ABSTRACT

Refactoring engines are standard tools in today’s Integrated Development Environments (IDEs). They allow programmers to perform one refactoring at a time, but programmers need more. Most design patterns in the Gang-of-Four text can be written as a refactoring script – a programmatic sequence of refactorings. In this paper, we present R3, a new Java refactoring engine that supports refactoring scripts. It builds a main-memory, non-persistent database to encode Java entity declarations (e.g., packages, classes, methods), their containment relationships, and language features such as inheritance and modifiers. Unlike classical refactoring engines that modify Abstract Syntax Trees (ASTs), R3 refactorings modify only the database; refactored code is produced only when pretty-printing ASTs that reference database changes. R3 performs comparable precondition checks to those of the Eclipse Java Development Tools (JDT) but R3’s codebase is about half the size of the JDT refactoring engine and runs an order of magnitude faster. Further, a user study shows that R3 improved the success rate of refactoring design patterns by 25% up to 50%.

1. INTRODUCTION

Refactoring is a core technology in software development. All major IDEs today offer some form of refactoring support; refactoring is central to popular software design movements, such as Agile [37] and Extreme Programming [9]. In the last decade, refactoring tools have revolutionized how software is developed. They enable 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 refactorings in their top menu and often compete on the basis of refactoring support.

Despite vast interest and progress, a key functionality that many have recognized to be missing in IDEs is scripting [10]. [27] [47]. Most design patterns in the Gang-of-Four text can be expressed as a refactoring script – a programmatic sequence of refactorings [22] [48]. Adding and removing design patterns manually is laborious, repetitious, error prone, and often too difficult to do – try creating a Visitor with over 10 methods; the benefits of scripting become clear.

We recently [34] added scripting to Eclipse JDT, 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 R2, is detailed in the next section. Experiments revealed JDT Refactoring Engine (JDTRE) is ill-suited for scripting for three reasons:

• Reliability. JDTRE is buggy [20] [25] [45]. We filed 29 new bugs to date, but only a fraction has been fixed in the latest version of Eclipse. Prior to the current release, one R2 script executed 6 JDT refactorings producing a program with 27 compilation errors. Another script invoked 96 refactorings, producing a program with 100 compilation errors. These errors are not due to R2, but are egregious bugs in JDTRE. We are constantly discovering more. Worse is waiting months or years for a repair [20]. We rediscovered a bug that took 5 years to be fixed [18].

Note: We are not in a 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. JDTRE refuses to move methods that include the super keyword; moving methods with super reference(s) is really useful. We also had to turn off parameter optimization, for example, to make JDT refactorings produce design patterns correctly [34].

• Speed. JDTRE’s Achilles heel is its speed: it is surprisingly slow. While a single JDT refactoring is fast, executing many is not. R2 scripts that invoke 20 refactorings take over 10 seconds. One script invoked 554 refactorings and took 5 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 R3, creates a database of program elements (such as classes, methods, fields), their containment relationships, and Java language features such as inheritance and modifiers. Precondition checks consult harvested values in database tuples; refactorings alter the database. ASTs are never changed; refactored code is produced only when pretty-printing ASTs that reference database changes. This strategy yields a 10× increase in refactoring speed and a 50% smaller codebase.
2. A RECAP OF R2

Most classical design patterns can be expressed by a series of refactorings \cite{32, 48}. In \cite{34}, we leveraged the JDTRE to provide a practical way to write such scripts. 

R2 is a Java package. Its objects correspond to Java entity declarations such as packages, classes, methods, etc. The program in Figure 1 has seven R2 objects. There are 3 classes: Graphic, Square, Picture and 4 methods: three draws and one add.

Methods of R2 objects are JDT refactorings or database retrievals. Representative methods are listed in Table 1.

<table>
<thead>
<tr>
<th>R2 Type</th>
<th>Method Name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPackage</td>
<td>newClass</td>
<td>add a new class to the package</td>
</tr>
<tr>
<td>RClass</td>
<td>addSingleton</td>
<td>apply Singleton pattern to the class</td>
</tr>
<tr>
<td>RClass</td>
<td>getAllMethods</td>
<td>return a list of R2 objects that are all methods of the class</td>
</tr>
<tr>
<td>RClass</td>
<td>getPackage</td>
<td>return the R2 object of its own package</td>
</tr>
<tr>
<td>RClass</td>
<td>newConstructor</td>
<td>as a new constructor to the class</td>
</tr>
<tr>
<td>RMethod</td>
<td>newMethod</td>
<td>add a new method stub to the class</td>
</tr>
<tr>
<td>RRelativeList</td>
<td>newField</td>
<td>add a new field to the class</td>
</tr>
<tr>
<td>RRelativeList</td>
<td>getRelatives</td>
<td>set to implement an interface</td>
</tr>
<tr>
<td>RMethod</td>
<td>addParameter</td>
<td>return a list of R2 objects of methods with the same signature</td>
</tr>
<tr>
<td>RRelativeList</td>
<td>moveAndDelegate</td>
<td>add a parameter with its default value to all methods</td>
</tr>
<tr>
<td>RRelativeList</td>
<td>rename</td>
<td>move the method to a class and leave behind its delegate</td>
</tr>
</tbody>
</table>

Table 1: Methods of R2.

