Copyright
by
Jongwook Kim
2017

The Dissertation Committee for Jongwook Kim
certifies that this is the approved version of the following dissertation:

Reflective and Relativistic Refactoring with Feature-Awareness

Committee:

Don S. Batory, Supervisor

Danny Dig

Dewayne E. Perry

Donald S. Fussell

Milos Gligoric

Reflective and Relativistic Refactoring with Feature-Awareness

by

Jongwook Kim, B.E., M.S.COMP.SCI.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to Jungsub Lee

Acknowledgments

I would like to express my sincere gratitude to Don Batory. I will never forget the day in 2013 when he encouraged me by saying that he would be my quarterback and defender so that I would be able to run as far as possible. Indeed, he assumed various roles in addition to serving as my Ph.D supervisor during the last several years. I thank him for being the best possible "quarterback," directing our research projects to score touchdowns. I was fortunate in that he guided me to become more than a paper-making machine, but rather to become a researcher who notices original problems that might not necessarily be fully appreciated by others now or for years or decades to come. He also defended me against those who had no way of knowing whether our research ideas would work out. Needless to say, my doctorate would not have been possible without his support, guidance, and patience.

I was also very fortunate to have Danny Dig, an authority on software refactoring, as a committee member. Indeed, it was my privilege to be able to ask for his state-of-the-art comments, which he always gave me without hesitation.

I sincerely thank Dewane Perry for serving as a committee member for both my master's and Ph.D degrees. I enjoyed and have profited from listening to his experience in Software Engineering to see the path that previous SE

people have paved.

I greatly appreciate Donald Fussell's service as a committee member. It was my great pleasure to hear his comments on my refactoring engine, which was inspired by "rendering" used in Computer Graphics.

Milos Gligoric gave me fresh perspectives in both research and paper writing. In particular, it was revolutionary for me to learn how he completes research projects in such an efficient and successful way.

Last but not least, I thank my family: Yongwoon, Choonja, Youngjoon, Kyoungkyune, Younji, Juyoung, Yunkyung, Hyuntae, Sihyun, Dohyup, Jihwan, and Diana. Words alone cannot convey my appreciation and love for them.

Reflective and Relativistic Refactoring with Feature-Awareness

Publication No. ______________

Jongwook Kim, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Don S. Batory

Refactoring is a core technology in modern software development. It is central to popular software design movements, such as Extreme Programming [23] and Agile software development [91], and all major *Integrated Development Environments (IDEs)* today offer some form of refactoring support. Despite this, refactoring engines have languished behind research. Modern IDEs offer no means to sequence refactorings to automate program changes. Further, current refactoring engines exhibit problems of speed and expressivity, which makes writing composite refactorings such as design patterns infeasible. Even worse, existing refactoring tools for *Object-Oriented* languages are unaware of configurations in *Software Product Lines (SPLs)* codebases. With this motivation in mind, this dissertation makes three contributions to address these issues:

First, we present the Java API library, called `R2`, to script Eclipse refactorings to retrofit design patterns into existing programs. We encoded

18 out of 23 design patterns described by Gang-of-Four [57] as `R2` scripts and explain why the remaining refactorings are inappropriate for refactoring engines. `R2` sheds light on why refactoring speed and expressiveness are critical issues for scripting.

Second, we present a new Java refactoring engine, called `R3`, that addresses an Achilles heel in contemporary refactoring technology, namely scripting performance. Unlike classical refactoring techniques that modify *Abstract Syntax Trees (ASTs)*, `R3` refactors programs by rendering ASTs via pretty printing. AST rendering *never* changes the AST; it only displays different views of the AST/program. Coupled with new ways to evaluate refactoring preconditions, `R3` increases refactoring speed by an order of magnitude over Eclipse and facilitates computing views of a program where the original behavior is preserved.

Third, we provide a feature-aware refactoring tool, called `X15`, for SPL codebases written in Java. `X15` takes advantage of `R3`'s view rendering to implement a projection technology in *Feature-Oriented Software Development*, which produces subprograms of the original SPL by hiding unneeded feature code. `X15` is the first feature-aware refactoring tool for Java that implements a theory of refactoring feature modules, and allows users to edit and refactor SPL programs via "views". In the most demanding experiments, `X15` barely runs a second slower than R3, giving evidence that refactoring engines for SPL codebases can indeed be efficient.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiii**

List of Figures **xiv**

Chapter 1. Introduction **1**

Chapter 2. Scripting Parametric Refactorings in Java to Retrofit Design Patterns **5**

2.1 Introduction 5
2.2 Motivating Example 7
2.2.1 Separation of Concerns 9
2.2.2 Need for Other (Primitive) Refactorings 10
2.2.3 Limited Scope 11
2.3 Reflective Refactoring 13
2.3.1 Automating the Visitor Pattern 15
2.3.2 Automating the Inverse Visitor 17
2.3.3 More Opportunities 20
2.4 Other Patterns 21
2.4.1 Fully Automatable Patterns 22
2.4.2 Partially Automatable Patterns 23
2.4.3 Remaining Patterns 26
2.5 Evaluation 27
2.5.1 Experiment 28
2.5.2 Results 30
2.6 Related Work 34

Chapter 3. Improving Refactoring Speed by 10× **38**
3.1 Introduction 38
3.2 R3 Concepts 40
3.2.1 Modularity Perspectives 40
3.2.2 The R3 Database 43
3.2.3 Primitive Refactorings 46
3.2.3.1 Rename Method 46
3.2.3.2 Change Method Signature 47
3.2.3.3 Move Method via Parameter 48
3.2.3.4 Move Method via Field 48
3.2.3.5 Introducing New Program Elements 48
3.2.3.6 Scripting Refactorings 50
3.2.4 Preconditions 50
3.2.4.1 Boolean Checks Made by a Single Tuple Lookup 51
3.2.4.2 Checks that Require Database Search 53
3.3 Implementation 55
3.4 Evaluation 57
3.4.1 Performance 60
3.4.2 Practicality 64
3.4.2.1 Experimental Design 65
3.4.2.2 Results 67
3.4.2.3 Threats to Validity 69
3.4.3 Perspective 70
3.4.4 Other Relevant Observations on R3 71
3.5 Related Work 72

Chapter 4. Refactoring Java Software Product Lines **74**
4.1 Introduction 74
4.2 X15 Encoding of Java SPLs 77
4.2.1 Refactorings are Not Edits 83
4.3 Algebras of Feature Compositions 84
4.3.1 Sum and Projection of Feature Modules 85

4.3.2 Theorem for Refactoring SPLs 86
4.4 Feature-Aware Preconditions 89
4.4.1 Safe Composition 89
4.4.2 Preconditions for SPL Refactorings 90
4.5 Implementation Notes on X15 92
4.6 Evaluation 94
4.6.1 Experimental Set-Up 94
4.6.2 Results 97
4.6.2.1 Table Organization 97
4.6.2.2 Answers to Research Questions 100
4.6.3 Threats to Validity 103
4.7 Related Work 104
4.7.1 A Survey of SPL Tools 104
4.7.2 Variation Control Systems 105
4.7.3 Other Java Variabilities 106
4.7.4 Variability-Aware Compilers 108
4.7.5 Refactoring Variability-Aware Codebases 109

Chapter 5. Conclusions and Future Work **111**
5.1 R2: Practical Scripting of Refactorings 112
5.2 R3: 10× Speed Improvement 114
5.3 X15: Refactoring Java Software Product Lines 115
5.4 Lessons Learned 116
5.5 Future Work 118

Appendices **121**

Appendix A. Visitor Variants **122**

Appendix B. Feature-Aware Preconditions for Design Pattern Refactorings **124**

Appendix C. X15 Design Rules **127**

Appendix D. A Challenge of R3 Implementation **132**

Bibliography **134**

List of Tables

2.1 Methods of R2. 14
2.2 Automation Potential of Gang-of-Four Design Patterns. 23
2.3 Applications and Visitor Pattern Results. 29
2.4 Inverse-Visitor Results. 33
2.5 Tools and Languages to Script Refactorings. 35

3.1 Make-Visitor Comparison with JDTRE and R3 59
3.2 Inverse-Visitor Result Comparison. 63
3.3 Experimental Results from UT (44 undergrad students) 68
3.4 Experimental Results from OSU (10 grad students) 68

4.1 `makeVisitor` Results 96
4.2 `inverseVisitor` Results 99
4.3 Dead Code and Safe Composition Check Results 102
4.4 Comparing Capabilities of SPL Tooling 106

List of Figures

1.1 Visitor Pattern Mechanics from Kerievsky (pages 325–330) [79]. 2

2.1 A Visitor Pattern Refactoring. 8
2.2 A JDT Refactoring Being Too Smart 10
2.3 Restriction of JDT *inline* Refactoring 11
2.4 Rewrite that Uses Super Delegate 12
2.5 Super Field Access 13
2.6 R2 makeVisitor Method. 16
2.7 Methods Altered by Change Signature. 17
2.8 A Common Programming Scenario. 18
2.9 An inverseVisitor Method. 19
2.10 Visitor with State. 21
2.11 Another inverseVisitor Variant. 22
2.12 Factory Pattern. 24
2.13 A makeConcreteFactory Method. 24
2.14 Adapter Pattern. 25
2.15 A makeAdapter Method. 25
2.16 Method Seeds and Method Roots. 32

3.1 An Absolute Function and its Relative Methods. 41
3.2 Nested Classes 42
3.3 R3 Database. 44
3.4 RClass Display Method. 45
3.5 Move-via-Field Refactoring 49
3.6 Generic Constraints 52
3.7 Target Constraints 52
3.8 Super Call Bugs 53
3.9 Method Binding Change 54

3.10 Duplicate Type Parameter 55
3.11 R3 Pipeline. 56
3.12 A Java Program with a Generic Method 56
3.13 A Java Program with Manufactured-identifiers 57
3.14 Performance Pipeline of R3. 61
3.15 Reference Binding in R3. 64
3.16 Probability of Failure 71
3.17 Error Expended with Script Reuse 71

4.1 E-Shop Feature Model. 77
4.2 Annotated Codebase and Feature Modules 78
4.3 The *Feature* Annotation Type 79
4.4 *Feature* Annotations 80
4.5 Encoding Different Method Bodies 81
4.6 A Parse Tree with an *Feature* Annotation. 82
4.7 Code Folding in X15. 82
4.8 Problems in Refactoring Separate Codebases. 84
4.9 Key Theorem of SPL Refactoring. 87
4.10 Rename-Method Refactoring 88
4.11 Renaming *foo* to *bar* Fails 90
4.12 Binding Resolution Constraint 92
4.13 X15 Pipeline. 93
4.14 Translation `javapp` to `@Feature` Annotations 104
4.15 Parameter Removal by Projection 107

A.1 An example of Visitor Pattern 123

B.1 Inlining Constraint 124
B.2 Variable Capturing Constraint 125
B.3 Non-default Constructor Constraint 125
B.4 Singleton Constraint 126

C.1 C-Preprocessor vs Java SPL Idioms 130

D.1 Push Down m() to Multiple Subclasses. 132
D.2 Pull-Up/Push-Down m() Refactoring 133

Chapter 1

Introduction

Refactoring tools have revolutionized how programmers develop software. They enable programmers to continuously explore the design space of large codebases while preserving existing behavior.[1] Modern *Integrated Development Environments (IDEs)*, such as Eclipse, NetBeans, IntelliJ IDEA, and Visual Studio, incorporate primitive refactorings (e.g., rename, move, change-method-signature) in their top menu and often compete based on refactoring support. Refactoring is also central to popular software design movements, such as Extreme Programming [23] and Agile software development [91].

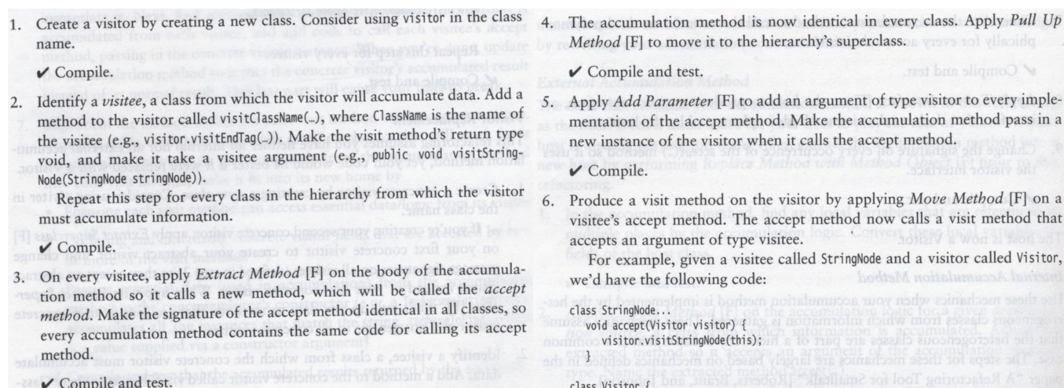
Design patterns are reusable solutions to design problems in *Object-Oriented (OO)* programming. In [57], the authors (referred to as the *Gang-of-Four (GoF)*) explore the capabilities and pitfalls of OO programming and introduce 23 design patterns. They describe the relationships and interactions between classes or objects of each pattern.

Despite vast interest and progress, a key functionality that many people have recognized to be missing in IDEs are refactoring scripts (e.g., [26,67,134]).

[1] Refactorings do not change program behavior by definition. However, that is not always true for current refactoring tools [64, 80, 82, 83, 127].

That is, IDEs do not support the writing of scripts that sequence refactoring steps to mechanize program changes.

It has been 20 years since design patterns were introduced [57] and even longer for refactorings [65, 103]. For at least 15 years, many design patterns were known to be expressible as a refactoring script – a programmatic sequence of refactorings [79, 140]. Contemporary texts on design patterns provide step-by-step descriptions, in which each step is typically a refactoring, on how each pattern can be introduced into a program [54, 57, 79]. Figure 1.1 shows a copy of a "mechanics" section for the Visitor pattern, the body of which describes informally how to retrofit a Visitor into an application, that moves all methods in a class hierarchy that have the same signature into a single class [79].



1. Create a visitor by creating a new class. Consider using `visitor` in the class name.

   ✔ Compile.

2. Identify a *visitee*, a class from which the visitor will accumulate data. Add a method to the visitor called `visitClassName(…)`, where `ClassName` is the name of the visitee (e.g., `visitor.visitEndTag(…)`). Make the visit method's return type void, and make it take a visitee argument (e.g., `public void visitStringNode(StringNode stringNode)`).

   Repeat this step for every class in the hierarchy from which the visitor must accumulate information.

   ✔ Compile.

3. On every visitee, apply *Extract Method* [F] on the body of the accumulation method so it calls a new method, which will be called the *accept method*. Make the signature of the accept method identical in all classes, so every accumulation method contains the same code for calling its accept method.

   ✔ Compile and test.

4. The accumulation method is now identical in every class. Apply *Pull Up Method* [F] to move it to the hierarchy's superclass.

   ✔ Compile and test.

5. Apply *Add Parameter* [F] to add an argument of type visitor to every implementation of the accept method. Make the accumulation method pass in a new instance of the visitor when it calls the accept method.

   ✔ Compile.

6. Produce a visit method on the visitor by applying *Move Method* [F] on a visitee's accept method. The accept method now calls a visit method that accepts an argument of type visitee.

   For example, given a visitee called `StringNode` and a visitor called `Visitor`, we'd have the following code:

```
class StringNode...
  void accept(Visitor visitor) {
    visitor.visitStringNode(this);
  }

class Visitor {
```

Figure 1.1: Visitor Pattern Mechanics from Kerievsky (pages 325–330) [79].

Reading these instructions, understanding them, and applying them are tedious, error-prone, and laborious. Even worse is repeating the same task every time a new pattern instance is introduced into a program. Thus, it is both surprising and disappointing that modern IDEs automate few patterns

and offer no means to script refactorings to introduce whole patterns.

A key question is: what language should be used to script refactorings? We found many proposals with distinguished merit [7, 22, 26, 29, 35, 67, 87, 94, 132, 145, 146] and all fall short in fundamental ways for our goal. *Domain Specific Languages (DSLs)* to write refactoring scripts have an unneeded overhead. They require knowledge of yet another programming language or programming paradigm. It is also unrealistic to expect that average programmers can quickly learn sophisticated *Program Transformation Systems (PTSs)* [22, 26, 28, 29] or utilities, such as *Eclipse Language Toolkits (LTKs)* [56], to manipulate programs. Although PTSs and LTKs are monuments of engineering prowess, their learning curve is measured in weeks or months.

For practicality, the language for scripting refactorings should be the same language whose programs are being refactored. Interestingly, this is a rarity (Section 2.6). We use Java to write scripts that will refactor Java programs. Further, our thought was to use existing IDE refactorings and provide a programming interface/façade that presents Java entity declarations (packages, classes, methods, fields, etc.) as Java objects, whose member methods are refactorings. This too is a rarity (Section 2.6).

Scripting refactorings should be fundamentally no different than having programmers import a Java package and use it to write Java programs (in this case, refactoring metaprograms a.k.a. scripts). This is our conjecture and our thesis. In this dissertation, we address these problems for which we have published new and novel solutions [80–82]. We developed a practical way to

move Java refactoring technology forward through three phases: implementation of (1) an API library for scripting refactorings, (2) a new refactoring engine that executes 10× faster than the Eclipse refactoring engine, and (3) an extension to this engine to refactor *Software Product Lines (SPLs)*. We present a user study that shows undergraduates can write refactoring scripts using our tool, which also improves the correctness of retrofitting a design pattern significantly.

