactoring engines; all were demonstrations that their particular infrastructure or tool could be used to implement refactoring or transformation scripts. Most research on refactoring engines mentions the importance of refactoring reliability or error detection [11], [44–47]. We refer readers to [9] for further details.

A critical property of $\mathcal{R}^2$ and $\mathcal{R}^3$ is that refactorings and refactoring scripts are written in the same language as the programs to be transformed (i.e., Java). We feel this property is important because programmers do not have to learn yet another language or programming paradigm to write refactoring scripts. Oddly, only two tools in these 13 papers had this property: Wrangler [47] and IP [15]. Wrangler refactorings and refactoring scripts transformed Erlang programs; IP programs called enzymes transform IP programs.

IP [15] was an inspiration for our work. IP separated ASTs from their user displays. $\mathcal{R}^3$ differs from IP in that $\mathcal{R}^3$ weaves the display of different ASTs together; IP displayed whole trees. We do not know if IP ever had a refactoring engine; the philosophy and infrastructure of IP would suggest that refactorings would have been implemented as AST rewrites. Standard precondition checks in today's refactoring engines to verify that name collisions do not arise (in the rename and move refactorings) were never part of IP; every IP entity has a unique internal identifier. This allowed any number of program elements to have the same display name (e.g., multiple variables with the name `foo` in the same function) and IP could easily distinguish them.

Finally, JDTRE does a shallow check for dereferenced `null`. It searches a project for method calls where a particular method argument is `null`, as opposed to analyses [48–53] which determine whether expressions of particular arguments could evaluate to `null`. We do not know if `null`-analyses have been used in refactoring engines or if they are fast enough to be used.

[7] Example: see checkInitialConditions, checkFinalConditions, and createChange methods in `MoveInstanceMethodProcessor.java`.

VII. Conclusions and Future Work

OO refactoring technology is now 25 years old [54], [55]. Most researchers, ourselves among them, tacitly assume that few significant advances in tooling *classical* Java refactorings are possible after this time. But looking closer, motivated by new needs and applications for refactoring, reveals that significant practical advances are indeed possible.

We have shown how many *classical* Java refactorings (move, rename, change-method-signature, etc.), refactorings that are essential to script the creation and removal of Gang-of-Four design patterns, can be implemented by a novel combination of database+AST pretty-printing. $\mathcal{R}^3$, our implementation, 1) does not rely on a huge codebase required by general-purpose program transformation systems; 2) has a much smaller code footprint than JDTRE; 3) supports the writing and execution of refactoring scripts; 4) executes refactoring scripts about 10× faster than JDTRE; and 5) is reliable as (possibly more so than) JDTRE itself.

Having said the above, $\mathcal{R}^3$ in no way dispenses with the need of program transformation systems. There are *many* refactorings that are not used in scripting design patterns (e.g., see [56], [57]). There are *many* refactorings that cannot simply be "pretty-printed", such as refactoring sequential legacy code into parallel code [58]. Our response is: let's do the basics better and to provide scripting for typical programmers. Experts can learn to use transformation systems for their work.

The next steps in our research are to: 1) show that a full compliment primitive refactorings comparable to that of JDTRE can be added to $\mathcal{R}^3$ and 2) adapt improved precondition checks identified by others [17], [23].

We believe that $\mathcal{R}^3$ provides a useful advance in creating next-generation OO refactoring engines.

Acknowledgments. We gratefully acknowledge support for this work by NSF grants CCF-1212683 and CCF-1439957.

References

[1] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[2] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

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

[4] F. Steimann and J. von Pilgrim, "Constraint-Based Refactoring with Foresight," in *ECOOP*, 2012.

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

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

[7] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," in *ASE*, 1999.

[8] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2006.

[9] J. Kim, D. Batory, and D. Dig, "Scripting refactorings in java to introduce design patterns," Dept Comp. Sci., UTexas-Austin, Tech. Rep. TR-14-14, 2014.

[10] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic testing of refactoring engines on real software projects," in *ECOOP*, 2013.

[11] G. Soares, R. Gheyi, and T. Massoni, "Automated Behavioral Testing of Refactoring Engines," *IEEE TSE*, Feb. 2013.

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

[13] "Eclipse Bug 217753," bugs.eclipse.org/bugs/show_bug.cgi?id=217753.

[14] D. Batory, E. Latimer, and M. Azanza, "Teaching model driven engineering from a relational database perspective," in *MODELS*, 2013.

[15] S. C. Charles Simonyi, Magnus Christerson, "Intentional software," in *ONWARD! OOPSLA*, 2006.

[16] Microsoft, "Intentional programming," Video, 2006.

[17] M. Schäfer, A. Thies, F. Steimann, and F. Tip, "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs," *IEEE TSE*, Nov. 2012.

[18] M. Vakilian and et al., "Use, disuse, and misuse of automated refactorings," in *ICSE*, 2012.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *ICSE*, 2009.

[20] "IntelliJ IDEA 14.1.2," jetbrains.com/idea/.

[21] "NetBeans 8.0.2," netbeans.org/.

[22] J. Kim, "Refactoring by Pretty-Printing," Ph.D. dissertation, University of Texas at Austin, forthcoming.

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

[24] "Eclipse Bug 439090," bugs.eclipse.org/bugs/show_bug.cgi?id=439090.

[25] "Eclipse Bug 467019," bugs.eclipse.org/bugs/show_bug.cgi?id=467019.

[26] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE TSE*, Jun. 2004.

[27] "Apache Commons Codec," commons.apache.org/proper/commons-codec/.

[28] "Apache Commons IO," commons.apache.org/proper/commons-io/.

[29] "JUnit," junit.org/.

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

[31] "VisualVM 1.3.8," visualvm.java.net/.

[32] M. R. Hoffmann, "EclEmma 2.3.2," http://www.eclemma.org, 2014.

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

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

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

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

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

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

[39] E. Balland and et al., "Tom: piggybacking rewriting on java," in *RTA*, 2007.

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

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

[42] 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.

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

[44] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic Testing of Refactoring Engines on Real Software Projects," in *ECOOP*, 2013.

[45] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *ESEC-FSE*, 2007.

[46] W. Jin, A. Orso, and T. Xie, "Automated Behavioral Regression Testing," in *ICST*, 2010.

[47] H. Li and S. Thompson, *Implementation and Application of Functional Languages*. Springer-Verlag, 2008.

[48] N. Ayewah and W. Pugh, "Null Dereference Analysis in Practice," in *PASTE*, 2010.

[49] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *PLDI*, 2002.

[50] M. G. Nanda and S. Sinha, "Accurate Interprocedural Null-Dereference Analysis for Java," in *ICSE*, 2009.

[51] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault Localization and Repair for Java Runtime Exceptions," in *ISSTA*, 2009.

[52] R. Madhavan and R. Komondoor, "Null Dereference Verification via Over-approximated Weakest Pre-conditions Analysis," in *OOPSLA*, 2011.

[53] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs," in *PASTE*, 2005.

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

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

[56] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When Does a Refactoring Induce Bugs? An Empirical Study," *SCAM*, 2012.

[57] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[58] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, Jan 2011.