R2 refactoring scripts are short Java scripts. Here are two examples. Figure 2 is an R2 script that creates an Adapter. A programmer uses the Eclipse GUI to identify a Java class c that is to be adapted to Java interface i. The programmer then invokes R2’s makeAdapter refactoring (just like a built-in Eclipse refactoring), which in turn invokes i.makeAdapter(c,N) where N is the name of the Adapter class to be created. Class N is created in the same package as interface i (Line 5), to which is added a field named adaptee of type c and a constructor to initialize adaptee (LINES 5-6). A stub is generated for each method in interface i (Line 9). The created class N implements interface i (Line 11). The R2 object for N is returned as the result of makeAdapter.

Figure 2: R2 makeAdapter Method.

Figure 3 is an R2 script to create a Visitor design pattern. Using the Eclipse GUI, a programmer identifies a method, called a seed, in a class hierarchy that s/he wants to create a Visitor for; s/he then invokes R2’s makeVisitor refactoring from the Eclipse GUI. Doing so invokes seed.makeVisitor(N), where seed is R2 object of the seed and N is the name of the Visitor class to be created. makeVisitor gets the seed’s package, creates a Visitor class v with name N in that package, and makes v a Singleton (LINES 11-12). Next, all methods with the same signature as the seed are collected onto a list. Every method on the list is renamed to accept (Line 5), and then a parameter of type v is added whose default value is the Singleton field of N (Line 10). The index value that is returned is the index number of the Visitor object. Only movable methods (e.g., abstract or interface methods cannot be moved) are relocated to class N, leaving behind delegates, respectively (Line 11). All methods in the Visitor class are renamed to visit, makeVisitor returns v, the R2 Visitor class object.

Figure 3: R2 makeVisitor Method.

These examples of 15 LOC are typical of R2 scripts; they are very short. We implemented 18 of the 23 design patterns in the Gang-of-Four text \cite{24} using R2. Eight patterns (including Visitor) are fully automatable as there are no programmer TO-DOs. Another ten are partially automatable. This includes Adapter, where only stubs are generated. Some of the remaining patterns are automatable, such as State and Mediator\footnote{State is a typical MDE application. Mediator is the essence of} while others, Fa¸ cade and Iterator, are
so application-specific that little or nothing is reusable [24].
Java is a practical scripting language – writing R2 scripts is like writing regular Java code. Programmers do not need to learn a Domain Specific Language (DSL) for program transformations. R2 scripts invoke JDT refactorings and create new program elements. Although using Java as a scripting language is great, JDTRE is not, as mentioned in the Introduction (reliability and speed being the biggest detractors). JDTRE was never designed for scripting refactorings. Hence the motivation for R3.

3. R3 CONCEPTS

3.1 Modularity Perspectives

Elementary physics inspired R3. A physical object looks different depending upon an observer’s location. Silhouette portraits of people are different from frontal portraits. Just as viewpoints of a physical object are created by rotations and translations, called coordinate transformations that preserve object properties, R3 does the same for programs: it refactor programs by pretty-printing without changing the program’s ASTs or behavior.

To see how, 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 4b is the signature of an absolute function foo with three parameters whose types are B, C, D.

Figure 4: An Absolute Function and its Relative Methods.

If foo is displayed as a member of class B, Figure 4a, is its signature: the B parameter becomes this and is otherwise implicit. If foo is displayed as a member of class C, Figure 4c, is its signature, where the C parameter is this. We say the natural homes of an absolute function are its parameter types. The natural homes for method foo are B, C, D. If foo is displayed as a member of class E, not a natural home, it appears as the static method of Figure 4d which has no implicit this parameter.