Our solution uses Java as a metaprogramming language and exposes refactoring APIs through a Java package, where scripts become compact Java methods. There is no need for a DSL. Scalability of refactorings is an essential attribute required for large-sized (commercial) programs. Further, we extend our tool to develop the first feature-aware refactoring engine for Java SPLs using only Java custom annotations. We show how a modification of standard IDE code folding allows us to project SPL products as a 'view' of an SPL codebase. A programmer can edit and refactor products; behind the curtains the corresponding edits and feature-aware refactorings are applied to SPL codebase.

Chapter 2

Scripting Parametric Refactorings in Java to Retrofit Design Patterns

2.1 Introduction

Most design patterns are not present in a program during the design phase, but appear later in maintenance and evolution [79].[1] Modern IDEs – Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio – offer primitive refactorings (e.g., rename, move, change-method-signature) that constitute basic steps to retrofit design patterns into a program [57, 58]. It has been over 20 years since design patterns were popularized [57, 58] and longer still for refactorings [65, 103, 104]. For at least 15 years it was known that many design patterns could be automated by scripting transformations [79, 140]. So it is both surprising and disappointing that modern IDEs automate few patterns and offer no means to script transformations or refactorings to introduce whole patterns.

Manually introducing design patterns using primitive IDE refactorings is error-prone. To retrofit a Visitor pattern into a program requires finding

[1]The contents of this chapter appeared in "Scripting Parametric Refactorings in Java to Retrofit Design Patterns" [80], where I was the primary author of the three authors including Don Batory and Danny Dig. This paper was published in the 31st IEEE International Conference on Software Maintenance and Evolution.

all relevant methods to move by hand and applying a sequence of refactorings in precise order. It is easy to make mistakes. Missing a single method in a class hierarchy produces an incomplete but executable Visitor. But a future extension that uses the Visitor can break the program (Section 2.3.1).

We present a practical way to move Java refactoring technology forward. We designed, implemented, and evaluated *Reflective Refactoring (`R2`)*, a Java package whose goal is to encode the construction of classical design patterns as Java methods. Using Eclipse *Java Development Tools (JDT)* [48], `R2` leverages reflection by presenting a JDT project, its package, class, method and field declarations as Java objects whose methods are JDT refactorings. Automating design patterns becomes no different than importing an existing Java package (`R2`) and using it to write programs (in this case, refactoring scripts).

This chapter makes the following contributions:

- **JDT Extensions.** JDT refactorings, as is, were never designed to script design patterns. We describe our repairs to make JDT supportive for scripting.
- **OO Metaprogramming.** We present the Java package, `R2`, with several novel features to improve refactoring technology. `R2` objects are Java entity declarations and `R2` methods are JDT refactorings, primitive `R2` transformations, `R2` pattern scripts, and program element navigations (i.e., `R2` object searches).

- **Generality.** We encoded 18 out of 23 Gang-of-Four design patterns [57], inverses, and variants as short Java methods in `R2`, several of which we illustrate. This shows that `R2` can express a wide range of patterns.

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

- **Evaluation.** A case study shows the productivity and scalability of `R2`. We applied a 20-line `R2` script to retrofit 52 pattern instances into 6 real-world applications. One case invoked 554 refactorings, showing that `R2` scales well to large programs.

2.2 Motivating Example

Among the most sophisticated patterns is Visitor. There are different ways to encode a Visitor; we use the one below. Figure 2.1a shows a hierarchy of graphics classes; *Graphic* is the superclass and *Picture*, *Square*, *Triangle* are its subclasses. Each class has its own distinct *draw* method.

Mechanics. To create a Visitor for the *draw* method (Figure 2.1b), a programmer first creates a singleton Visitor class *DrawVisitor*. Next, s/he moves each *draw* method into the *DrawVisitor* class, renames it to *visit*, and adds an extra parameter (namely the class from which the method was moved). Referenced declarations (e.g., fields and methods) must become visible by changing their access modifiers after a method move [133]. Further,

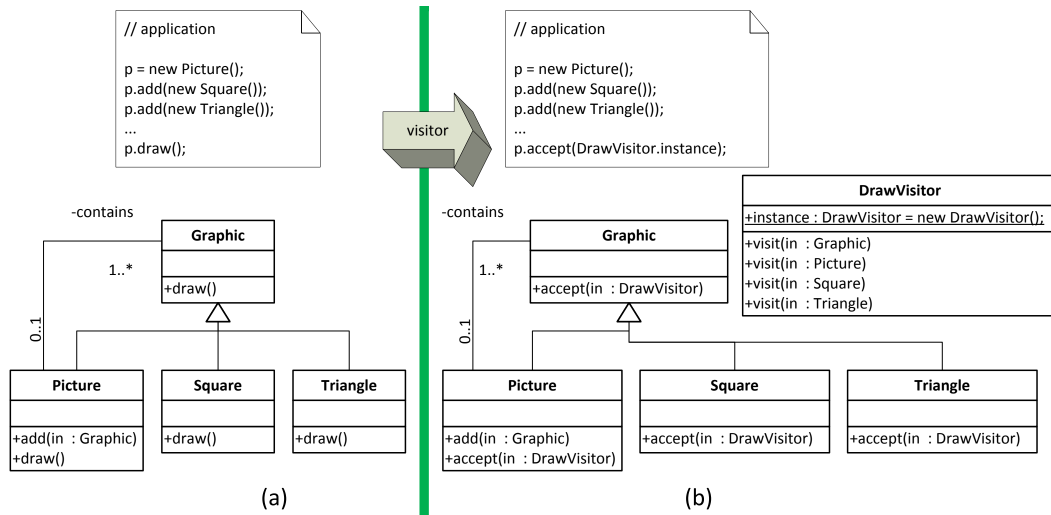


Figure 2.1: A Visitor Pattern Refactoring.

s/he creates a delegate (named *accept*) for each moved method, taking its place in the original class. The signature of the *accept* method extends the original *draw* signature with a *DrawVisitor* parameter and whose code for our example is:

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

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

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

Complicating Issues. JDT refactorings were never designed with scripting in mind. We encountered a series of design and implementation issues in the latest version of Eclipse JDT (Luna 4.4.1, Dec. 2014) [50] that compromises its ability to support refactoring scripts without considerable effort. These issues need to be addressed, regardless of our work. Here are examples.

2.2.1 Separation of Concerns

Figure 2.2a shows method *draw* in class *Square*, after a *DrawVisitor* parameter was added. Figure 2.2b shows the result of Eclipse moving *draw* from *Square* to *DrawVisitor* and leaving a delegate behind. Not only was the method moved, its signature was also optimized. Eclipse realizes that the original *draw* method did not need its *Square* parameter, so Eclipse simply removes it.

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

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

(a) Before

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

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

  void draw() {}
}
```

(b) After Moving *draw*

Figure 2.2: A JDT Refactoring Being Too Smart

As a refactoring, this optimization is *not* an error. But when an entire set of refactorings must produce a consistent result, it *is* an error. Preserving all parameters of moved methods in a Visitor pattern is essential. Two concerns – method movement and method signature optimization – were bundled into a single refactoring, instead of being separated into distinct refactorings. We programmatically deactivated method signature optimizations in R2; users cannot disable such optimizations from the Eclipse *GUI (Graphical User Interface)*.[2]

2.2.2 Need for Other (Primitive) Refactorings

Suppose that we want to "undo" an existing Visitor – eliminate the target Visitor class by moving its contents back into existing class hierarchies. Each *visit* method in the Visitor is moved back to its original class. As an

[2]What this really means is that it is not possible, in general, to use Eclipse refactorings as is to create the Visitor design pattern. Modifications, which we explain subsequent sections, are needed to Eclipse refactorings.

example, Figure 2.3a shows class *Triangle* after such a move: *Triangle* has both *accept* and *visit* methods. When the *visit* method is inlined, the *accept* method absorbs the *visit* method body (Figure 2.3b).

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

  void visit(DrawVisitor v) {
    if(true) return;
  }
}
```

(a) Before

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

(b) After Inlining *visit*

Figure 2.3: Restriction of JDT *inline* Refactoring

Unfortunately, Eclipse refuses to inline the *visit* method since a *return* statement potentially interrupts execution flow. However, it is not true for the example of Figure 2.3b. This precondition prevents automating a Visitor "undo". We had to deactivate this precondition check to script the Inverse-Visitor described in Section 2.3.2, in effect adding a new refactoring to JDT, to accomplish our task.

2.2.3 Limited Scope

A benefit of Visitor is that a single Visitor class enables a programmer to quickly review all variants of a method. Often, such methods invoke the corresponding method of their parent class. Moving methods with *super* calls is not only possible, it is desirable. Unfortunately, JDT refuses to move methods that reference *super*. It is not an error, but a strong limitation. We removed this limitation by replacing each *super*.x() call with a call to a manufactured

method $super_x\theta()$, whose body calls $super.x()$; θ is just a random number to make the name of the manufactured method unique.[3,4]

In Figure 2.4a, the *super* keyword invokes an overridden method $A.foo()$. We remove *super* by calling a delegate method which calls the overridden method $A.foo()$. Figure 2.4b shows a super delegate $super_foo\theta()$ which replaces the $super.foo()$ call in $B.bar()$, thus allowing JDT to move $B.bar()$ to the Visitor class. Of course, *super*-delegates throw the same exception types as its *super* invocation.

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

class B extends A {
  void foo() {}

  void bar() {
    super.foo();
  }
}
```

(a) Before

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

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

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

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

  void visit(B b) {
    b.super_fooθ();
  }
}
```

(b) After

Figure 2.4: Rewrite that Uses Super Delegate

[3]If $super.x()$ returns a result of type X, $super_x\theta()$ also returns type X.

[4]A unique name is needed for a refactoring that "undoes" or "removes" a Visitor (Section 2.3.2). It guarantees the correct *super*-delegate is called, as the meaning of *this* and *super* depends on the position in a class hierarchy from which it is invoked.

Now consider the use of *super* to reference fields of a parent class. Again, JDT refuses to move methods with *super*-references to fields. Here is how we fixed this: fields in Java are hidden and not overridden. So we can get *super* references simply by casting to their declared type. In Figure 2.5, method $B.foo()$ references field $A.i$ with the expression $super.i$. When $B.foo()$ is moved to class $Visitor$, expression $super.i$ is replaced with $((A)b).i$.

```
class A {
  int i;
}

class B extends A {
  int i;

  void foo() {
    super.i = 0;
  }
}
```

(a) Before

```
class A {
  int i;
}

class B extends A {
  int i;

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

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

  void visit(B b) {
    ((A)b).i = 0;
  }
}
```

(b) After

Figure 2.5: Super Field Access

2.3 Reflective Refactoring

Let $\mathcal{P}$ be a JDT project. We leverage the idea of reflection; `R2` defines class *RClass* whose instances are the class declarations in $\mathcal{P}$; instances of classes *RMethod* and *RField* are the method and field declarations of $\mathcal{P}$, and so on. When $\mathcal{P}$ is compiled, `R2` creates a set of main-memory database tables

(one for *RClass*, *RMethod*, *RField*, etc.) where each row corresponds to a class, a method, or a field declaration of $\mathcal{P}$. These tables are not persistent; they exist only when the JDT project for $\mathcal{P}$ is open.

The fields of *RClass*, *RMethod*, *RField*, etc. – henceforth called `R2` *classes* – also define association, inheritance, dependency relationships among table rows (foo is a method of class A, A is a superclass of B, B belongs to package C, etc.). The member methods of `R2` classes are JDT refactorings, simple `R2` transformations, composite refactorings (our scripts), and ways to locate program elements (i.e., `R2` objects). Representative methods are listed in Table 2.1.

R2 Type	**Method Name**	**Semantics**
RPackage	newClass	add a new class to the package
RClass	addSingleton	apply Singleton pattern to the class
	getAllMethods	return a list of `R2` objects that are all methods of theclass
	getPackage	return the `R2` object of its own package
	newConstructor	add a new constructor to the class
	newMethod	add a new method stub to the class
	newField	add a new field to the class
	setInterface	set to implement an interface
RMethod	getRelatives	return a list of `R2` objects of methods with the same signature
RRelativeList	addParameter	add a parameter with its default value to all methods
	moveAndDelegate	move methods to a class, leaving behind a delegate
	rename	rename all methods

Table 2.1: Methods of R2.

Internally, we leveraged XML scripts which Eclipse uses only to replay refactoring histories. An `R2` method call generates an XML script which we then feed to JDT to execute. In this way, we automate exactly the same procedures Eclipse users would follow manually. `R2` exposes every available JDT refactoring as a method and a few more (Section 2.2). Overall, we changed 51 lines in 8 JDT internal packages; the `R2` package consists of $\sim$ 5K LOC.

2.3.1 Automating the Visitor Pattern

Visitor is fully automatable as an `R2` script. Figure 2.6 is an `R2` script to create a Visitor design pattern. For a programmer to create a Visitor for some method m, s/he identifies a method called a "seed" in a class hierarchy that s/he wants to create a Visitor; 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 3–5). 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 8), 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 parameter. 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.

Looping through a list of methods and invoking a refactoring on each method would be the obvious way to add a parameter to relatives. But this is not how the JDT change-method-signature refactoring works (Line 10). It is applied to the seed method only. Consider Figure 2.7. Suppose $D.m$ is the method that seeds a change-method-signature. All m methods in D's class hierarchy $\{A.m, B.m, C.m, D.m\}$ and interconnected interface and class hier-

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

Figure 2.6: R2 makeVisitor Method.

archies $\{I1.m, I2.m, E.m\}$ are affected by this refactoring. That is, all of these methods (relatives) will have their signature changed. The *relatives* variable in Line 7 is the list of all methods in $\mathcal{P}$ whose signature will change. This list includes methods that cannot be moved, such as $interface$ and *abstract* methods. In this example, the methods moved into the Visitor are from classes $\{A, B, C, D, E\}$.

Note: Although Eclipse provides ways to find methods, it is still easy to miss program methods (relatives) that are distributed over the entire program. Forgetting to move a method when creating a Visitor manually is easy, yet it is hard to detect as no compilation errors identify non-moved methods. `R2` eliminates such errors by invoking a trustworthy `R2`

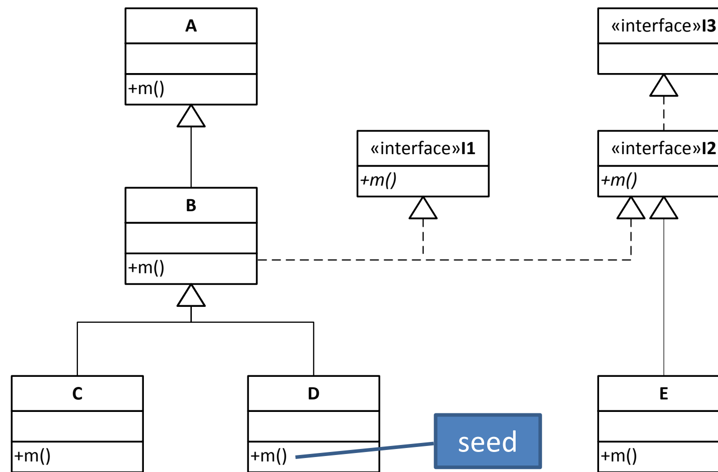


Figure 2.7: Methods Altered by Change Signature.

$getRelatives()$ method.

We show below how $makeVisitor$ is used in an example of Visitor pattern script where method $C.m()$ in package p of project R is the seed.

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

2.3.2 Automating the Inverse Visitor

Figure 2.8 depicts a common scenario: An R2 programmer creates a Visitor to provide a convenient view that allows her/him to inspect all *draw* methods in the graphics class hierarchy from our motivating example of Figure 2.1. The programmer then updates the program, including Visitor methods, as part of some debugging or functionality-enhancement process. At

which point, s/he wants to remove the Visitor to return the program back to its original structure.[5]

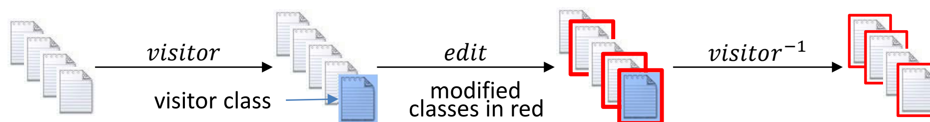


Figure 2.8: A Common Programming Scenario.

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

Figure 2.9 shows our *inverseVisitor*, a method of *RClass*, that moves *visit* methods back to their original classes and deletes the Visitor class. Here is how it works: Lines 8–9 recover the original class of a *visit* method. As we turned off method signature optimization in Section 2.2.1, the original class is encoded as the type of the *visit* method's first parameter. Line 11 moves the method back to its original class. Lines 13–14 inline *super*-delegates if

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

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

Figure 2.9: An inverseVisitor Method.

they exist by replacing each call to *super_x0*() with *super.x*() (Section 2.2.3) and then restore the original method body (which is the body of the *visit* method) by inlining. Lines 6–14 are performed for all *visit* methods. At this point, the *accept* methods (i.e., the delegate methods) contain the body of the original methods. Lines 17–20 collect all of the *accept* methods, remove their first parameter (of type Visitor class), and restore the original name of the method. The Visitor class is then deleted in Line 22.

Note: The challenge is to determine the correct order to apply move and inline refactorings. What if every *visit* method is moved and then inline is applied to each *visit*? To see the problem, let class A be the parent of class B and suppose both A and B have *visit* methods. Now, *B.visit* is inlined. B still inherits *A.visit*. Eclipse recognizes that inlining might alter program semantics and issues a warning: "method to be inlined overrides method from the parent class". A similar warning arises had *A.visit* been inlined first. The solution is to move one method at a time, followed by an inline, as done in Figure 2.9, to avoid warnings.

2.3.3 More Opportunities

Design patterns have many variations; Visitor is no exception. Consider Visitor PV of Figure 2.10 adapted from [136]. It differs from the Visitor of our example of Section 2.2 in several ways: PV is not a Singleton, it includes state $totalPostage$, it has a custom non-*visit* method $getTotalPostage()$, and at least one of its *visit* methods *visit*(*Book*) references $totalPostage$.

The Visitor variant of Figure 2.10 requires a slight modification of our `R2` $inverseVisitor$ method. Figure 2.11 shows the modified method; it differs from Figure 2.9 by moving only methods named *newName*, not removing the Visitor parameter, and not deleting the Visitor class.

These examples illustrate the power of `R2`: (1) we can automate these patterns (by transforming a program without these patterns into programs with these patterns), (2) we can remove these patterns (by transforming pro-

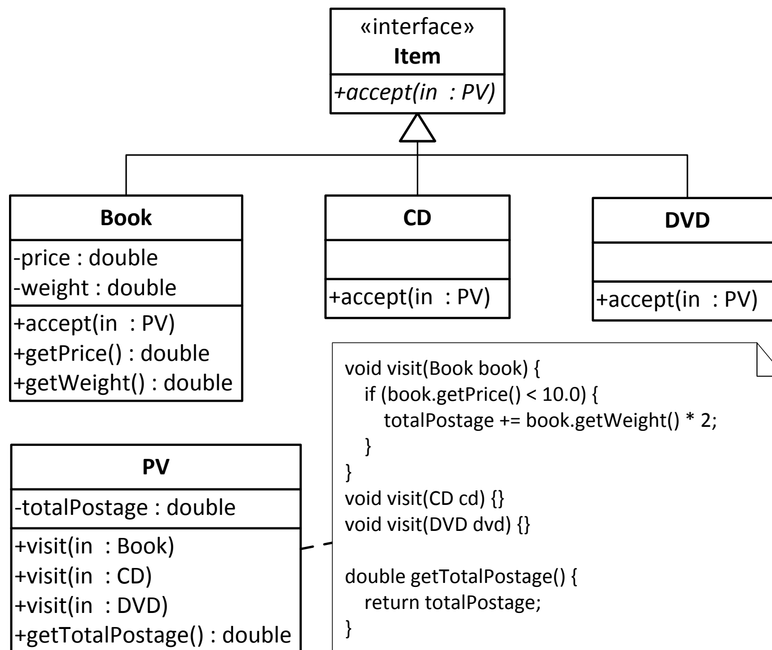


Figure 2.10: Visitor with State.

grams with hand-crafted patterns into programs without those patterns), and (3) express common variations that arise in design patterns. R2 offers a practical way to cover all of these possibilities.

2.4 Other Patterns

Table 2.2 is our review of the Gang-of-Four Design Patterns text [57]: 8 out of 23 patterns are fully automatable, 10 are partially automatable. For the remaining 5 patterns, we are unsure of their role in a refactoring tool (although some are automatable). R2 scripts for all of the 18 automatable patterns are listed in [110]. We elaborate our key findings below.

```
 1 // member of RClass class
 2 void inverseVisitorWithState(String originalName,
 3                              String newName) {
 4   RMethod aDelegate = null;
 5
 6   for(RMethod m : this.getMethodList(newName)) {
 7     aDelegate = m.getDelegate();
 8
 9     RParameter para = m.getParameter(0);
10     RClass returnToClass = para.getClass();
11
12     m.move(returnToClass);
13
14     m.inlineSuperDelegate();
15     m.inline();
16   }
17
18   RRelativeList relatives = aDelegate.getRelatives();
19   relatives.rename(originalName);
20 }
```

Figure 2.11: Another inverseVisitor Variant.

2.4.1 Fully Automatable Patterns

The Visitor pattern, its inverse and variants are fully automatable as they produce no "TO DOs" for a user. Another is Abstract Factory which provides an interface to concrete factories. Figure 2.12b shows interface *AbstractFactory* that exposes factory methods for every *public* constructor of each *public* class in a given package: the package of Figure 2.12a contains classes *A* and *B*; the interface *AbstractFactory* is implemented by concrete factory class *ConcreteFactory* in Figure 2.12b. Figure 2.13 is the `R2` method that produces a concrete factory for a package. A similar `R2` script creates the

Design Pattern	Automation Possibility		
	Full	Some	Unsure
Abstract Factory	✓		
Adapter		✓	
Bridge		✓	
Builder	✓		
Chain of Responsibility		✓	
Command	✓		
Composite		✓	
Decorator		✓	
Façade			✓
Factory Method	✓		
Flyweight	✓		
Interpreter			✓
Iterator			✓
Mediator			✓
Memento	✓		
Observer		✓	
Prototype		✓	
Proxy		✓	
Singleton	✓		
State			✓
Strategy		✓	
Template Method		✓	
Visitor	✓		
Total	8	10	5

Table 2.2: Automation Potential of Gang-of-Four Design Patterns.

AbstractFactory interface.

2.4.2 Partially Automatable Patterns

10 out of 23 patterns are partially automatable, i.e., these patterns produce “TO DOs” that must be completed by a user. The Adapter pattern is typical. It resolves incompatibilities between a client interface and a legacy class. Given interface *Target* and class *Legacy* in Figure 2.14, an intermediate class (called *Adapter*) adapts *Target* to *Legacy*.

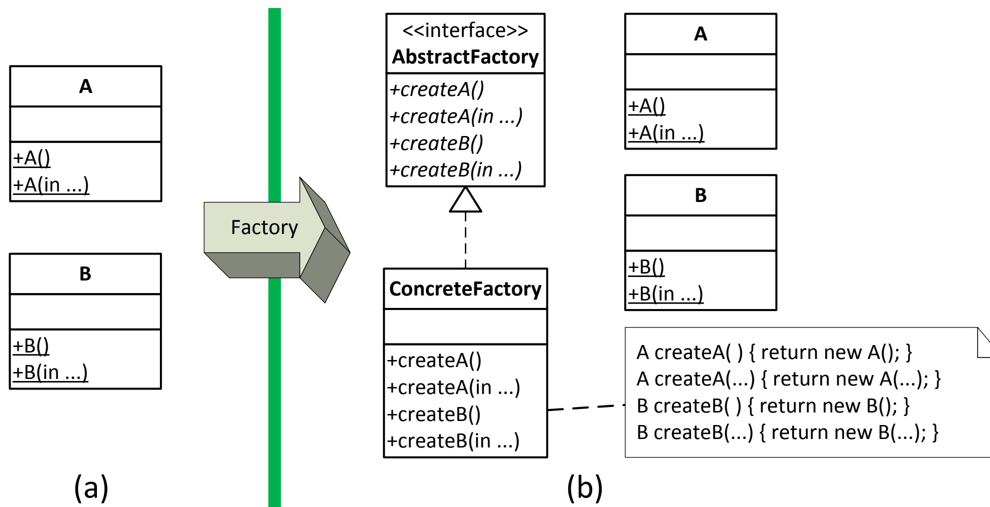


Figure 2.12: Factory Pattern.

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

Figure 2.13: A makeConcreteFactory Method.

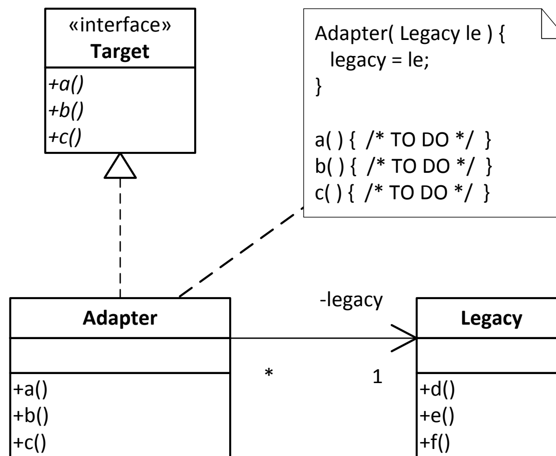


Figure 2.14: Adapter Pattern.

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

Figure 2.15: A makeAdapter Method.

Figure 2.15 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 3), to which is added a field named *legacy* of type c and a constructor to initialize *legacy* (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*. Programmers must provide bodies for the generated method stubs; these are the user "TO DOs". Although partially automated – method bodies are still needed – tedious and error-prone work is done by `R2`.

2.4.3 Remaining Patterns

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

- Façade is a convenient class abstraction for a package. Creating a façade requires deep knowledge of an application that only an expert, not a refactoring tool, will have. An `R2` script can be written to produce a particular façade, but it will be application-specific and unlikely to be reusable.

- Interpreter is common in compiler-compiler tools [13, 112]; given a language's grammar, a class hierarchy for creating language ASTs can be generated. Providing a grammar to a refactoring engine to generate a class hierarchy is possible, but seems inappropriate.

- State is a common application of *Model Driven Engineering (MDE)*. Given a statechart of a finite state machine, MDE tools can generate the class hierarchies and method stubs that implement the State pattern [12]. Again, providing a statechart to a refactoring engine to generate the code of a State pattern is possible, but also seems inappropriate.

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

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

2.5 Evaluation

We evaluated `R2` by answering two research questions:

- RQ1: Does `R2` improve productivity?

- RQ2: Can `R2` be applied to large programs?

Both questions address the higher level question "Is `R2` useful?" from different angles: Productivity measures whether `R2` methods save programmer time. Scalability measures whether `R2` can work with large programs.

2.5.1 Experiment

Some design patterns (e.g., Adapter) are relatively simple: create a few program elements, change class relationships, or make minor code changes. Others are different. All patterns are tedious and error-prone to create manually when the target program is non-trivial. There are `R2` scripts for all 18 automatable patterns. We evaluate `R2` using patterns that (a) exercise most `R2` methods and capabilities and (b) are difficult to create manually. These are the Make-Visitor and Inverse-Visitor patterns, which we have already presented.

We used six real-world Java applications that satisfied the following criteria: (1) they were publicly available, (2) they had non-trivial class hierarchies, (3) regression tests were available for us to determine if our refactorings altered application behavior, and (4) there were numerous method candidates that could "seed" a Visitor. We randomly selected methods among these candidates. We believe this selection process presents both a representative set of applications and a fair test for `R2`. The *Subject* column of Table 2.3 lists these applications, their versions, application size in LOC, and the number of regression tests. We used an Intel CPU i7-2600 3.40GHz, 16 GB main memory, Windows 7 64-bit OS, and Eclipse JDT 4.4.1 (Luna).

Seed ID	Subject (Ver#, LOC, #Tests)	Seed Method Name	Super Delegate	Change Signature	Move	Rename	# of Refactorings	Time	# of Errors
A1	**A**HEAD jak2java [13] (130320, 26K, 75)	getAST_Exp	0	26	26	52	104	72s	26
A2		getExpression	0	17	17	34	68	54s	17
A3		printorder	0	1	276	277	554	604s	0
A4		reduce2Ast	1	1	29	30	60	46s	23
A5		reduce2Java	7	1	47	48	96	84s	100
C1	Commons **C**odec [3] (1.8, 16K, 6103)	encode	0	1	2	3	6	5s	27
C2		getCharset	0	4	4	8	16	13s	0
C3		getDefaultCharset	0	4	4	8	16	12s	0
C4		getEncoding	0	2	4	6	12	10s	3
C5		isInAlphabet	0	1	2	3	6	5s	2
I1	Commons **I**O [4] (2.4, 24K, 810)	getDefaultEncoding	0	1	1	2	4	4s	0
I2		getEncoding	0	1	1	2	4	5s	0
I3		getFileFilters	0	1	2	3	6	5s	0
I4		getSize	0	1	1	2	4	5s	0
I5		setFileFilters	0	1	2	3	6	5s	0
J1	**J**Unit [74] (4.11, 23K, 2807)	countTestCases	1	1	7	8	16	13s	1
J2		failedTest	0	1	1	2	4	3s	0
J3		getName	0	4	5	9	18	40s	2
J4		run	2	1	9	10	20	20s	4
J5		testCount	0	1	1	2	4	4s	2
Q	**Q**uark [13] (1.0, 575, 9)	apply	0	1	7	8	16	13s	0
W1	Refactoring Cra**w**ler [42] (1.0.0, 7K, 15)	computeLikeliness	0	1	13	14	28	24s	14
W2		extractFully–	0	1	1	2	4	6s	0
W3		isRename	0	1	12	13	26	26s	10
W4		pruneFalsePositives	1	1	4	5	10	10s	1
W5		pruneOriginal–	7	1	13	14	28	25s	4

Table 2.3: Applications and Visitor Pattern Results.

2.5.2 Results

We have two sets of results: creating a Visitor and removing a Visitor. First consider creating a Visitor. Table 2.3 lists results of Make-Visitor applied to different methods in multiple applications. Each row represents data from a subject program. The columns are:

- $SeedID$ identifies the experiment.
- *Subject* is the Java subject program.
- $SeedMethodName$ is the seed of the Visitor.
- *SuperDelegate* is the number of *super*-delegates created (Section 2.2.3).
- $ChangeSignature$ is the number of change-method-signatures applied.
- $Move$ is the number of methods moved into the Visitor.
- *Rename* is the number of methods renamed.
- $\#ofRefactorings$ is the total number of JDT refactorings invoked by the $makeVisitor$ call.
- $Time$ is average clock time (in seconds) to perform $makeVisitor$.
- $\#ofErrors$ is the total number of errors created by JDT bugs in the old version of Eclipse (Juno 4.2.2 [49]) that we started with.

RQ1: Does R2 Improve Productivity? Table 2.3 shows that R2 performs tasks that are unachievable manually. Our largest experiment, $A3$, invoked 554 JDT refactorings took 10 minutes. Had programmers attempted $A3$ by hand, we believe that most would have given up at its sheer scale.

R2 offers a huge improvement in productivity even for programmers who are experts in JDT refactorings. An R2 script takes a fraction of the time (with no user intervention): the order in which refactorings should be sequenced, their parameters, and which refactorings to use has already been determined, in addition to choosing the "correct" options for refactorings (should there be options). The hard work has been done; R2 eliminates the errors and tedium of the process.

RQ2: Can R2 be applied to large programs? Table 2.3 clearly demonstrates that R2 can be applied to non-trivial programs. A number of these programs are more complicated than they appear as we explain below.

Recall $makeVisitor$ invokes $addParameter$ to the list of methods that are relatives of the method seed. Ideally, these relatives are descendant from a single root method ($A.m$ in Figure 2.16a). This means that the $addParameter$ invokes the JDT change-method-signature refactoring once on $A.m$ to add an extra parameter to all of its relatives $B.m$ and $C.m$.

In general, there can be multiple roots.[6] Figure 2.16b shows a seed whose relatives are not descendant from a single root. This means that the

[6]Some may argue that using multiple roots is too general; only one root should ever be used. This is programmatically adjustable within R2.

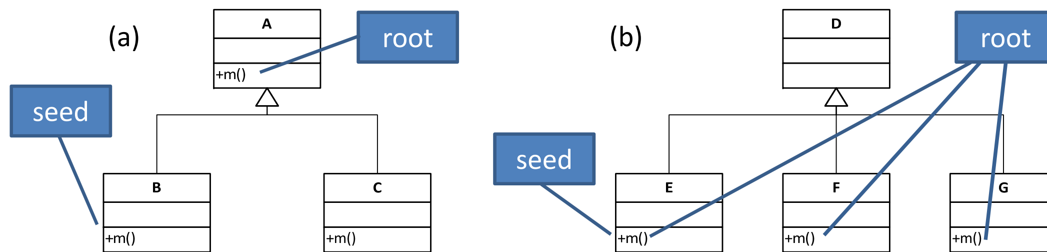


Figure 2.16: Method Seeds and Method Roots.

`R2` *addParameter* invokes change-method-signature refactoring three times, once for each root $E.m$, $F.m$, $G.m$, to add an extra parameter to all relatives. Programmers who apply JDT refactorings manually would have to realize this situation and make these extra renames.

Now look at row/experiment **A**3 in Table 2.3. Our tool created a Visitor for the *printorder* method in **A**HEAD. `R2` moved 276 methods into a Visitor, created no *super*-delegates, and applied one change-method-signature. The number of renames (277) was determined in this way: each method that is moved is renamed to *visit* (276). Although 276 method delegates were created, only one had to be renamed to *accept*. By renaming a root method, all of its descendants were renamed. Thus the total number of renames is $276 + 1 = 277$.

Now consider row/experiment **J**3. `R2` created a Visitor for the *getName* method in **J**Unit. `R2` moved 5 methods into a Visitor, created no *super*-delegates, and applied 4 change-method-signatures. The reason for 4 is that there were 4 method roots for the given seed (Figure 2.16b). Thus, the number of *renames* performed is 9; 5 methods were moved, and 4 (root) delegates were

renamed.

Finally, consider row/experiment **W**5. Our tool created a Visitor for the *pruneOriginalCandidates* method in RefactoringCra**w**ler. `R2` moved 13 methods into a Visitor, where these methods had 7 "*super*" references and thus required a *super*-delegate for each to be created.

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

Table 2.4: Inverse-Visitor Results.

Removing a Visitor. Table 2.4 lists the results of inverting (removing) the Visitors created in Table 2.3.

Consider row/experiment **A**5. Our tool removed a Visitor of the *reduce-2Java* in **A**HEAD. 47 visit methods were moved back to original classes. The

number of inlines (54) was determined in this way: each *visit* method that is moved is inlined (47) and 7 *super*-delegates are also inlined. Only one had to be renamed to its original name ($reduce2Java$) and removed a Visitor-type parameter. That is because, by changing a root method's signature, all of its descendants were updated. In addition, we turned off an inline precondition described in Section 2.2.2 for **A**4, **A**5, and **C**1. Note the difference in execution time between creating and removing a Visitor is due to different numbers and types of refactorings.