A modularity perspective assigns absolute functions to class declarations. The idea generalizes to other entity declarations (e.g., packages, classes, fields) and their containment relationships. To illustrate, nested classes generalize absolute functions in an interesting way. Figure 5a shows class B nested inside class A. Method m of class B has the absolute function:

```java
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). We see that a() can be displayed as a member of class A using our modularity perspective techniques by making the B parameter explicit. See Figure 5b.

A ‘coordinate transformation’ interpretation also explains why refactoring engines do not move methods of anonymous classes. Consider Figure 5c. 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.

In R3, by creating a database of program elements and their containment relationships, classical refactorings become simple database modifications and never alter the ASTs of the target program. The AST is ‘absolute’ or immutable; it appears different relative to the modularity perspective from which it is displayed. The move-instance-method refactoring, which is what Figure 4b is about, is a coordinate transformation for software; it preserves the semantic properties of a program. The same holds for other primitive refactorings.

3.2 The R3 Database

R3 maintains an internal, non-persistent database to record changes in perspective. When R3 parses compilation units of a program, it creates relational database tables for all declaration types in a program. Each tuple of the RClass table represents a unique 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 unique method (or 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 Gang-of-Four design patterns and almost all classical refactorings.

Program source is compiled into ASTs which are traversed to populate R3 tables. Figure 5 shows the basic set-up. Three RClass tuples (Graphic, Square, Picture) are created. So too are four RMethod tuples (Graphic.draw, Square.draw, Picture.add, Picture.draw) that are linked to the RClass tuple for which each is a member.

Refactorings update this database. Renaming a method updates the name field of that method’s R3 tuple. Moving a method to another class updates the method’s R3 tuple to point to its new class. Only when an AST is rendered (displayed) is the information in the R3 database revealed. When a method’s AST is displayed, the name of the method is extracted from the method’s R3 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 current class. Figure 5 sketches the RClass display method: it prints the class keyword, the current class name, extends clause with its superclass name, and implements clause with interface name(s); 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 ‘}’. R3 reproduces the original order in which members appeared for ease of subsequent reference by programmers and preserves all source code comments.

Rendering 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 3) is moved from class B to C. All invocations of foo, such as b.foo(c, d), are 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.

Typical refactoring engines modify ASTs. In contrast, R3 eliminates AST manipulation. R3 still needs to create ASTs when new program elements are needed, but other than that, R3 does not manipulate ASTs. As we report later, a consequence of the above is that the codebase for R3 is much smaller and simpler than JDTRE.

### 3.3 Primitive Refactorings

We now explain some representative primitive refactorings to see how they are implemented in R3. In the refactoring community, behavior preservation is determined by statically analyzing whether the input code passes the refactoring’s preconditions. If all preconditions are met, the refactoring engine is allowed to change the program code. We partition our discussion on refactorings into two segments: database changes corresponding to code transformations in conventional refactorings (considered in this section) and precondition checks (discussed in the next section).

#### 3.3.1 Rename Method

Rename-instance-method modifies the name field of the Method tuple. This refactoring, like most, have a database-transaction quality. Consider a class hierarchy where all classes have their own method foo. To rename foo to bar can be expressed as a loop, where getRelatives() finds all overriding/overridden methods with the same signature as foo:

```java
for (RMethod m : foo.getRelatives()) {
    m.rename("bar");
}
```

Until the loop completes, not all methods are renamed and preserving program semantics is not guaranteed. R3 performs renames on sets of overriding/overridden methods with identical signatures, and by being a set operation, does not expose an inconsistent database to users:

```java
RelativeList relatives = foo.getRelatives();
relatives.rename("bar");
```

#### 3.3.2 Change Method Signature

Change-method-signature adds, removes, and reorders method parameters. Encoded in the R3 database is a list of formal parameters for every method. Adding a parameter to a method simply adds the 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 [39, 50] found that highly-parameterized refactorings with options (name, parameter add/delete/reorder, exception, delegate) discourage the use of refactorings and make them harder to understand. Accordingly, R3 has separate methods to add, remove, and reorder parameters. Line 1 below finds the R3 tuple for a field 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 (Line 2). The new parameter, by default, becomes the last formal parameter of method m. Line 3 makes it the first parameter of method m:

```java
1   RField v = RField.find("p", "C", "f");
2   RParameter newParam = m.addParameter(v);
3   newParam.setIndex(0);
```

Like rename, addParameter has a set-based version.

#### 3.3.3 Move Method via Parameter

The move-instance-method refactoring in R3 changes the home class of a method. Recall that a home parameter is any parameter of a, and a home class is the class of a home parameter. Moving a to a home class simply updates a’s R3 tuple to point to the tuple of its home class. Presuming c is a home class, the code below moves method m to class c:

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

#### 3.3.4 Move Method via Field

The move-via-field refactoring is illustrated in Figure 8. Method m in class A, whose absolute signature is C m(A a, B b), is moved to class D via field d. A local invocation, m(b), becomes d.m(this, b). Here is where scripting comes in handy: move-via-field is the following R3 script:

```java
// member of RMethod class
void moveViaField(RField f) {
    RParameter newHome = addParameter(f);
    move(newHome);
}
```
3.3.5 Introducing New Program Elements

R3 introduces complex new code declarations (classes, methods, fields, etc.) into an existing program by creating a compilation unit with these declarations. The file is compiled and the database is updated with new declarations which are then embedded into the existing program via move refactorings. The code below shows how to create a custom method `mul()`, whose R3 object is `mth`:

```java
class A extends B {
  D d;
  C m(B b) {…}
  … m(b) … // a call
}
class D  {
}
class A  { 
  D d;
  … d.m(this, b) … // a call
}
```

Once the needed methods and fields are removed from compilation unit `cu`, the unit can be marked deleted in the database using the R3 remove refactoring. The AST of `cu` remains, but at pretty-printing time no text of its (now empty) compilation unit is produced.

3.3.6 Scripting Refactorings

R3 supports all refactorings that are essential to introduce or remove design patterns from existing programs. R3’s interface is compatible with R2. That is, R2 scripts port to R3. 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 3.3.4.

3.4 Preconditions

Precondition checks are the major performance drain in refactoring engines. JDTRE is typical: it checks preconditions as needed. Every refactoring call `r()` on an R3 object `obj` requires a conjunction of precondition checks `obj.p1()∧obj.p2()∧…∧obj.pn()` where `p1()` is a primitive precondition. For example, the JDT move-instance-method refactoring has 19 distinct checks (which are also present in R3); if any one fails, the move is disallowed. Since JDTRE does not know if a programmer will invoke `obj.r()`, JDTRE does the obvious thing by evaluating `obj.p1()∧obj.p2()∧…∧obj.pn()` only when needed.

R3 is different. We too do not know what refactorings a programmer will invoke. But we can precompute the value of many – not all – `p1()` for all R3 objects at database build time, even though we may never use these values. For each `p1()`, we add a boolean attribute to R3 tables to indicate whether a tuple’s AST satisfies `p1()`. The checks for a refactoring then become a conjunction of these boolean attributes.

The R3 database is created by traversing the ASTs of a program and collecting semantic information. Doing so populates the R3 database with tuples and assigns boolean values to these checks. Further, in cases where harvested boolean values are insufficient, we optimized the R3 database to facilitate fast searches, e.g., R3 collects all references of a declaration to reduce search overhead. We will see in Section 3.4.1 these techniques improve performance significantly.

3.4.1 Boolean Checks Made by a Single Tuple Lookup

In R3, fifteen preconditions (which JDT move-instance-method uses and are shared by other refactorings) are AST-harvestable at database build time as boolean values. Here is a representative 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 4 illustrates a non-local type parameter which prevents a move of method `m`. In contrast, method `m` in Figure 5 can be moved as its parameter is local.

![Figure 8: Move via Field Refactoring.](image)

![Figure 9: Generic Constraints.](image)

![Figure 10: Target Constraints.](image)

![Figure 11: Super Call Bugs.](image)
3.4.2 Checks that Require Database Search

Not all primitive preconditions are reducible to boolean attributes; these outliers require a database search, which R3 performs efficiently. Here are some for the move-instance-method:

- **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. R3 does the same.

  Associated with each MethodInvocation object m is a list of its references (this list is collected at database creation time). R3 traverses this list to ensure that m is still visible to each reference. Similarly, R3 maintains a second list of tuples (again collected at database creation time) that are referenced in m’s body. R3 traverses this list to ensure that all referenced declarations remain visible to m. R3 makes the same adjustments in modifiers as JDTRE.

- **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 12. A method call m(...) inside B.n() invokes A.m(c). When JDTRE moves method C.m(B) to class B, the method call changes its binding to the newly moved method B.m(c).

  ```java
  class A {
      void m(C c) { }
  }
  class B extends A {
      void m() {
          m(...);
      }
  }
  class C {
      void m(B b) { }
  }
  ```

  **Figure 12: 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 [19]. R3 does better by traversing the class hierarchy and evaluating access modifiers to find conflicts [14].

- **Duplicate Type Parameter** – JDTRE moves method m in Figure 13 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.

  ```java
  class A {
      <T> void m(B<T> b) { }
  }
  class B<T> { }
  ```

  **Figure 13: Duplicate Type Parameter.**

R3 harvests type parameter names and stores them in the database tuple where they are declared. R3 searches the type parameter collections to find a match.

### 4. CURRENT R3 IMPLEMENTATION

JDTRE does not use a standard pretty-print AST method. To minimize R3 coding, we used a pipeline of tools, relying on Eclipse minimally and using AHEAD [5], which has pretty-print methods ideal for R3. Figure 11 shows the R3 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.

- **A** Eclipse parses a Java program into ASTs.
- **B** R3 parses a Java program into ASTs.
- **C** AHEAD requires a context-free parser. To satisfy this constraint, a version of the original program is output (shown below) where white space and comments are preserved and all identifiers are replaced with manufactured and unique identifiers ID, #: symbols “<” and “>” that indicate generics are replaced with unambiguous symbols “<” and “>”. AHEAD can parse the revised compilation unit.

  ```java
  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 R3 database. Later, step (E) completes the database.**

- **C** AHEAD requires a context-free parser. To satisfy this constraint, a version of the original program is output (shown below) where white space and comments are preserved and all identifiers are replaced with manufactured and unique identifiers ID, #: symbols “<” and “>” that indicate generics are replaced with unambiguous symbols “<” and “>”. AHEAD can parse the revised compilation unit.

  ```java
  package ID_0;
  class ID_1 {
      // generic method
      static <ID_2> void ID_3(ID_4[] ID_5) {
          for(ID_6 e : ID_8)
              System.out.printf("\%s ", e);
      }
  }
  ```

  **(C) AHEAD requires a context-free parser. To satisfy this constraint, a version of the original program is output (shown below) where white space and comments are preserved and all identifiers are replaced with manufactured and unique identifiers ID, #: symbols “<” and “>” that indicate generics are replaced with unambiguous symbols “<” and “>”. AHEAD can parse the revised compilation unit.**

  ```java
  package ID_0;
  class ID_1 {
      // generic method
      static <ID_2> void ID_3(ID_4[] ID_5) {
          for(ID_6 e : ID_8)
              System.out.printf("\%s ", e);
      }
  }
  ```

  **(D) AHEAD parses the manufactured-identifier program.**

- **E** R3 database tuples are doubly-linked to their AHEAD AST nodes so each pretty-printer of an AST node can reference the corresponding R3 tuple and vice versa.