2.6 Related Work

Writing program transformations is a non-trivial exercise as research has shown [7,22,26,28,29,35,56,60,67,87,94,97,103,113,117,132,141,144–146]. Prior work introduced a number of impressive metaprogramming languages such as ASF+DSF [145], iXj [26], JunGL [146], Parlanse [22], Rascal [67], Refacola [132], SOUL [94], Stratego [29], Tom [7], and TXL [35]. None match our requirements.

There are two primary distinctions between `R2` and prior work. First, `R2` uses the base language – the language in which programs to be refactored are written – as the scripting language. Interestingly, the base and scripting language are identical only in Wrangler [87]; all others use a different scripting language (possibly even a different programming paradigm) than the base. The second is whether a user has to implement primitive refactorings in order to script them. Since writing primitive refactorings (e.g., rename, move, change-

Tool or DSL	Scripting Language	Base Language	Paradigm	Primitives Available
R2	Java	Java	Imperative	✓
Wrangler	Erlang	Erlang	Functional	
ASF+SDF	ASF, SDF	•	Term Rewrite	
iXj	iXj	Java	Imperative	
JunGL	JunGL	C#	Functional Logic Query	
DMS	Parlanse	•	Lisp-like	
Rascal	Rascal	Java	Imperative Non-OO	✓
Refacola	Refacola	Eiffel, Java	Constraint	
SOUL	SOUL	Java, C, Cobol, Smalltalk	Logic Programming	✓
XT	Stratego	•	Term Rewrite	
Tom	Tom	C, Java, Python, C++, C#, etc.	Term Rewrite	
TXL	TXL	•	Functional Term Rewrite	
Codelink	(GUI-based)	•	(N/A)	
SmaCC	SmallTalk	Java, C#, Delphi	Imperative	

• indicates arbitrary languages that can be defined by users.

Table 2.5: Tools and Languages to Script Refactorings.

method-signature) is non-trivial, it is important to distinguish approaches that can leverage existing refactoring engines from those where primitives need to be written by users. Only SOUL and Rascal (besides R2) satisfy the second criterion. Table 2.5 categorizes these distinctions to the best of our knowledge.

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

Program transformation systems are monuments of engineering prowess.

Among them are Codelink [141], DMS [22], SmaCC [28], Wrangler [87], and XT [29]. Wrangler, mentioned earlier, is a tool (refactoring framework) implemented in Erlang which is also the base language. Wrangler supports refactoring commands for locating program elements and provides a custom DSL to execute the commands.

Like `R2`, Rascal [67] also uses JDT refactorings, which are available as APIs in the Rascal JDTRefactoring library. They too target Java, but their scripting language (Rascal) is not an OO language. Further, manual code changes are required in their transformation process to fix incorrect access modifiers, clean up unnecessary codes, etc., which we would have preferred to be automated.

SOUL [94] uses declarative metaprogramming to define design patterns and their constraints in a language-independent manner. Their use of a variant of Prolog is elegant, as they tackle problems similar to `R2`.

Moreover, `R2` deals with scripting high-level refactorings, *not* with recommending when and which refactorings to apply or detecting existing refactorings. There are excellent papers [16–21, 24, 34, 40, 63, 92, 95, 96, 115, 122, 124, 128, 138, 142] on this, but all are orthogonal to the use and goals of `R2`.

Finally, refactoring research has grown enormously in the last decade. Traditional refactorings improve design, like `R2`. More recent refactorings improve non-functional qualities (e.g., energy consumption [114]), address more challenging languages (e.g., Yahoo! Pipes [135]), or use novel paradigms to

check refactoring safety [32]. These works are beyond the scope of `R2`.

Chapter 3

Improving Refactoring Speed by 10×

3.1 Introduction

In R2, we added scripting to Eclipse JDT, exposing the core declarations of a Java program (packages, classes, methods, etc.) as objects whose methods are JDT refactorings.[1] Refactoring scripts that add or remove design patterns are short Java methods. R2 is an Eclipse plug-in that uses the *JDT Refactoring Engine (JDTRE)* as it represents state-of-the-practice in refactoring. However, experiments revealed JDTRE is ill-suited for scripting for three reasons:

- **Reliability.** JDTRE is buggy [64, 127]. We filed 39 new bug reports to date, but only a fraction has been fixed in the latest version of Eclipse [47]. 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

[1]The contents of this chapter appeared in "Improving Refactoring Speed by 10X" [82], where I was the primary author of the four authors including Don Batory, Danny Dig, and Maider Azanza. This paper was published in the 38th ACM/IEEE International Conference on Software Engineering.

or years for a repair [47]. We rediscovered a bug that took 5 years to be fixed [45]. **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 (Section 2.2.3). We also had to turn off parameter optimization, for example, to make JDT refactorings produce design patterns correctly (Section 2.2.1).

- **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.

The contributions of this chapter are:

- A novel foundation (R3) of database+pretty printing for designing a new generation of refactoring engines that support scripting,
- R3's codebase is a mere 4K LOC and does not use LTK [56] utilities,
- Efficient ways to evaluate refactoring preconditions: boolean properties of ASTs are harvested during database creation where precondition checks consult their values and the database supports fast searches,
- An empirical evaluation of R3 on 6 case studies executed 52 scripts. R3 runs at least 10× faster on average, in two cases 285× faster than JDTRE, and
- A user study involving 2 classes (44 undergraduates and 10 graduates) showed R3 improved the success rate of retrofitting design patterns by 25% up to 50%.

3.2 R3 Concepts

3.2.1 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 refactors 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 3.1a is the signature of an absolute function foo with three parameters whose types are B, C, D.

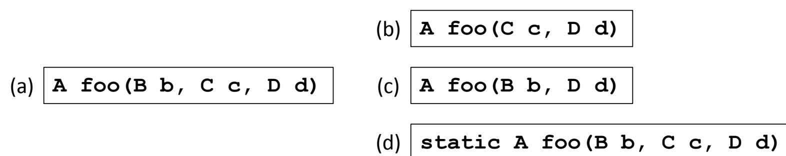


Figure 3.1: An Absolute Function and its Relative Methods.

If foo is displayed as a member of class B, Figure 3.1b is its signature: the B parameter becomes *this* and is otherwise implicit. If foo is displayed as a member of class C, Figure 3.1c 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 3.1d 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 gen-

eralize absolute functions in an interesting way. Figure 3.2a 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). We see that $m()$ can be displayed as a member of class A using our modularity perspective techniques by making the B parameter explicit. See Figure 3.2b.

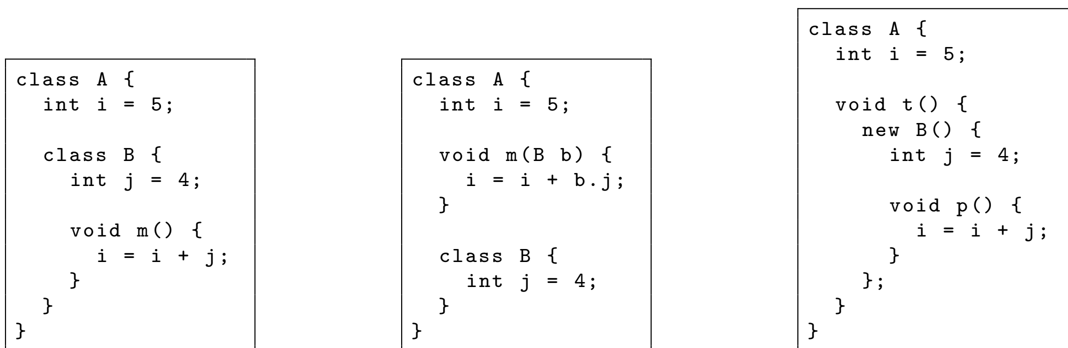


Figure 3.2: Nested Classes

A 'coordinate transformation' interpretation also explains why refactoring engines do not move methods of anonymous classes. Consider Figure 3.2c. 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 3.1 is about, is a coordinate transformation for software; it preserves the semantic properties of a program. The same holds for other primitive refactorings.

3.2.2 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

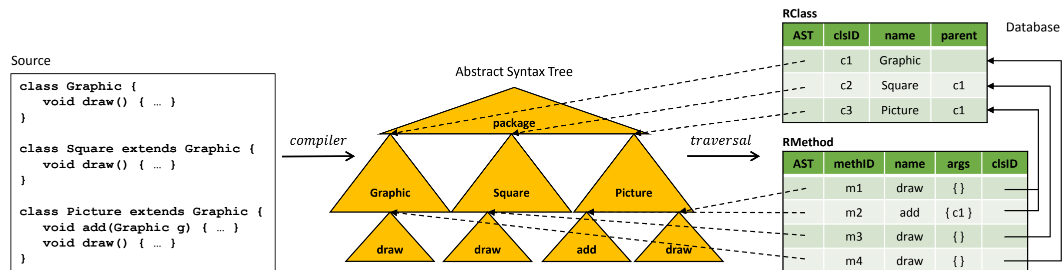


Figure 3.3: R3 Database.

`R3` tables. Figure 3.3 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 3.4 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 clos-

ing brace '}'. `R3` reproduces the original order in which members appeared for ease of subsequent reference by programmers and preserves all source code comments.

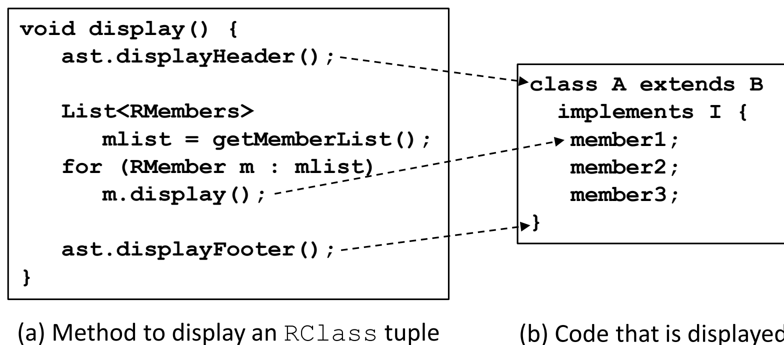


Figure 3.4: RClass Display Method.

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 3.1a) 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.

3.2.3 Primitive Refactorings

We now explain some representative primitive refactorings to see how they are implemented in R3. All R3 methods are listed in [111]. In the refactoring community, behavior preservation is determined by statically analyzing whether the input code passes the refactoring's preconditions [103]. 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).

3.2.3.1 Rename Method

Rename-instance-method refactoring modifies the *name* field of the method's $RMethod$ 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:

```
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:

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

3.2.3.2 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 [100,143] 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 type serves as the type 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 methods in *relatives* list. Line 3 makes it the first parameter of *relatives* methods:

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

3.2.3.3 Move Method via Parameter

The move-instance-method refactoring in `R3` changes the home class of a method m. Recall that 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 `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:

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

3.2.3.4 Move Method via Field

The move-via-field refactoring is illustrated in Figure 3.5. 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:

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

A member method *addParameter* of *RMethod*, whose access modifier if *private*, is invisible to `R3` users.

3.2.3.5 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 decla-

```
class A {
  D d;

  C m(B b) {}

  C c = m(b); // a call
}

class D {}
```

(a) Before

```
class A {
  D d;

  C c = d.m(this, b); // a call
}

class D {
  C m(A a, B b) {}
}
```

(b) After Moving via Field d

Figure 3.5: Move-via-Field Refactoring

rations. 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*:

```
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 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.

3.2.3.6 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 2.15), $makeVisitor$ (Figure 2.6), and $moveViaField$ in Section 3.2.3.4.

3.2.4 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 $obj.\rho_1() \wedge\ obj.\rho_2() \wedge \ldots \wedge obj.\rho_n()$ where $\rho_i()$ 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 $obj.\rho_1() \wedge obj.\rho_2() \wedge \ldots \wedge obj.\rho_n()$ 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 – $\rho_i()$ *for all* R3 objects at database build time, *even though we may never use these values*. For each $\rho_i()$, we add a boolean attribute to R3 tables to indicate whether a tuple's AST satisfies $\rho_i()$. The checks for a refactoring then become

a conjunction of these boolean attributes.

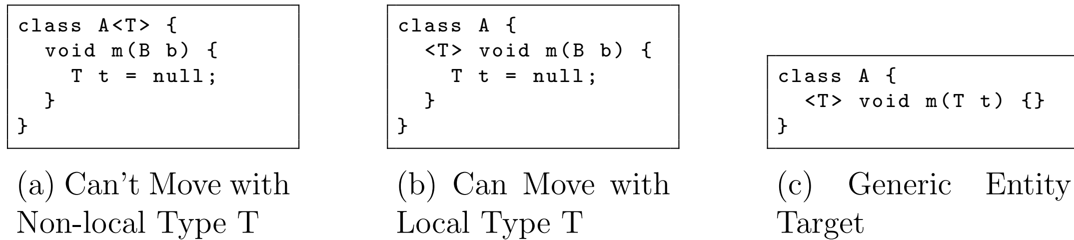
The `R3` database is created by traversing the ASTs of a program and collecting semantic information. Doing so populates the `R3` database with tuples and assigns boolean values to these checks. Further, in cases where harvested boolean values are insufficient, we optimized the `R3` database to facilitate fast searches, e.g., `R3` collects all references of a declaration to reduce search overhead. We will see in Section 3.4 these techniques improve performance significantly.

3.2.4.1 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 3.6a illustrates a non-local type parameter which prevents

a move of method m. In contrast, method m in Figure 3.6b can be moved as its parameter is local.



(a) Can't Move with Non-local Type T

(b) Can Move with Local Type T

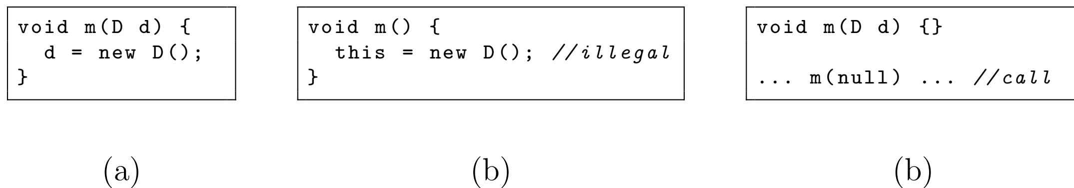
(c) Generic Entity Target

Figure 3.6: Generic Constraints

- **Generic Entity Target** – moving a method via a type parameter is disallowed (Figure 3.6c).

- **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 3.7a since the assignment statement becomes illegal in Java after move (Figure 3.7b).

- **Null Home Value** – if a method call has a *null* home parameter as in Figure 3.7b, a move to that home is disallowed as it will dereference *null*.



(a)

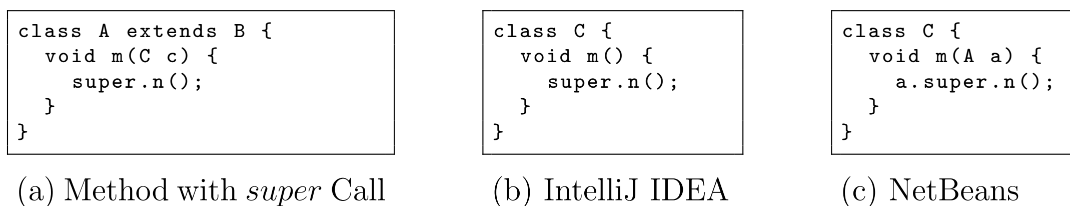
(b)

(b)

Figure 3.7: 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.

- **Super Reference** – JDTRE refuses to move any method that uses the *super* keyword. To write general purpose refactoring scripts, we removed this precondition in both `R2` and `R3` (Section 2.2.3). Other IDEs, such as IntelliJ IDEA [71] and NetBeans [102], do move such methods, but do so erroneously (Figure 3.8).



(a) Method with *super* Call (b) IntelliJ IDEA (c) NetBeans

Figure 3.8: Super Call Bugs

3.2.4.2 Checks that Require Database Search

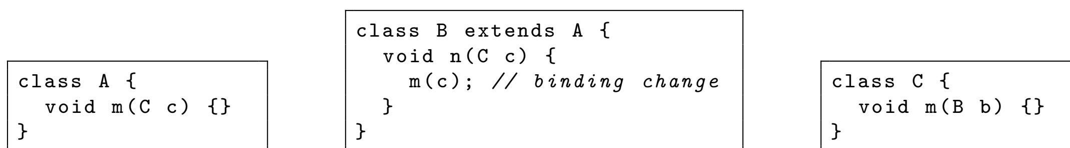
Not all primitive preconditions are reducible to boolean attributes; these outliers require a database search, which `R3` performs efficiently. Here are some for the move-instance-method:

- **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 *RMethod* 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 3.9. A method call $m(...)$ inside $B.n(C)$ 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 $B.m(C)$.



```
class A {
  void m(C c) {}
}
```

```
class B extends A {
  void n(C c) {
    m(c); // binding change
  }
}
```

```
class C {
  void m(B b) {}
}
```

Figure 3.9: 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 [46]. R3 does better by traversing the class hierarchy and evaluating access modifiers to find conflicts [119].

- **Duplicate Type Parameter** – JDTRE moves method m in Figure 3.10 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 its binding to the existing T in class B.

```
class A {
  <T> void m(B<T> b) {
    T t = null;
  }
}

class B<T> {}
```

(a) Before

```
class A {}

class B<T> {
  void m() {
    T t = null;
  }
}
```

(b) After Moving m

Figure 3.10: 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.