- **F** R3 refactorings are executed. They modify only the R3 database, not AHEAD parse trees.

- **G** The source code of the refactored program is pretty-printed as described earlier.

### 5. EVALUATION OF R3

To evaluate the usefulness of R3, we answer the following research questions:

- **RQ1 (Performance):** How fast is R3 compared to JDTRE?
- **RQ2 (Correctness):** Does R3 improve the correctness of the result when retrofitting a design pattern?
- **RQ3 (Productivity):** Does R3 reduce the required time to retrofit a design pattern?

Previously, in [34], we evaluated the expressiveness of the R3’s predecessor, R2, by demonstrating that its scripts can retrofit design patterns into real-world programs. We focused on patterns that (a) were the hardest to manually create and (b) executed the most JDT refactorings. We used the same R2 tests for R3, not only to show that R3 is
similarly expressive and can handle the complexities of real-world programs, but also to measure R3’s performance w.r.t. JDTRE – noting that JDTRE is representative of the state of the practice in refactoring engines. In addition, in this paper we also focus on practicality. Namely, can programmers use R3 effectively?

To answer these questions, we use a combination of two empirical methods: a case study using 6 Java real-world programs and user studies (with 44 undergraduates and 10 graduate students) that complement each other. The user study allows us to quantify programmer time and program errors, while the case studies give more confidence that R3 generalizes to real-world situations.

### 5.1 Performance

The first column of Table 2 lists the programs of the R3 evaluation, along with their version, LOC, and number of regression tests. We performed two sets of experiments.

The first set of experiments retrofitted a Visitor pattern into six Java applications. The second set removed a Visitor by executing an Inverse-Visitor script that exercises a different set of refactorings. Inverse-Visitor does not undo existing changes, but is a script that removes an instance of a Visitor design pattern by moving visit methods back to their original classes. These experiments engage the primitive refactoring tasks that are used the most often in design patterns. We ran the regression tests on each application after script execution to confirm there was no difference in program behavior. We used Intel CPU i7-2600 3.40GHz, 16 GB main memory, Windows 7 64-bit OS, and Eclipse JDT 4.4.2 (Luna) in our work.

Table 2 shows the performance results of the first set of experiments. Each program (with the exception of Quark) has five methods that serve as a Visitor seed. The complexity of a refactoring task is measured by (1) the number of JDT refactorings executed; this number is given in the # of RefacS column and (2) the CPU time listed in the JDTRE time (seconds) column.

<table>
<thead>
<tr>
<th>Application</th>
<th>Seed ID</th>
<th># of RefacS</th>
<th>JDTRE time (seconds)</th>
<th>Build DB (B)</th>
<th>Link AST (E)</th>
<th>Precon Check (F1)</th>
<th>Perform Change (G)</th>
<th>Total (H)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHEAD</td>
<td>A1</td>
<td>1294</td>
<td>18.58</td>
<td>0.000</td>
<td>0.002</td>
<td>0.010</td>
<td>0.010</td>
<td>0.024</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>68</td>
<td>18.49</td>
<td>0.000</td>
<td>0.010</td>
<td>0.111</td>
<td>0.113</td>
<td>0.124</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>554</td>
<td>200.85</td>
<td>0.017</td>
<td>0.230</td>
<td>1.87</td>
<td>2.12</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td>60</td>
<td>14.69</td>
<td>0.032</td>
<td>0.370</td>
<td>0.54</td>
<td>0.57</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A5</td>
<td>96</td>
<td>35.46</td>
<td>0.003</td>
<td>0.047</td>
<td>0.96</td>
<td>1.01</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C1</td>
<td>724</td>
<td>42.64</td>
<td>0.000</td>
<td>0.006</td>
<td>0.011</td>
<td>0.016</td>
<td>0.032</td>
<td>91</td>
</tr>
<tr>
<td>Commonwealth</td>
<td>C2</td>
<td>16</td>
<td>4.26</td>
<td>0.000</td>
<td>0.007</td>
<td>0.30</td>
<td>0.31</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Coda</td>
<td>C3</td>
<td>16</td>
<td>4.30</td>
<td>0.003</td>
<td>0.007</td>
<td>0.24</td>
<td>0.24</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4</td>
<td>12</td>
<td>3.91</td>
<td>0.000</td>
<td>0.001</td>
<td>0.21</td>
<td>0.22</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C5</td>
<td>4</td>
<td>1.51</td>
<td>0.000</td>
<td>0.000</td>
<td>0.27</td>
<td>0.27</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>CommonKode</td>
<td>C6</td>
<td>2</td>
<td>0.67</td>
<td>0.000</td>
<td>0.000</td>
<td>0.45</td>
<td>0.45</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Jakarta</td>
<td>J1</td>
<td>16</td>
<td>4.49</td>
<td>0.000</td>
<td>0.011</td>
<td>0.17</td>
<td>0.18</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J2</td>
<td>4</td>
<td>0.31</td>
<td>0.000</td>
<td>0.004</td>
<td>0.05</td>
<td>0.05</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J3</td>
<td>4</td>
<td>3.02</td>
<td>0.000</td>
<td>0.004</td>
<td>0.32</td>
<td>0.33</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J4</td>
<td>20</td>
<td>8.10</td>
<td>0.000</td>
<td>0.011</td>
<td>0.44</td>
<td>0.45</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J5</td>
<td>4</td>
<td>1.41</td>
<td>0.000</td>
<td>0.003</td>
<td>0.09</td>
<td>0.10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td>16</td>
<td>3.40</td>
<td>0.021</td>
<td>0.000</td>
<td>0.09</td>
<td>0.10</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Refactoring</td>
<td>W1</td>
<td>28</td>
<td>0.99</td>
<td>0.006</td>
<td>0.014</td>
<td>0.33</td>
<td>0.35</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W2</td>
<td>4</td>
<td>1.80</td>
<td>0.000</td>
<td>0.004</td>
<td>0.12</td>
<td>0.12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Crawlers</td>
<td>W3</td>
<td>60</td>
<td>11.82</td>
<td>0.000</td>
<td>0.013</td>
<td>0.32</td>
<td>0.34</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W4</td>
<td>10</td>
<td>4.11</td>
<td>0.000</td>
<td>0.007</td>
<td>0.19</td>
<td>0.20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W5</td>
<td>28</td>
<td>9.69</td>
<td>0.000</td>
<td>0.015</td>
<td>0.33</td>
<td>0.34</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Applications and Comparison with JDTRE and R3

JDTRE 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 4.4 and a check/parse to see if the compilation units (Java files) involved in the refactoring are ‘broken’ – meaning that 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. **Note:** precondition checks in JDTRE consume about 87% of refactoring execution time.

R3 execution time covers six steps (B)-(G) in Figure 15. Steps (C)-(D) are due to our use of AHEAD for coding convenience and would be unnecessary if JDTRE had usable pretty-print methods. We exclude times for (C)-(D) as they have nothing do with R3 performance.

![Figure 15: Performance Pipeline of R3.](image)

A cost of R3 is (B) creating the database and (E) linking database tuples to AST nodes, shown as columns in Build DB and Link AST in Table 2. These execution times are minute. During the brief interval that it takes to display the R3 GUI refactoring menu, a database can be created+linked with an unnoticeable delay.

The true execution time for R3 is (F1) checking preconditions, (F2) updating database, and (G) at the end of the script execution pretty-printing the compilation units that have changed. The sum of these numbers, the **Total** column, is R3’s run-time.

We compute the ratio of the JDTRE and R3 total columns, listed in the Speed Up column. R3 ranges from 5× to 163× faster than JDTRE. The longest JDTRE execution time was 54. A profiling tool VisualVM (ver. 1.3.8) to measure CPU times in running the JDTRE and R3 scripts. We repeated each experiment five times and report the average execution time.
seed A3 to create a Visitor of 276 methods, taking 298 seconds of CPU time. In contrast, R3’s execution time was 2.2 seconds. Interestingly, even if the number of refactorings executed in a makeVisitor script are small (4 – 6), R3 was 17× faster on average; for larger numbers of refactorings (> 50), the speed-up was 91× faster. On average for these experiments, R3 was 38× faster than JDTR

Table [1] shows the corresponding run-times for our second set of experiments that removed a Visitor. Although a different set of refactorings are exercised, we reach similar conclusions. R3 ranges from 5× to 291× faster than JDTR. On average, R3 was 55× faster than JDTR.

<table>
<thead>
<tr>
<th>Seed</th>
<th># of Refs</th>
<th>JDTR Time (seconds)</th>
<th>R3 Time (seconds)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>100</td>
<td>8.98</td>
<td>16.23</td>
<td>0.56</td>
</tr>
<tr>
<td>A2</td>
<td>101</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>A3</td>
<td>102</td>
<td>58.88</td>
<td>16.23</td>
<td>3.62</td>
</tr>
<tr>
<td>B1</td>
<td>500</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>C1</td>
<td>103</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>C2</td>
<td>104</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>C3</td>
<td>105</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>D1</td>
<td>106</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
<tr>
<td>D2</td>
<td>107</td>
<td>58.88</td>
<td>2.29</td>
<td>26.3</td>
</tr>
</tbody>
</table>

Table 3: Inverse Visitor Results.