3.3 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 [13], which has pretty-print methods ideal for R3. Figure 3.11 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. Figure 3.12 is a target program with a generic method that prints its array argument of different types.

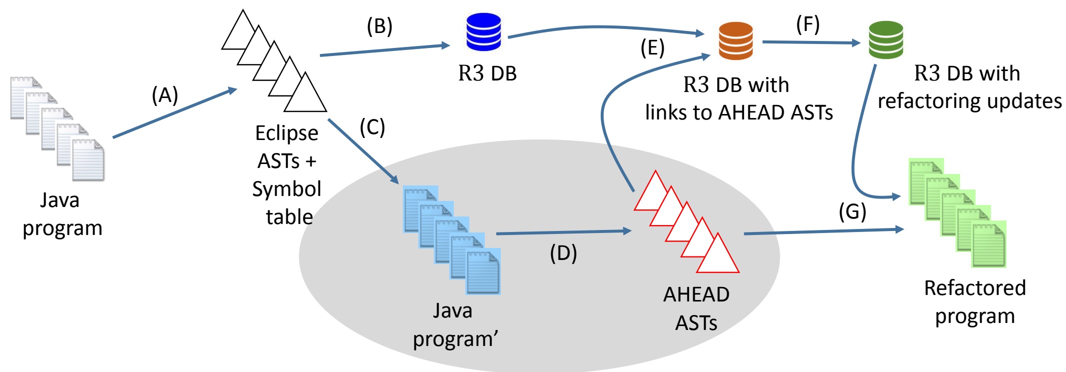


Figure 3.11: R3 Pipeline.

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

Figure 3.12: A Java Program with a Generic Method

(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 (Figure 3.13) 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 and with the database of (B) can reconstruct the *identical* 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\n", ID_12);
  }
}
```

Figure 3.13: A Java Program with Manufactured-identifiers

(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.

3.4 Evaluation

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, we evaluated the *expressiveness* of 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, 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 programmer errors, while the case studies give more confidence that R3 generalizes to real-world situations.

Application (Ver#, LOC, #Tests)	Seed ID	# of Refacs	JDTRE time (seconds)			R3 time (seconds)						Speed Up
			Precon Check	Perform Change	Total	Build DB (B)	Link AST (E)	Precon Check (F1)	DB Update (F2)	Proj (G)	Total	
AHEAD jak2java [13] (130320, 26K, 75)	A1	104	16.58	2.31	18.89	1.66	0.06	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 Codec [3] (1.8, 16K, 6103)	C1	6	1.80	1.39	3.19	1.18	0.03	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 IO [4] (2.4, 24K, 810)	I1	4	1.20	0.19	1.40	1.75	0.04	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 [74] (4.11, 23K, 2807)	J1	16	4.49	0.70	5.20	2.01	0.04	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 [13] (1.0, 575, 9)	Q	16	3.40	0.40	3.80	0.24	0.01	0.000	0.009	0.09	0.10	40
Refactoring Crawler [42] (1.0.0, 7K, 15)	W1	28	6.99	0.90	7.90	0.79	0.02	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 3.1: Make-Visitor Comparison with JDTRE and R3

3.4.1 Performance

The first column of Table 3.1 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 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. These experiments engage the primitive refactorings 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 their behavior. We used an 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 3.1 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. 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[2] and (2) the CPU time listed in the *Total* column.[3]

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 3.2.4 *and* a check/parse

[2]Our *makeVisitor* and *inverseVisitor* scripts create and delete program elements but these operations are not counted as JDT refactorings.

[3]We used profiling tool *VisualVM* (ver. 1.3.8) [147] to measure CPU times in running the JDTRE and `R3` scripts. We repeated each experiment five times and report the average execution time.

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 3.14. 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.

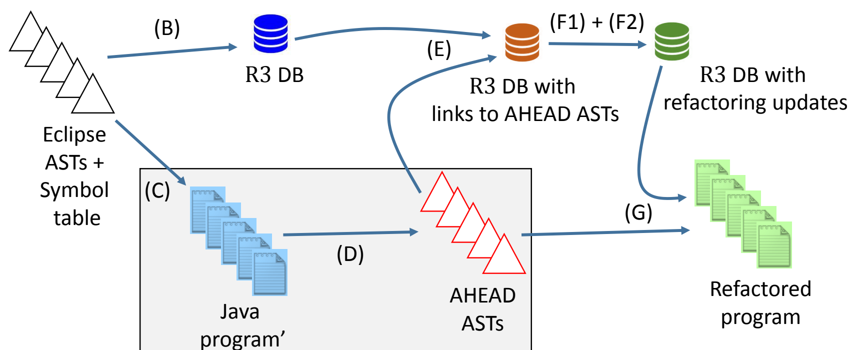


Figure 3.14: Performance Pipeline of R3.

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 2.3. These execution times are minuscule. 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 databse, 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 seed **A**3 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 JDTRE.[4]

Table 3.2 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 JDTRE. On average, `R3` was 55× faster than JDTRE.[5]

There are three basic reasons for the huge difference in performance. First, as mentioned earlier, JDTRE 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,

[4]Had we included database creation time for steps (B) and (E) in our calculations, the average speed-up ratio drops to 11×.

[5]Had we included database build time for steps (B) and (E) in our calculations, the average speed-up ratio drops to 10×.

Seed ID	# of Refa	JDTRE time (seconds)			R3 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 3.2: Inverse-Visitor Result Comparison.

adjusting declaration visibility, etc. Profiling experiments indicate that the vast majority of time (*avg*: 60%, *sd*: 15%) of the *Precon Check* column for JDTRE is simply due to AST searching. `R3` reduces the overhead by collecting all program elements and values needed for precondition 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 rebinding of all references to one declaration to those of another (Figure 3.15a). 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 3.15b).

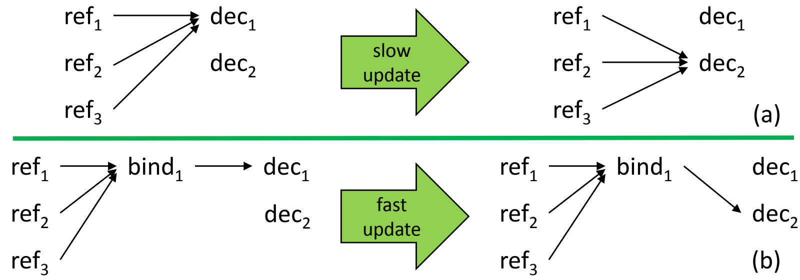


Figure 3.15: Reference Binding in R3.

Third, JDTRE 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, JDTRE was not designed for efficient scripting.

3.4.2 Practicality

We conducted an evaluation of R3's practicality. We designed two controlled experiments (the Adapter experiment and the Visitor experiment) as course assignments to assess how users worked with R3.[6] We ran the experiments with 44 students in Spring 2015 at the undergraduate CS373S Software Design [129] course at the University of Texas at Austin. The course exposes

[6]Our experiments were not research projects but formed into homework assignments.

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 CS561 Advanced Software Engineering [1] 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.

3.4.2.1 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 representa-

tive 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 [80],
- 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, $RefactoringCrawler$ [42], 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 **W**1.

Group B applied Eclipse refactorings manually to make the same Visitor. In 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.

3.4.2.2 Results

Tables 3.3 and 3.4 summarize the results we obtained from the UT and OSU executions respectively. As Shapiro-Wilk tests showed a significant

	Visitor						
Metric	**Baseline**		**R3**		**z**	**p**	**r**
Success	39.5%		78.0%		3.441	0.001	0.519
	Mean	**SD**	**Mean**	**SD**			
Score	73.5	24.8	93.5	13.6	3.629	0.000	0.547
Time	37.2	29.7	91.8	46.9	4.918	0.000	0.741

	Adapter						
Metric	**Baseline**		**R3**		**z**	**p**	**r**
Success	54.5%		81.8%		3.207	0.001	0.483
	Mean	**SD**	**Mean**	**SD**			
Score	96.0	5.2	97.9	5.1	2.315	0.021	0.349
Time	19.9	9.2	43.7	27.2	5.152	0.000	0.777

Table 3.3: Experimental Results from UT (44 undergrad students)

	Visitor						
Metric	**Baseline**		**R3**		**z**	**p**	**r**
Success	20.0%		70.0%		2.236	0.025	0.707
	Mean	**SD**	**Mean**	**SD**			
Score	56.0	39.2	91.0	12.9	2.176	0.030	0.688
Time	66.6	38.3	92.1	37.7	2.075	0.038	0.656

Table 3.4: Experimental Results from OSU (10 grad students)

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 3.4.3.

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 3.4.3.

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.

3.4.2.3 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 [14, 55]. 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.

3.4.3 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 3.16 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 3.1 row **W**1, the value of n is 28. From Table 3.3, the value of P is $1 - 0.395 = 0.605$. Solving $0.605 = (1 - (1 - e)^{28})$ 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 3.16 and Figure 3.17 indicate the point on this graph that corresponds to our user study. Figure 3.16 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 3.17 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.

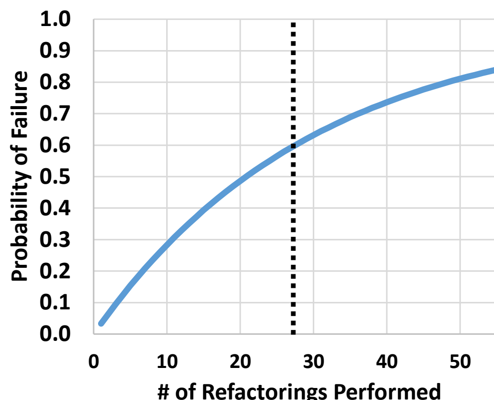


Figure 3.16: Probability of Failure

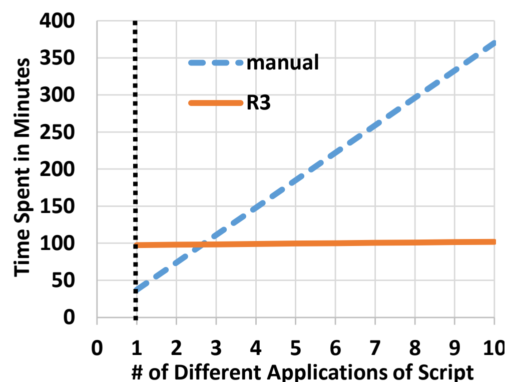


Figure 3.17: Error Expended with Script Reuse

3.4.4 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.jdt.ui.tests.refactoring* [105]) 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.[7] Further, in building `R2` and `R3`, we discovered and reported 39 bugs in the JDTRE, 7 of which have now been corrected [47].

[7]`R3` does not produce exactly the same refactored source as JDTRE. For example, `R3` 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.

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 [68] to see what volume of code was executed by JDTRE and R3 when the $makeVisitor$ script runs – this gives us an estimate of the number of *Unique LOC (ULOC)* executed 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.[8] We conservatively estimate R3's codebase to be 2× simpler than JDTRE.

3.5 Related Work

We said in Section 3.2.1 that R3 was inspired by elementary physics. Another inspiration was *Intentional Programming (IP)* [33]. IP is a structure 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* a structure editor or a small tweak on IP. IP displays entire trees; R3 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. The phi-

[8]Example: see checkInitialConditions, checkFinalConditions, and createChange methods in `MoveInstanceMethodProcessor.java` [99]

losophy 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 (e.g.,, rename and move) 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 in the same function) and IP could easily distinguish them.

In developing `R2` [80], we found 13 prior works [7, 22, 26, 28, 29, 35, 67, 87, 94, 132, 141, 145, 146] that could be used to implement refactoring scripts. We classified them as program transformation systems, DSLs, 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 refactorings or transformation scripts. Most research on refactoring engines mentions the importance of refactoring reliability or error detection [38, 64, 73, 86, 127].

Like `R2`, a critical property of `R3` 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 crucial because programmers do not have to learn yet another language or programming paradigm to write refactoring scripts. As we discussed in Section 2.6, only one prior tool had this property: Wrangler [86]. Wrangler refactorings and refactoring scripts were written in Erlang to modify Erlang programs.

Chapter 4

Refactoring Java Software Product Lines

4.1 Introduction

An SPL is a family of related programs [5, 125, 130].[1] Amortizing the cost to design and maintain their commonalities makes SPLs economical [5]. Programs of an SPL are distinguished by *features* — increments in program functionality. Each program, henceforth *product*, in an SPL is defined by a unique set of features called a *configuration* [5].

Variability in a SPL codebase relies on *presence conditions*, a predicate expressed in terms of features, that indicate when a fragment of code, file or package is to be included in an SPL product [5]. A typical use-case is with `#if-#endif` preprocessor constructs: if the presence condition of `#if` is *true* for a configuration, the content that is enclosed by `#if-#endif` is included in the product; otherwise it is erased [5]. The Linux Kernel is a huge SPL, consisting of 8M LOC and over 10K features [90, 125]. It uses the *C-preprocessor* (CPP) to remove code and files to produce the C codebase for a

[1] The contents of this chapter appeared in "Refactoring and Retrofitting Design Patterns in Java Software Product Lines" [81], where I was the primary author of the three authors including Don Batory and Danny Dig. This paper was published as a technical report in the Department of Computer Science at University of Texas at Austin.

configuration.

The presence or absence of a feature in Java can be encoded by a global `static boolean` declaration; the Java compiler can evaluate feature predicates to remove unreachable code in `if(feature_expression)` statements. But removing entire declarations (packages, types, fields, and methods) is not possible with existing Java constructs. So Java SPLs are hacked in some manner to achieve this additional and essential effect.

Preprocessing is the standard solution [72, 108, 126], although officially Java shuns preprocessors [53]. Another way is to copy and assemble code fragments from an SPL codebase $\mathbb{P}$ to produce an SPL product P_C where C is P_C's configuration [6, 13, 25, 76]. Both create a separate codebase for P_C that a user edits to improve, tune, and repair P_C. Doing so exposes two critical problems in SPL tooling.

First, given an edited product P_C, how are its edits propagated back to $\mathbb{P}$, the SPL codebase? Early SPL tools [13, 25] had back-propagation capabilities. Furthermore, there are many prototype tools for projecting CPP codebases to 'view' codebases that can be edited and their changes back-propagated to $\mathbb{P}$ (see [131, 148] for surveys). But none correctly propagates changes from P_C to $\mathbb{P}$ *made by refactorings*. Why? Renaming a field in P_C is easy, but not all references to the field reside in P_C; other references may exist in $\mathbb{P}$ that are not in P_C. Thus, back-propagating edits will rename some, but not all, references to a field, breaking $\mathbb{P}$. *In short, SPL back-propagation tools must become 'refactoring-aware'* [43].

Second, the first refactoring engine for C-language SPLs appeared in `2015`, offering the inline, rename, and extract refactorings [88]. One might ask: why did this tool not appear a decade earlier? There are many reasons: (i) Existing SPL tools rely on preprocessors that lack type information needed for precondition checks and code transformations of refactorings, (ii) Special-purpose compilers for main-stream languages integrated with CPP constructs are hard to build [22, 77, 88, 149], (iii) Refactoring engines are also hard to build, and (iv) SPL tool and refactoring engine integration requires a rare combination of both.

We present `X15`, the first feature-aware refactoring engine for Java that solves the above problems. `X15` (i) uses a standard Java compiler, (ii) relies on Java custom annotations to encode SPL variability in a simple and intuitive way, (iii) incorporates code folds of an SPL codebase to produce a 'view' of an SPL product that programmers can edit and refactor; behind the curtains `X15` applies corresponding edits and feature-aware refactorings to $\mathbb{P}$. The novel contributions are:

- The `X15` tool for editing, projecting, and refactoring Java SPLs and their products;
- Identifying primitive refactoring preconditions that must become feature-aware; and
- Case studies that apply $2,316$ refactorings in 8 Java SPLs and show `X15` is as efficient, expressive, and scalable as state-of-the-art feature-unaware

refactoring engine R3.

4.2 X15 Encoding of Java SPLs

Every feature-based SPL has a *feature model* (FM) that defines the features of an SPL and their relationships (mandatory vs. optional; alternative vs. multiple-choice) [5]. Figure 4.1 depicts an E-Shop FM with a single cross-tree constraint that *CreditCard* implies *High* [5,39,150]. It is well-known that FMs can be mapped to a propositional formula where features are the boolean variables [5, 9]. Each solution to this formula — a `true` or `false` assignment to every variable — defines a combination features that uniquely identify a product in an SPL. A common name for a solution is a *configuration*.

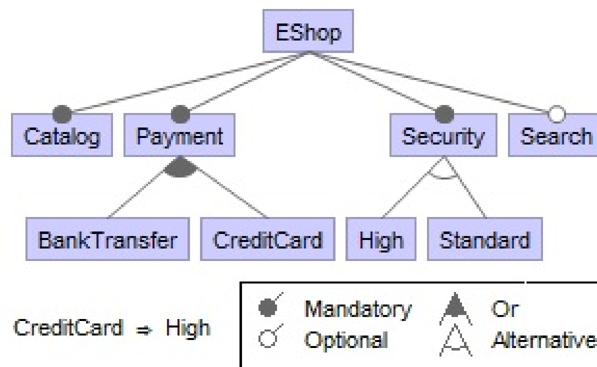
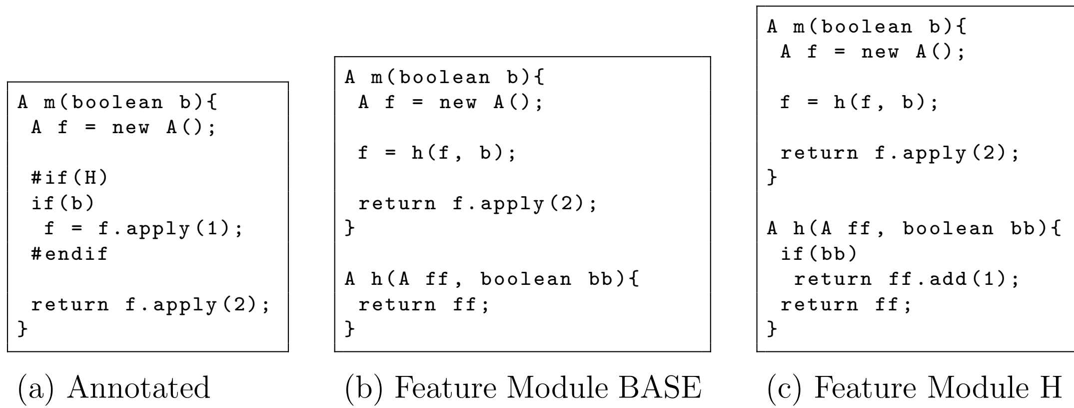


Figure 4.1: E-Shop Feature Model.

Feature modules can be implemented in many ways, ranging from pre-processor or annotative means [76, 137] to specialized languages that support explicit feature modules and their composition [6, 13, 27, 118].

Explicit feature modules have the advantage of clean encapsulations of

large (packages, classes) and medium scale (fields, methods, method wrappers) program declarations in Java; annotative approaches do better when feature-specific code fragments are tiny and modularizing them as explicit methods clutters designs [76].