There are three basic reasons for the huge difference in performance. First, as mentioned earlier, JDTR evaluates preconditions by searching ASTs, and piggy-backs the collection of information to know what text changes to make to perform an actual refactoring, such as creating a method delegate, adjusting declaration visibility, etc. Profiling experiments indicate that the vast majority of time (avg: 60%, sd: 15%) of the Precon Check column for JDTR is simply due to AST searching. R3 reduces the overhead by collecting all program elements and values needed for preconditions checks or code transformation in advance.

Second, the R3 database has been optimized to make normally slow operations lightning fast. One such operation is the re-binding of all references to one declaration of another (Figure 10(a)). The move-and-delegate refactoring is an example. Following the ‘one-fact-in-one-place’ mantra of database normalization, R3 introduced an RBinding table where declaration bindings are represented once and with one update, all references are rebound (Figure 10(b)).

Figure 16: Reference Binding in R3.

Third, JDTR parses all files involved in a refactoring and writes out changed files after each refactoring. In contrast, R3 refactorings are virtually instantaneous database updates. Projection (i.e., writing out changed files) is performed only once after the script execution is finished.

In short, JDTR was not designed for efficient scripting.

5.2 Practicality

We conducted an evaluation of R3’s practicality. We designed two controlled experiments (the Adapter experiment and the Visitor experiment) to assess how users worked with R3. We ran the experiments with 44 students in Spring 2015 at the undergraduate [CS373S Software Design] course at the University of Texas at Austin. The course exposes students to fundamental structures and concepts in software development, with an emphasis on automation. Two lectures were devoted to refactoring and seven more were dedicated to design patterns.

We ran another Visitor experiment with 10 students in Fall 2014 at the graduate [CS661 Advanced Software Engineering] course at the Oregon State University. This course exposes students to seminal topics and recent trends in software evolution; in particular automating common changes to improve software quality. Results from both executions were consistent.

5.2.1 Experimental Design

We had two dependent variables: correctness and time. Correctness was first measured as a boolean metric: either the result was correct or not. We also used a score that measured the degree of correctness (0 meant nothing had been done to the existing code, and 100 meant the pattern had been correctly introduced). Time was measured in minutes. The only independent variable was the method used to retrofit the pattern (i.e., R3 scripts vs. using available JDT refactorings or manual edits).

As an approximation, the complexity of a pattern instance is the number of refactorings that must be applied to produce the instance. There is clearly more: programmers must order refactorings in a proper sequence to achieve the desired result. In any case, creating and removing Visitor and Adapter pattern instances require sequences of refactorings of different length using different sets of primitive refactorings. We believe both are representative of refactoring scripts that programmers can (or would like to) apply.

Based on these patterns, we designed two separate experiments: one for Visitor and another for Adapter. To counteract the impact of the order of the method participants used, we counterbalanced it. Each experiment consisted of two tasks. Group A performed the first task using R3 and the second using the available JDT refactorings; Group B did in the opposite order. Further, we balanced Group A and Group B w.r.t. their backgrounds, using information that students provided in a survey at the beginning of the course.

To ensure uniform knowledge among participants, each participant read and practiced online tutorials to:

- make and remove a Visitor and Adapter manually [54],
- write and run R3 scripts, and
- apply JDT refactorings such as rename, move, and change-method-signature, with an explanation of their options.

Students submitted practice assignments (code and scripts); only when they passed the tutorial assignments could they proceed to the real experiment.
In the Visitor experiment, each student received a target program, RefactoringCrawler [17], an open-source Eclipse plugin. RefactoringCrawler has 119 Java classes, 17 interfaces and 7K LOC, including a suite of JUnit tests.

In the first task, Group A wrote a general R3 script to make a Visitor, and applied this script to create a Visitor with 13 methods given seed W1. Group B applied Eclipse refactorings manually to make the same Visitor. The second task, (1) participants removed an existing Visitor with 12 methods from the target program, but from a different class hierarchy and (2) we flipped the control group: Group A applied Eclipse refactorings manually and Group B wrote and applied a general R3 script. In the Adapter experiment, Group A was required to write a general R3 script to make an Adapter that implements 35 methods. Group B created the same Adapter by hand as JTDRE offers no useful refactorings for this task. In the second task, we flipped the control group and targeted a different Adapter of the same size.

We capped each task to 2 hours, although some participants extended this limit. Participants were not allowed to take extended breaks but were free to abort after spending the maximum time. Participants had to verify their work by running the regression tests that came with RefactoringCrawler.

Tasks were homework assignments. Participants had access to classroom material and tool tutorials. To determine participant success or failure, we analyzed their refactored programs and R3 scripts, ran the regression tests, and manually inspected their code. Students also reported the time they spent on each task and completed a follow-up survey.

5.2.2 Results

Tables 4 and 5 summarize the results we obtained from the UT and OSU executions respectively. As Shapiro-Wilk tests showed a significant deviance from normality for score and time, we resorted to non-parametric Wilcoxon signed-rank tests for all the analyses. Both tables present the percentage of successful submissions, means and standard deviations for the score they obtained, and time spent. Tables also show the test result (z), its corresponding p value and the effect size (r) in the cases where statistically significant differences were found between both methods (p < 0.05).