```
A m(boolean b){
 A f = new A();

 #if(H)
 if(b)
  f = f.apply(1);
 #endif

 return f.apply(2);
}
```

(a) Annotated

```
A m(boolean b){
 A f = new A();

 f = h(f, b);

 return f.apply(2);
}

A h(A ff, boolean bb){
 return ff;
}
```

(b) Feature Module BASE

```
A m(boolean b){
 A f = new A();

 f = h(f, b);

 return f.apply(2);
}

A h(A ff, boolean bb){
 if(bb)
  return ff.add(1);
 return ff;
}
```

(c) Feature Module H

Figure 4.2: Annotated Codebase and Feature Modules

Figure 4.2a shows a tiny code fragment in the middle of method m that appears when feature H is selected. This fragment is equivalent to a composed pair of feature modules, $BASE$ and H, in Figure 4.2b-c. Figure 4.2b shows the $BASE$ module that lifts this code fragment into a tiny method, h, to define the default do-nothing action. The H module in Figure 4.2c overrides h with the revised definition [6, 13, 118]. Figure 4.2a encodes two distinct products: one with H absent and one with H present. In a feature module approach, these codebases are represented by: $BASE$ and $BASE + H$. In feature modularizing legacy applications, it has been observed that methods often have many such optional code fragments, causing their feature modules to have many tiny methods [76]. In summary, features can be used in

both feature-module implementations and annotation-based implementations of SPLs.

`X15` relies on annotation-based implementations of SPLs. `X15` uses the Java custom annotation type *Feature* to encode a configuration file. Every feature F of an SPL has a *static boolean* variable F declared inside *Feature* whose value indicates whether F is selected (*true*) or not ($false$). Figure 4.3 shows a *Feature* declaration with three features X, Y, and Z where X and Y are selected and Z is not. The specified configuration is $\{X,Y\}$. *Feature*.`java` is generated by a feature model configuration tool [5, 9].

```
@interface Feature {
  static final boolean X = true;
  static final boolean Y = true;
  static final boolean Z = false;

  boolean value();
}
```

Figure 4.3: The *Feature* Annotation Type

`X15` uses Java's built-in annotative means to encode variability. (Doing so exposes basic SPL design rules or guidelines for `X15` SPLs, which are described in Appendix C.) Let $\mathbb{P}$ denote the code base of a Java SPL. Every Java declaration (class, method, field, constructor, initializer) in $\mathbb{P}$ has an optional *Feature* annotation with a *boolean* expression of *Feature* variables.[2] If the expression is *true* for a configuration, the declaration is present in that configuration's product; otherwise it is not. If a declaration has no *Feature* annotation, it is included in every product of the SPL.

Figure 4.4a shows three declarations: *Graphics*, *Square*, and *Picture*.

[2]Package-level annotations in Java are placed in a `package-info.java` file.

Interface *Graphics* belongs to every program of the SPL as it has no *Feature* annotation. *Square* is added by feature X. *Picture* is added whenever a pair of features, Y and Z, are both present.

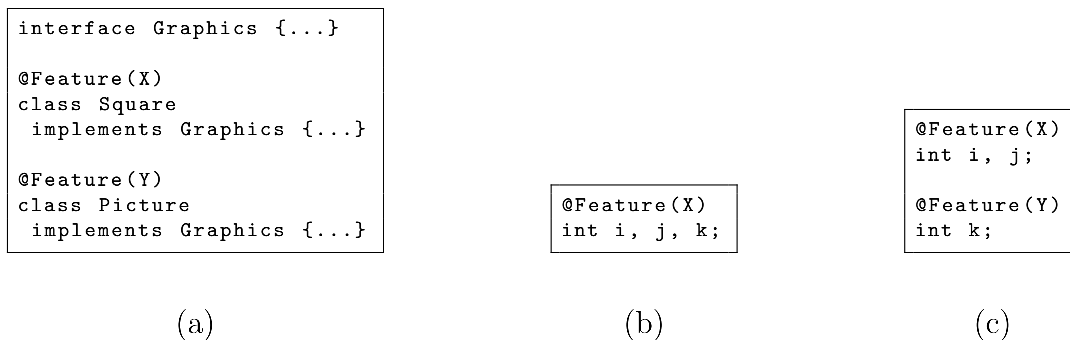


Figure 4.4: *Feature* Annotations

Figure 4.4b shows a declaration of three integer fields i, j, k, all belonging to feature X; the *Feature* annotation is for the entire line. If fields i and j belong to feature X, and k to feature Y, Figure 4.4c is used.

Variability in executable code is written using `if(feature_expression)` statements. For example, it is common to have different bodies for a single method in an SPL. Suppose features X and Y are never both selected. Figure 4.5a is a CPP encoding that introduces at most one declaration of method m in any program; Figure 4.5b shows the cascading *if-else* statements used in `X15` to encode the same variability inside one declaration of m.

Here is how `X15` works: It parses $\mathbb{P}$ and looks for the parse tree of `Feature.java`, from which it extracts the boolean value for each feature. These values define the *current configuration* C.

```
#if(X)
  int m() { return 1; }
#elif(Y)
  int m() { return 2; }
#else
  int m() { return 0; }
#endif
```

(a)

```
int m() {
  if(X)       return 1;
  else if(Y) return 2;
  else        return 0;
}
```

(b)

Figure 4.5: Encoding Different Method Bodies

Let P_C be the source of the SPL product with configuration C. Figure 4.6 sketches the parse tree of a `Feature`-annotated class declaration of $\mathbb{P}$. `X15` sees the `Feature` annotation and evaluates the feature expression knowing the current configuration. If the expression is *true* then `X15` pretty-prints the parse tree including the *Feature* declaration (minus code fragments that are configuration-disqualified). If the expression is $false$, `X15` comments-out the source of the entire parse tree, effectively erasing the entire declaration. This is how `X15` *projects* $\mathbb{P}$ w.r.t. C to produce P_C. `X15` *never* changes a parse tree during a projection.

`X15` uses projection in two distinct ways. One projection is sent to the Java compiler to produce an executable. The second projection relies on the standard IDE functionality of 'code folding', where code that is not part of P_C is hidden in a code fold. A code fold indicates the location of a *variation point* (VP) — where code in some SPL product is known to exist, but is not present in P_C [5]. Code folds also provide a practical way for programmers to edit code that is visible (i.e., code that belongs to P_C). Programmers can inspect, but *not* edit, folded contents. Figure 4.7a shows $\mathbb{P}$, Figure 4.7b shows P_C with

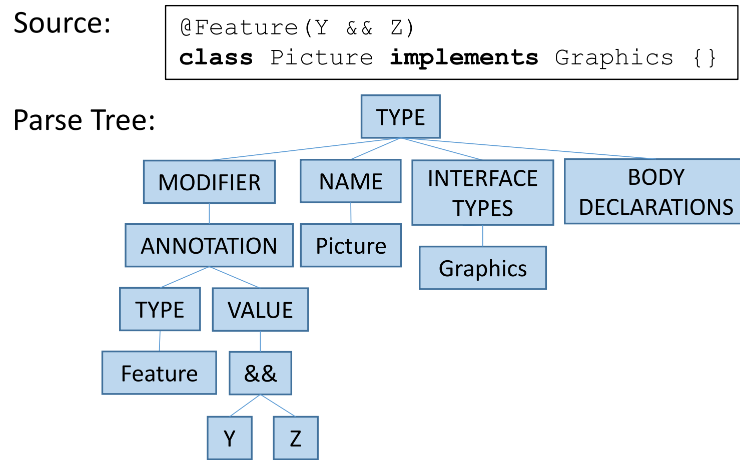


Figure 4.6: A Parse Tree with an *Feature* Annotation.

folded code when $BLUE{=}false$, and Figure 4.7c shows P_C with unfolded code when $BLUE{=}false$.[3]

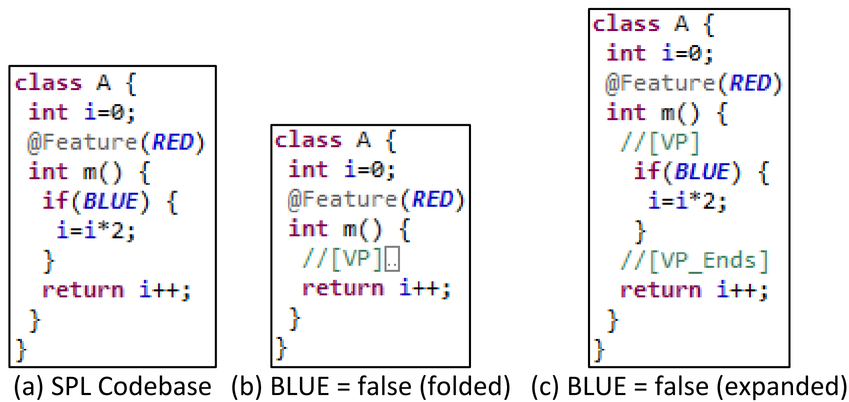


Figure 4.7: Code Folding in X15.

Together, both projections provide a useful end-user functionality: an

[3]Of course, there are situations where to correctly edit P_C, programmers must edit $\mathbb{P}$. Suppose a programmer wants to provide a new body to an existing method. To do so, s/he must edit $\mathbb{P}$ to achieve the desired projection. `X15` offers a GUI button for users to toggle between editing $\mathbb{P}$ and P_C, should the need arise. [131] has other examples.

SPL programmer can see and edit a 'view' (projection) of P_C, the SPL product of the current configuration. Further, s/he can compile P_C and debug it through the code-folded projection, giving the impression that the SPL programmer is editing, debugging, and developing a single product P_C, even though behind the curtains edits are being made directly to $\mathbb{P}$.

4.2.1 Refactorings are Not Edits

If refactorings were just text edits, we would be done. A programmer invokes a refactoring on product P_C, the code of P_C is changed and the edits are made directly to $\mathbb{P}$. End of story.

The problem is that *refactorings are more than text edits*. Consider the SPL codebase $\mathbb{P}$ of Figure 4.8a. The separate codebase P_X for configuration $\{X\}$ is Figure 4.8b. Figure 4.8c shows P_X after renaming $Grafix$ to $Graphics$. The problem is evident in Figure 4.8d: propagating *text* changes made to P_X back to $\mathbb{P}$ breaks $\mathbb{P}$ because not all occurrences of $Grafix$ in $\mathbb{P}$ are renamed to $Graphics$ — the program for configuration $\{Y\}$ no longer compiles.

In a nutshell, *text-edit back-propagation tools for SPLs are not 'refactoring-aware'; they are inadequate to deal with the changes made by refactorings.* Dig and Johnson demonstrated an analogous problem for version control [43]. In effect, future SPL tools must provide 'refactoring aware' back-propagation. The key insight to achieve 'refactoring awareness' is discussed next.

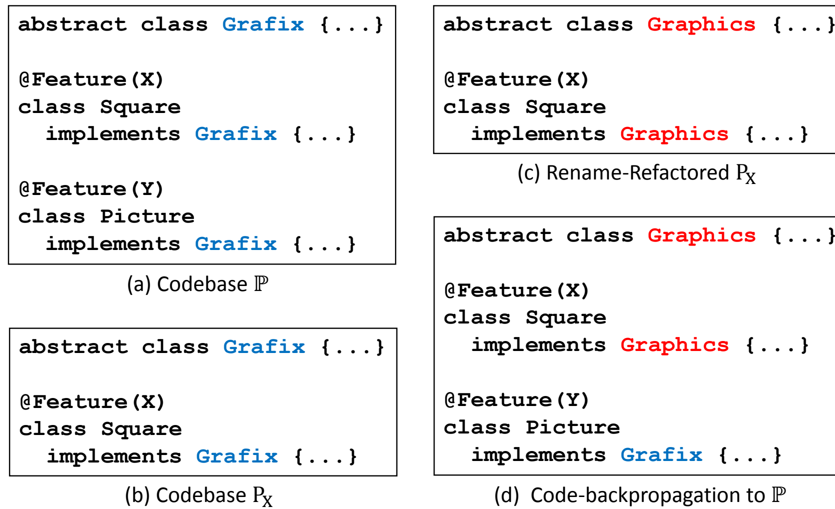


Figure 4.8: Problems in Refactoring Separate Codebases.

4.3 Algebras of Feature Compositions

Features have long been viewed as the conceptual modules or building blocks of SPL products. Early research (AHEAD [13], FeatureHouse [6], DOP [118]) not only developed algebras for feature compositions, but also invented OO language extensions to define concrete feature modules. While the ideas behind these language extensions — role-based programming, mixin-layers, and context-oriented programming — have been widely explored, they have not yet caught on. In an annotative approach, the code fragments of a feature are distributed throughout codebase $\mathbb{P}$. A feature module, in contrast, collects these same fragments in a single package-like structure. So any theorem that can be proven using feature modules should hold for both annotative and feature-module implementations. These algebras provide insight on how OO refactoring engines and back-propagation tools can become feature aware. We

sketch known ideas and then present the insight that made `X15` possible.

4.3.1 Sum and Projection of Feature Modules

Algebras axiomatize the summation or composition of feature modules to produce SPL products [10, 11].[4] The ideas are simple and can be informally conveyed; see citations for details.

A *feature module* F_i encapsulates the implementation of feature i. Product P_C with configuration C is produced by summing the modules of its features [6, 13, 118]. Thus, if $C = \{X, Y, Z\}$ where X, Y, and Z are features, product P_C is:

$$P_C \;=\; \sum_{i \in C} F_i \;=\; F_X + F_Y + F_Z \tag{4.1}$$

Let $\mathbb{F}$ be the set of all features. The codebase $\mathbb{P}$ of an SPL is:[5]

$$\mathbb{P} \;=\; \sum_{i \in \mathbb{F}} F_i \tag{4.2}$$

Projection, as discussed in Section 4.2, is a complementary operation to summation. The C-projection of $\mathbb{P}$ yields P_C:

$$\Pi_C(\mathbb{P}) \;=\; P_C \tag{4.3}$$

[4] A cross-product of features exposes the submodules of features that arise from feature interactions [10, 123]. Cross-products rely on module summation, and are otherwise orthogonal to this paper.

[5] A common name for $\mathbb{P}$ is a *150% design* – it includes all possibilities.

Think of projection as the operation that eliminates feature modules that do not belong to C. Let C_1 and C_2 be different sets of features from the same SPL (i.e., $C_1, C_2 \subseteq \mathbb{F}$). An axiom that relates projection and summation is:

$$\Pi_{C_1}(\sum_{i \in C_2} F_i) \;=\; \sum_{i \in C_1 \cap C_2} F_i \qquad (4.4)$$

Equation (4.1) follows from (4.2) and (4.4):

$$\begin{aligned} \Pi_C(\mathbb{P}) &= \Pi_C(\sum_{i \in \mathbb{F}} F_i) && // \ (4.2) \\ &= \sum_{i \in C \cap \mathbb{F}} F_i && // \ (4.4) \\ &= \sum_{i \in C} F_i && // \ \textit{where } C \subseteq \mathbb{F} \end{aligned}$$

As said in Section 4.2, `X15` implements projection in two different ways: Π_C^{fold} code-folds $\mathbb{P}$ to expose only the code of P_C for viewing, editing and refactoring. $\Pi_C^{comment}$ comments-out unnecessary code which is then fed to the Java compiler to produce bytecodes for P_C; this compiled version enables programmers to execute, debug, and step-through the code folded version of P_C.

4.3.2 Theorem for Refactoring SPLs

The unknown is this: how do refactorings extend the algebras of feature compositions? No one to our knowledge has answered this question before; an answer will tell us how SPL codebases can be refactored.

Let $\mathcal{R}$ be a refactoring. If we $\mathcal{R}$-refactor P_C, we get $P_C^{\mathcal{R}}$:

$$\mathcal{R}(\ \Pi_C(\mathbb{P})\ ) \;=\; \mathcal{R}(P_C) \;=\; P_C^{\mathcal{R}} \qquad (4.5)$$

As $\mathcal{R}$ changes P_C, $\mathcal{R}$ must also change $\mathbb{P}$. But how? Our conjecture and theorem is this: $P_C^{\mathcal{R}}$ can be computed by the $\mathcal{R}$-refactoring of $\mathbb{P}$ followed by a C-projection:

$$\Pi_C(\ \mathcal{R}(\mathbb{P})\ )\ =\ P_C^{\mathcal{R}} \tag{4.6}$$

Equivalently, (4.6) is the commuting diagram of Figure 4.9 where the operations of projection and refactoring commute [106].

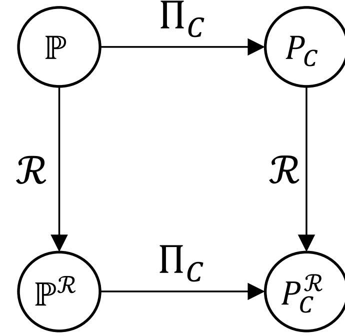


Figure 4.9: Key Theorem of SPL Refactoring.

SPL programmers must realize that refactoring an SPL codebase $\mathbb{P}$ has more constraints than just refactoring a single product P_C. We explain in Section 4.4.2 that the preconditions to $\mathcal{R}$-refactor $\mathbb{P}$ imply the preconditions to $\mathcal{R}$-refactor P_C. Our proof of (4.6) assumes the preconditions to $\mathcal{R}$-refactor $\mathbb{P}$ are satisfied. Therefore $\mathcal{R}$ in (4.6) really represents the code transformation that is made by an $\mathcal{R}$ refactoring.

We observed the following distributivity identity over years of developing feature-based SPLs: the $\mathcal{R}$-refactoring of a sum of feature modules A and B equals the sum of the each $\mathcal{R}$-refactored feature module:

$$\mathcal{R}(A+B)\ =\ \mathcal{R}(A)+\mathcal{R}(B) \tag{4.7}$$

This axiom is intuitive: *common refactorings are largely oblivious to feature module boundaries.* That is, when a program $P = A + B$ is $\mathcal{R}$-refactored, one expects both modules A and B to be modified by $\mathcal{R}$, namely $P^{\mathcal{R}} = A^{\mathcal{R}} + B^{\mathcal{R}}$.

Example: Method m in Figure 4.10 is defined in class/feature A. Class/feature B calls m. When m is renamed to n, both features A and B are modified to $A^{\mathcal{R}}$ and $B^{\mathcal{R}}$.

```
@Feature(A)
class A {
  void m() {}
}

@Feature(B)
class B {
  void foo(A a) {
    a.m();
  }
}
```

(a) Before

```
@Feature(A)
class A {
  void n() {}
}

@Feature(B)
class B {
  void foo(A a) {
    a.n();
  }
}
```

(b) After Renaming m to n

Figure 4.10: Rename-Method Refactoring

The proof of (4.6) follows from (4.5) and (4.7):

$$
\begin{aligned}
\Pi_C(\ \mathcal{R}(\mathbb{P})\ ) &= \Pi_C(\ \mathcal{R}(\sum_{i\in\mathbb{F}} F_i)\ ) && \textit{// by } (4.2) \\
&= \Pi_C(\sum_{i\in\mathbb{F}} \mathcal{R}(F_i)) && \textit{// by } (4.7) \\
&= \sum_{i\in C} \mathcal{R}(F_i) && \textit{// by } (4.4) \\
&= \mathcal{R}(\sum_{i\in C} F_i) && \textit{// by } (4.7) \\
&= \mathcal{R}(P_C) && \textit{// by } (4.5) \\
&= P_C^{\mathcal{R}} && \textit{// by } (4.5)
\end{aligned}
$$

If axiom (4.7) *holds, it tells us two things:* (1) *it reaffirms that algebras for feature summation and refactoring are intuitively simple;*[6] *and* (2) *tells*

[6] The name of this algebraic structure is a 'left M-semimodule over a monoid' [69].

us how to translate refactorings of views of SPL products ($namely\ P_C$) *to refactorings of the SPL codebase* $\mathbb{P}$. That is, when an SPL programmer applies a refactoring $\mathcal{R}$ on an `X15` view of P_C, s/he sees $\mathcal{R}(P_C) = P_C^{\mathcal{R}}$ as the result. But behind the curtains, `X15` is really applying $\mathcal{R}$ to $\mathbb{P}$, and taking its C-projection to present $P_C^{\mathcal{R}}$ to the programmer.

Example: A `X15` user renames *Grafix* to *Graphics* in P_X of Figure 4.8. `X15` applies this refactoring to the entire codebase $\mathbb{P}$. The result is that *all* references to *Grafix* are renamed to *Graphics* and that the resulting projection (view) of P_X is correct as in Figure 4.8c. `X15` updates *all* programs in an SPL that are affected by this rename, and thus keeps $\mathbb{P}$ consistent.

4.4 Feature-Aware Preconditions

Applying a code transformation $\mathcal{R}$ to a codebase is well-understood [29,82], both in pretty-printing refactorings (Chapter 3) and in AST transformations. An interesting part about `X15` is how it handles refactoring preconditions. We begin by reviewing a fundamental SPL analysis, and then show how this analysis is relevant to refactoring preconditions.

4.4.1 Safe Composition

Safe Composition (SC) is a common SPL analysis. It is the verification that every program of an SPL compiles without error [5,36,77,78,101,139].

Suppose that field x is added by feature X, field y is added by feature Y,

and statement "$x = y$;" is added by feature F. This relationship is expressed by the presence condition $\psi := (F \Rightarrow X \wedge Y)$. That is, when statement "$x = y$;" appears in a product, so too must the declarations for x and y.

Let ϕ be the propositional formula of the SPL's FM [5, 9]. If $\phi \wedge \neg\psi$ is satisfiable, then at least one program in the SPL does not satisfy ψ and hence will not compile [36]. Similarly, *dead code* is source that appears in no SPL program. Let δ be the presence condition for code fragment ℓ. If $\phi \wedge \delta$ is unsatisfiable, then ℓ is dead code.

An SC tool culls $\mathbb{P}$ for all distinct ψ and δ and verifies that no program in the SPL violates either constraint. We say $\mathbb{P}$ *satisfies* SC if no presence condition ψ is violated and $\mathbb{P}$ is *dead code free* if no dead code fragments are found.

4.4.2 Preconditions for SPL Refactorings

Theorem (4.6) assumes the preconditions for $\mathcal{R}$-refactoring $\mathbb{P}$ are satisfied. But what are these preconditions? Consider the example of Figure 4.11:

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

  @Feature(X)
  void bar() {...}
}
```

Figure 4.11: Renaming foo to bar Fails

A programmer wants to refactor the base product P_{base} whose SPL codebase $\mathbb{P}$ is Figure 4.11. Method bar is invisible to the programmer as it belongs to unselected feature X. If the programmer tries to rename foo to bar, the rename fails since there is at least one product in the SPL (any configuration with X) where this rename fails, even though renaming foo to bar in P_{base} is

legal. We use the rule of Liebig, et al. [88]: *An $\mathcal{R}$-refactoring of an SPL fails if $\mathcal{R}$ fails on any product of that SPL.*

`X15` reports precondition failures of a refactoring $\mathcal{R}$ by citing a condition or SPL configuration where it fails. This is done by '*lifting*' a refactoring precondition to a SC constraint ψ and verifying all SPL products satisfy ψ. (By definition the lifted constraint implies the precondition on program P_C). `R3` supports 34 different primitive refactorings and uses 39 distinct primitive precondition checks, where each `R3` refactoring uses a subset of these 39 checks. `X15` supports all of `R3`'s primitive refactorings and preconditions.

We expected most `R3` preconditions would be feature-aware, but were surprised when only 5 of the 39 required lifting. Why?

1. Java annotations cannot be attached to *any* code fragment, such as a Java modifier. Thus, preconditions dealing with modifiers are not lifted, and thus remain identical to their unlifted `R3` counterparts. And

2. Some preconditions are feature-independent, such as Declaring Type[7] and Constructor,[8] so lifting them is unnecessary.

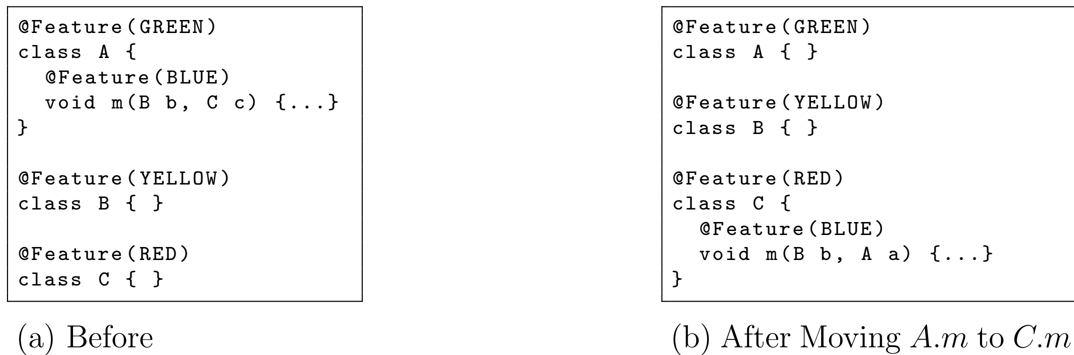
Here is a precondition that required lifting:

- **Binding Resolution.** Before a method is moved, a lifted check is performed: the moved method should still be present in all programs in

[7]A method cannot be moved if its enclosing type is an *annotation* or *interface*.

[8]A *constructor* cannot be moved.

which it appeared before the move *and* all declarations referenced in its body are still present and visible, otherwise the move refactoring is rejected. Figure 4.12 shows the before and after result of moving method $A.m$ to class C. One SC check for parameter type B prior to the move is $GREEN \wedge BLUE \Rightarrow YELLOW$[9] and after the move the check becomes $RED \wedge BLUE \Rightarrow YELLOW$.



```
@Feature(GREEN)
class A {
  @Feature(BLUE)
  void m(B b, C c) {...}
}

@Feature(YELLOW)
class B { }

@Feature(RED)
class C { }
```

(a) Before

```
@Feature(GREEN)
class A { }

@Feature(YELLOW)
class B { }

@Feature(RED)
class C {
  @Feature(BLUE)
  void m(B b, A a) {...}
}
```

(b) After Moving $A.m$ to $C.m$

Figure 4.12: Binding Resolution Constraint

The remaining four other `R3` preconditions that became 'feature aware' in `X15` are reviewed in Appendix B.

4.5 Implementation Notes on X15

The execution pipeline of `X15` appears in Figure 4.13. The only additions to the `R3` pipeline in Figure 3.11 are steps α, β, and γ.

(α) `X15` constructs feature predicates for all identifiers and stores them

[9]Also, the presence of method m implies the presence of class C.

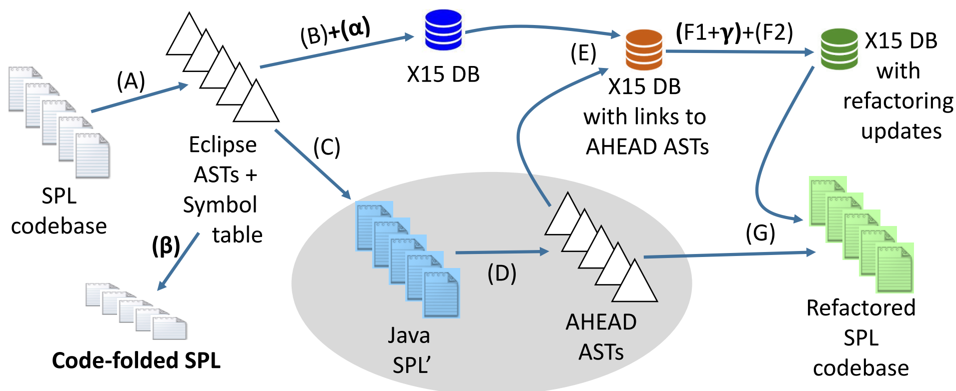


Figure 4.13: X15 Pipeline.

in the database;

(β) At a user's request, `X15` checks for dead code, validates SC, and performs code folding for viewing, editing, and debugging SPL programs;

(γ) adds feature-awareness to precondition checks.

Feature models of SPLs are rather static; they do change but slowly. `X15` culls $\mathbb{P}$ for constraints which are translated to a large number of SAT problems to solve. From experimental results in Section 4.6, a crude estimate is about 1 SAT check per every 2 lines of source. A saving grace is that the number of unique SAT checks is small, possibly orders of magnitude smaller than the crude estimate [139].

`X15` leverages the stability of an SPL's feature model by caching the results of SAT checks. When a feature-aware condition arises, `X15` identifies the unique SAT checks to verify, and looks in its SAT cache. Only when a

previously unseen SAT check is encountered will a SAT solver be invoked, and of course, its result is henceforth cached. The cache is cleared whenever the feature model is updated.

4.6 Evaluation

We use `makeVisitor` and `inverseVisitor` scripts to compare `X15`'s performance w.r.t. `R3`. `X15` has the same expressivity as `R3` – except of course in an SPL context. Like `R3`, `X15` supports 18 of the 23 design patterns in [57]; the other 5 patterns do not benefit from automation [82].

We consider three research questions:

- **RQ1:** Can `X15` refactor Java SPLs?
- **RQ2:** How fast is `X15` compared to its feature-unaware counterpart `R3`?
- **RQ3:** Does caching SAT checks improve performance?

4.6.1 Experimental Set-Up

Many public SPLs are written in the C language [77, 88, 125], but not so many for Java; there are even fewer Java SPLs with regression tests. We selected 8 Java SPLs for our studies. Three (**A**HEAD, **C**alcuator, and **E**levator) had regression tests that could validate `X15` transformations worked correctly. Two (**N**otepad and **S**odoku) lacked regression tests but could be checked by manually invoking their GUIs before and after running `X15` scripts to confirm behavior preservation. The remaining three (**L**ampiro, **M**obileMedia,

and **P**revayler) also lacked regression tests. We did not know how to execute these programs, so we could only verify that they compiled without errors before and after refactoring.

A significant challenge is translating these SPLs into `X15` codebases. SPLs that used AHEAD [13] and FeatureHouse [6], namely **M**ixin, **C**alcuator and **E**levator, were partially translated by tools – manual work was still needed. The remaining five applications (**N**otepad, **S**udoku, **L**ampiro, **M**obileMedia, and **P**revayler) used CIDE [76], which could be transformed into `javapp` automatically, and then into `X15` form. More on this in Section 4.6.3.

Like the `R3` evaluation, `makeVisitor` 'seed' methods were chosen so that they created the largest Visitors in terms of the number of 'visit' methods. As the number of refactorings needed to make a Visitor is proportional to the number of 'visit' methods, large-sized Visitors were appropriate for `X15` evaluations.

Application (LOC, #Tests, #Features)	Seed ID	# of Refs	R3 Time (seconds)						X15 Time (seconds)					Overhead	
			Bld DB	Link AST	Pre Chk	DB Upd	Proj	Tot	Pred Coll	Use SAT Caching?					
										No	Yes	No	Yes	No	Yes
			(B)	(E)	(F1)	(F2)	(G)	(R3T)	(α)	Ext Prec (γ)		Tot=(R3T)+(α)+(γ)			
	A1	54			0.000	0.026	0.191	2.016		0.092 [104]	0.035 [2]	3.074	3.017	1.058	1.001
AHEAD	**A**2	56			0.000	0.020	0.100	1.919		0.075 [56]	0.030 [2]	2.960	2.915	1.041	0.996
mixin	**A**3	58	1.712	0.087	0.000	0.030	0.460	2.289	0.966	0.110 [133]	0.040 [9]	3.365	3.295	1.076	1.006
(26K, 56, 120)	**A**4	124			0.000	0.030	0.250	2.079		0.087 [128]	0.031 [3]	3.132	3.076	1.053	0.997
	A5	552			0.020	0.221	1.601	3.641		0.170 [287]	0.040 [3]	4.777	4.647	1.136	1.006
Calculator	**C**1	4			0.000	0.006	0.046	0.235		0.036 [17]	0.030 [3]	0.316	0.310	0.081	0.075
(312, 17, 6)	**C**2	4	0.176	0.007	0.000	0.003	0.070	0.256	0.045	0.030 [4]	0.030 [4]	0.331	0.331	0.075	0.075
	C3	4			0.000	0.003	0.060	0.246		0.026 [3]	0.030 [3]	0.317	0.321	0.071	0.075
Elevator	**E**1	4			0.000	0.003	0.071	0.360		0.020 [3]	0.030 [2]	0.466	0.476	0.106	0.116
(973, 6, 6)	**E**2	4	0.263	0.023	0.000	0.003	0.067	0.356	0.086	0.026 [4]	0.030 [2]	0.468	0.472	0.112	0.116
	E3	4			0.000	0.003	0.072	0.361		0.026 [3]	0.020 [2]	0.473	0.467	0.112	0.106
Notepad	**N**1	4			0.000	0.003	0.250	0.595		0.060 [28]	0.033 [3]	0.785	0.758	0.190	0.163
(1192, 21*, 27)	**N**2	4	0.325	0.017	0.000	0.001	0.350	0.693	0.130	0.030 [2]	0.030 [2]	0.853	0.853	0.160	0.160
	N3	4			0.000	0.003	0.333	0.678		0.030 [5]	0.033 [3]	0.838	0.841	0.160	0.163
Sudoku	**S**1	4			0.000	0.004	0.055	0.383		0.035 [8]	0.030 [1]	0.574	0.569	0.191	0.186
(1975, 22*, 6)	**S**2	4	0.307	0.017	0.000	0.004	0.053	0.381	0.156	0.038 [18]	0.040 [1]	0.575	0.577	0.194	0.196
	S3	6			0.000	0.006	0.063	0.393		0.048 [47]	0.033 [2]	0.597	0.582	0.204	0.189
	L1	16			0.000	0.010	1.430	3.284		0.099 [147]	0.030 [1]	4.043	3.974	0.759	0.690
Lampiro	**L**2	26			0.001	0.020	1.002	2.867		1.164 [937]	0.050 [3]	4.691	3.577	1.824	0.710
(44K, 0, 16)	**L**3	26	1.743	0.101	0.000	0.020	1.470	3.334	0.660	0.106 [207]	0.030 [1]	4.100	4.024	0.766	0.690
	L4	32			0.003	0.019	1.033	2.899		0.874 [723]	0.050 [1]	4.433	3.609	1.534	0.710
	L5	42			0.001	0.029	1.748	3.622		1.160 [1294]	0.066 [2]	5.442	4.348	1.820	0.726
	M1	6			0.000	0.005	0.139	0.657		0.077 [79]	0.030 [2]	0.874	0.827	0.217	0.170
MobileMedia	**M**2	6			0.000	0.005	0.115	0.633		0.043 [16]	0.033 [3]	0.816	0.806	0.183	0.173
(4653, 0, 7)	**M**3	6	0.486	0.027	0.000	0.005	0.117	0.635	0.140	0.041 [14]	0.033 [3]	0.816	0.808	0.181	0.173
	M4	8			0.000	0.005	0.132	0.650		0.073 [76]	0.026 [2]	0.863	0.816	0.213	0.166
	M5	34			0.000	0.015	0.222	0.750		0.194 [432]	0.061 [18]	1.084	0.951	0.334	0.201
	P1	10			0.000	0.008	0.042	0.920		0.041 [26]	0.030 [1]	1.284	1.273	0.364	0.353
Prevayler	**P**2	10			0.000	0.007	0.044	0.921		0.039 [26]	0.030 [1]	1.283	1.274	0.362	0.353
(8009, 0, 5)	**P**3	10	0.831	0.039	0.000	0.007	0.079	0.956	0.323	0.039 [22]	0.040 [3]	1.318	1.319	0.362	0.363
	P4	16			0.000	0.007	0.076	0.953		0.036 [17]	0.026 [1]	1.312	1.302	0.359	0.349
	P5	16			0.000	0.007	0.073	0.950		0.038 [17]	0.030 [1]	1.311	1.303	0.361	0.353

– * indicates the regression tests done by invoking user interface operations manually.
– N of [N] indicates the number of SAT problems solved for extra precondition checks.