Results are consistent in both executions. For RQ2 (Correctness), we found statistically significant differences that favor R3 in both success and score in both UT and OSU. Moreover, the effect size introduced by R3 was large (r > 0.5) for the Visitor experiment and medium (r > 0.3) for the Adapter experiment, showing that R3 has a significant impact on success and score rates. We hypothesize that even greater benefits for R3 accrue when the complexity of a pattern (i.e., the types and numbers of required refactorings) increases. More on RQ2 in Section 5.3.

For RQ3 (Productivity), results show statistically significant differences that favor using JDT refactorings in the required time to apply the design pattern. Effect sizes are large in all cases. In other words, for this experiment and design pattern instances, it was faster to manually invoke JDT refactorings than to write an R3 script from scratch (however, once a script is written, it can be reused many times). More on RQ3 in Section 5.3.

Clearly students can write R3 scripts. In a follow-up poll, 91% of them said that writing (R3) refactoring scripts would be a useful addition to their IDE and 79.5% said that writing scripts improved their understanding of the Visitor and Adapter patterns. Their response was gratifying as it supported primary motivation for our research. R3 and more details about our empirical evaluation can be downloaded at [42].

5.2.3 Threats to Validity

Every user study has limitations. First, although our results were comparable with undergraduate and graduate students, the results might not be translatable to more experienced programmers. Second, there might have been control loss due to the tasks being homework assignments. This was unavoidable considering the course design. The problem of reconciling classroom objectives and experimental designs has been largely recorded in the literature [6, 22]. Lastly, students were aware that R3 was developed by their instructors and, while we asked for their honest answers and were careful not to influence them on this point, this might have impacted the results.

5.3 Perspective

There are at least two dimensions that are not captured by our user study. There is a non-zero probability e that each manually performed refactoring will be erroneous. Assuming Bernoulli trials, Figure 17 shows the probability P = (1 − (1 − e)n) that one or more errors will occur in a manual retrofit of a design pattern requiring n refactorings. From Table 2 row W1, the value of n is 28. From Table 4 the value of P is 1 − 0.395 = 0.605. Solving 0.605 = (1 − (1 − e)n) yields e = 1/30.6. That is, our students made an error, on average, every 30.6 manual refactorings. The dashed vertical lines in Figure 17 and Figure 13 indicate the point on this graph that corresponds to our user study. Figure 17 predicts the results of additional
future user studies on RQ2. As refactoring tasks become more complicated, R3 wins easily; it can perform tasks correctly that humans can not.

A second dimension is time spent per refactoring task/script. We gave students only 1 manual refactoring task in our evaluation of RQ3. The real benefit is when a design pattern script is reused. Figure 18 shows that the break-even point of writing a script rather than manual pattern construction is on its third use. R3 wins easily on further reuse.

5.4 Other Relevant Observations on R3

R3 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.jdtui.tests.refactoring) tests that are relevant to R3 refactorings. We excluded tests on Java 8 features (e.g., lambda expressions), as R3 presently works on Java Runtime Environment (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. R3 satisfies all 464 extracted tests; they are now part of the R3 regression suite. Further, in building R2 and R3, we discovered and reported 29 bugs in the JDTRE, 7 of which have now been corrected.

Comparing the size of R3 to JDTRE in LOC is misleading, as JDTRE relies on layers of Eclipse functionality, whereas R3 is self-contained. To level the playing field, we used the EcLEmma code coverage tool to see what volume of code was executed by JDTRE and when the makeVisitor script runs – this gives us an estimate of the number of Unique LOC (ULOC) for equivalent functionalities.

R3 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, JDTRE 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. We conservatively estimate R3’s codebase to be 2× simpler than JDTRE.

6. RELATED WORK

We said earlier that R3 was inspired by elementary physics. Another inspiration was Intentional Programming (IP) [13]. IP is a structuring editor whose ASTs could be adorned with different pretty-print methods, allowing the contents of an AST to be printed textually or graphically. R3 is not

3. supports the writing and execution of refactoring scripts,
4. executes refactoring scripts 10× faster than JDTRE, and
5. significantly improves correctness when retrofitting design patterns in a user study.

Having said the above, R3 in no way eliminates the need for general-purpose program transformation systems. There are many refactorings that are not used in scripting design patterns [7,21] and there are many refactorings that cannot simply be “pretty-printed”, such as refactoring sequential legacy code into parallel code [16]. Never-the-less, standard OO refactoring engines leave a lot to be desired – slow speed, no support for scripting, and overly complex code bases. Our response is: let’s do the basics better and to provide scripting for the vast majority of programmers, which we believe is critical to next-generation OO refactoring engines. For these reasons, R3 offers a promising way forward.

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

[40] NetBeans 8.0.2. https://netbeans.org/