Table 4.1: `makeVisitor` Results

4.6.2 Results

4.6.2.1 Table Organization

Table 4.1 shows the results of `makeVisitor`. The first column lists the eight target programs along with their lines of code, number of regression tests, and number of features. Each row is an experiment that corresponds to `makeVisitor` invoked on the 'seed' method in the **Seed ID** column. The third column, **# of Refs**, is the total number of refactorings executed to make a Visitor for that seed.

Each of our SPLs has a 'max' configuration – all features are selected. We let `R3` execute the same refactoring script on the 'max' configuration program of each SPL to estimate the overhead of `X15` w.r.t. `R3`. The next six columns show the times spent on each `R3` pipeline step of Section 3.3:[10]

- **Bld DB** (B): time to build the `R3` database by harvesting type information from Eclipse ASTs and symbol tables.
- **Link AST** (E): time to link AHEAD AST nodes with `R3` database tuples.
- **Pre Chk** (F1): time to check feature-unaware preconditions.
- **DB Upd** (F2): time to update the `R3` database during a script execution.
- **Proj** (G): time to write the refactored code to files.

[10] The run times of `X15` scripts were measured by the *VisualVM* (ver. 1.3.8) [147]. Each experiment was executed five times to calculate our reported averages.

- **Tot** (R3T): total time in pipeline stages (B), (E), (F1), (F2), and (G).

The next three columns list the computations for feature-aware refactorings in `X15`:

- **Pred Coll** (α): time to collect presence conditions on all declarations and references.
- **Ext Prec** (γ): time spent on precondition checks, including SAT invocations to check feature-aware preconditions and the time when caching SAT solutions.
- **Tot** (X15T): total time of (R3T)+(α)+(γ) with/without caching.

By comparing the total times using `R3` and `X15`, we estimate the overhead of feature-aware refactorings in our experiments, the subject of the last column:

- **Overhead**: the overhead difference (X15T) - (R3T) in terms of execution time with/without caching.

Table 4.2 lists the results of `inverseVisitor` in an identical tabular structure.

Seed ID	# of Refs	R3 Time (seconds) Bld DB (B)	Link AST (E)	Prec Chk (F1)	DB Upd (F2)	Proj (G)	Tot (R3T)	X15 Time (seconds) Pred Coll (α)	Use SAT Caching? No: Ext Prec (γ)	Yes: Ext Prec (γ)	No: **Tot**=(R3T)+(α)+(γ)	Yes: **Tot**=(R3T)+(α)+(γ)	Overhead No	Overhead Yes
A1$_V$	54	1.714	0.091	0.000	0.010	0.180	1.995	0.912	0.142 [208]	0.026 [2]	3.049	2.933	1.054	0.938
A2$_V$	56	1.736	0.093	0.000	0.010	0.086	1.925	0.924	0.097 [112]	0.030 [2]	2.946	2.879	1.021	0.954
A3$_V$	58	1.700	0.090	0.000	0.010	0.447	2.247	0.910	0.129 [191]	0.040 [7]	3.286	3.197	1.039	0.950
A4$_V$	124	1.613	0.106	0.000	0.010	0.226	1.955	0.874	0.160 [254]	0.030 [3]	2.989	2.859	1.034	0.904
A5$_V$	552	1.478	0.101	0.010	0.010	1.490	3.089	0.693	0.350 [841]	0.058 [3]	4.132	3.840	1.043	0.751
C1$_V$	4	0.155	0.007	0.000	0.001	0.050	0.213	0.026	0.040 [19]	0.030 [3]	0.279	0.269	0.066	0.056
C2$_V$	4	0.150	0.010	0.000	0.001	0.070	0.231	0.023	0.046 [9]	0.033 [4]	0.300	0.287	0.069	0.056
C3$_V$	4	0.158	0.009	0.000	0.001	0.060	0.228	0.046	0.035 [4]	0.033 [3]	0.309	0.307	0.081	0.079
E1$_V$	4	0.255	0.013	0.000	0.001	0.060	0.330	0.073	0.030 [6]	0.026 [2]	0.433	0.429	0.103	0.099
E2$_V$	4	0.250	0.010	0.000	0.001	0.070	0.332	0.070	0.030 [6]	0.030 [2]	0.432	0.432	0.100	0.100
E3$_V$	4	0.260	0.011	0.000	0.001	0.065	0.338	0.059	0.030 [6]	0.023 [2]	0.427	0.420	0.089	0.082
N1$_V$	4	0.280	0.015	0.000	0.001	0.250	0.546	0.125	0.060 [28]	0.040 [4]	0.731	0.711	0.185	0.165
N2$_V$	4	0.263	0.020	0.000	0.001	0.020	0.304	0.126	0.030 [3]	0.036 [3]	0.460	0.466	0.156	0.162
N3$_V$	4	0.273	0.020	0.000	0.001	0.043	0.337	0.120	0.040 [6]	0.040 [3]	0.497	0.497	0.160	0.160
S1$_V$	4	0.325	0.020	0.000	0.001	0.050	0.396	0.170	0.030 [9]	0.030 [1]	0.596	0.596	0.200	0.200
S2$_V$	4	0.320	0.025	0.000	0.001	0.050	0.396	0.175	0.030 [9]	0.035 [1]	0.601	0.606	0.205	0.210
S3$_V$	6	0.340	0.020	0.000	0.001	0.070	0.431	0.173	0.050 [65]	0.036 [3]	0.654	0.640	0.223	0.209
L1$_V$	16	1.701	0.155	0.000	0.010	1.415	3.281	0.632	0.080 [90]	0.028 [1]	3.993	3.941	0.712	0.660
L2$_V$	26	1.690	0.170	0.001	0.009	1.003	2.873	0.639	0.246 [552]	0.050 [3]	3.758	3.562	0.885	0.689
L3$_V$	26	1.750	0.090	0.000	0.010	1.476	3.326	0.637	0.105 [165]	0.036 [1]	4.068	3.999	0.742	0.673
L4$_V$	32	1.765	0.106	0.002	0.008	1.055	2.936	0.630	0.301 [780]	0.050 [1]	3.867	3.616	0.931	0.680
L5$_V$	42	1.849	0.113	0.003	0.007	1.766	3.738	0.661	0.395 [1020]	0.053 [1]	4.794	4.452	1.056	0.714
M1$_V$	6	0.480	0.025	0.000	0.002	0.135	0.642	0.150	0.073 [92]	0.032 [6]	0.865	0.824	0.223	0.182
M2$_V$	6	0.470	0.026	0.000	0.001	0.110	0.607	0.165	0.050 [18]	0.030 [6]	0.822	0.802	0.215	0.195
M3$_V$	6	0.490	0.023	0.000	0.003	0.106	0.622	0.150	0.040 [18]	0.030 [6]	0.812	0.802	0.190	0.180
M4$_V$	8	0.510	0.021	0.000	0.002	0.130	0.663	0.166	0.090 [93]	0.040 [2]	0.919	0.869	0.256	0.206
M5$_V$	34	0.525	0.020	0.000	0.006	0.273	0.824	0.150	0.266 [549]	0.064 [20]	1.240	1.038	0.416	0.214
P1$_V$	10	0.745	0.026	0.000	0.002	0.030	0.803	0.370	0.040 [37]	0.025 [1]	1.213	1.198	0.410	0.395
P2$_V$	10	0.740	0.033	0.000	0.001	0.030	0.804	0.365	0.040 [37]	0.030 [1]	1.209	1.199	0.405	0.395
P3$_V$	10	0.760	0.030	0.000	0.001	0.070	0.861	0.390	0.035 [27]	0.033 [3]	1.286	1.284	0.425	0.423
P4$_V$	16	0.790	0.028	0.000	0.002	0.070	0.890	0.356	0.040 [27]	0.033 [1]	1.286	1.279	0.396	0.389
P5$_V$	16	0.765	0.030	0.000	0.001	0.065	0.861	0.366	0.040 [27]	0.030 [1]	1.267	1.257	0.406	0.396

– N of [N] indicates the number of SAT problems solved for extra precondition checks.
– X#$_V$ indicates the application where a Visitor is retrofitted into **Seed ID** X# of Table 4.1.

Table 4.2: `inverseVisitor` Results

4.6.2.2 Answers to Research Questions

RQ1: Can X15 refactor Java SPLs? X15 successfully retrofitted 64 design pattern instances on our SPLs using a total of 2,316 refactorings: 32 added a visitor pattern and 32 removed a visitor. The most challenging experiments, **A**5 in Tables 4.1 and 4.2, executed 552 primitive refactorings, respectively. Most other experiments required fewer as they have fewer 'visit' methods. Our conclusion is that X15 can indeed refactor SPL codebases.

RQ2: How fast is X15 compared to R3? To answer this question, we used three measures:

1. Consider the execution times for X15 for all `makeVisitor` and `inverseVisitor` experiments. The largest X15 experiment, Row **A**5, took 4.8 seconds. The comparable experiment using R3 took 3.6 seconds. (For a perspective on R3's improvement over the Eclipse refactoring engine, a comparable refactoring to **A**5 took Eclipse 298 seconds to execute, a speedup of over 100× [80].)

   Row **L**5 took 5.4 seconds; the comparable experiment using R3 took 3.6 seconds. The numbers for `inverseVisitor` in Table 4.2 are similar. For less demanding scripts – remember: *rows are not individual refactorings* – all X15 executions complete in under 1.4 seconds; the corresponding R3 executions finish in under 1 second. On average across all experiments, X15 was 1/2 seconds slower than R3 per experiment.

2. Database creation time is small for `R3`; the largest experiments (**A** and **L** rows) take less than 2 seconds. `X15` additionally collects feature presence predicates column (see column α of Table 4.1); this adds one more second of execution time for the largest SPLs. For smaller SPLs, `X15` and `R3` database build times are indistinguishable. For a perspective, between the time a user clicks the Eclipse GUI and the list of available scripts is displayed, both `R3` and `X15` harvesting can be done with time to spare.

3. Over 80% of Eclipse refactoring execution time is consumed by checking preconditions [82]. In contrast, `R3` precondition checking is almost instantaneous (see column (F1)). `X15` takes advantage of `R3`'s speed, but spends extra time for feature-aware precondition checks. They do indeed incur additional overhead (see column (γ)). In the largest SPLs, this adds another 1.2 seconds without theorem caching. As before, for smaller SPLs, the additional time is unnoticeable.

Our conclusion is that `X15` refactors SPLs at comparable speeds to `R3`, a feature-unaware refactoring engine.

RQ3: Does caching SAT checks improve performance? To answer this question, we used two measures:

1. The average overhead for checking feature-aware preconditions in the `makeVisitor` experiment was 0.5 seconds without caching SAT solutions. With caching, the average overhead dropped to 0.4 seconds. For

App	No-caching (seconds)		Caching (seconds)		Speed Up	
	Dead Code	**Safe Composition**	**Dead Code**	**Safe Composition**	**Dead Code**	**Safe Composition**
A	1.179 [182]	94.675 [19,811]	1.130 [176]	4.210 [62]	1.04 (6)	22.46 (19,749)
C	0.110 [42]	0.140 [108]	0.100 [39]	0.080 [9]	1.10 (3)	1.75 (99)
E	0.230 [158]	0.256 [676]	0.229 [155]	0.130 [16]	1.00 (3)	1.97 (660)
N	0.380 [188]	0.497 [635]	0.470 [188]	0.240 [86]	0.81 (0)	2.07 (549)
S	0.285 [79]	0.426 [854]	0.300 [64]	0.250 [14]	0.95 (15)	1.70 (840)
L	0.780 [138]	6.741 [29,501]	0.680 [62]	1.260 [11]	1.15 (76)	5.35 (29,490)
M	0.362 [125]	0.869 [1,976]	0.270 [87]	0.220 [25]	1.34 (38)	3.95 (1,951)
P	0.445 [94]	1.244 [3,329]	0.470 [88]	0.470 [12]	0.95 (6)	2.65 (3,317)

– N of [N] is the number of SAT problems solved for extra precondition checks.
– N of (N) is the number of SAT problems whose solution was found in the cache.

Table 4.3: Dead Code and Safe Composition Check Results

a perspective, experiment **L**5 spent 1.2 seconds proving 1,294 theorems, a vast majority of which were duplicates. With caching, only one extra theorem required a SAT proof, taking 0.1 seconds.

2. Table 4.3 shows the time and number of SAT problems for dead code and SC checks on the SPLs in Table 4.1. $\mathbb{P}$ satisfying SC and being dead code free is a precondition for `X15` refactorings. Again, we took two different approaches (non-caching and caching) to measure how much time `X15` can save by reusing SAT solutions. On average for our experiments, caching increased the speed of dead code checks by 1.03× and SC by 15×.

Table 4.2 shows results of our `inverseVisitor` experiment. On average, the overhead for feature-awareness in `inverseVisitor` refactorings was 0.5 seconds without caching and 0.4 seconds with caching, which is miniscule. The results of `inverseVisitor` are no different than those of makeVisitor.

Readers may be surprised at the response time of our SAT invocations. This is due to the fact that the feature models of our SPLs are relatively simple. Having said this, our observations are consistent with prior work that SAT problems for feature models are 'easy' [93].

Our conclusion is that caching solutions to SAT checks does indeed improve performance.

4.6.3 Threats to Validity

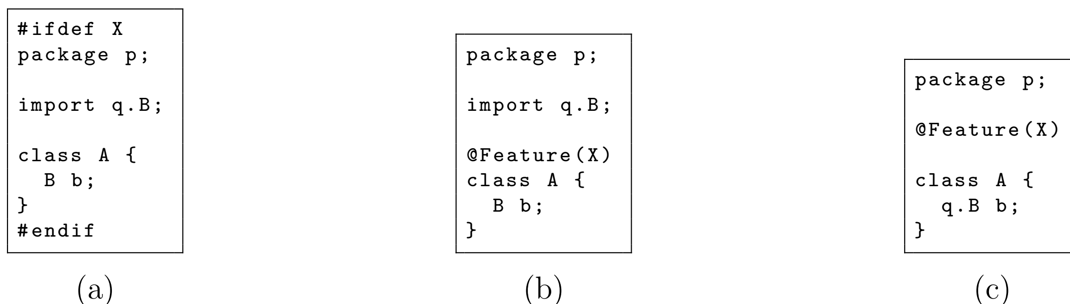
Every SPL tool today uses a unique means to encode variability.[11] In order to use these SPLs in our experiments, we had to modify them to use `X15` annotations.

SPLs that used AHEAD [13] and FeatureHouse [6], namely **M**ixin, **C**alcuator and **E**levator, were partially translated by tools – manual work was still needed. The remaining five applications (**N**otepad, **S**udoku, **L**ampiro, **M**obileMedia, and **P**revayler) used CIDE [76], which could be transformed into `javapp` automatically, and then into `X15` form.

In Section 4.6.1, we said that the four applications in Table 4.1 used `javapp` to specify features [72]. In order to use them, we had to reformat `javapp` to Java custom annotations by hand. We did our best to keep the original feature specification but there were some code fragments that required special care. Example: Figure 4.14a shows a compilation unit belonging to

[11]Even variability-aware compilers require source adjustments to be used [78,88]; there is no free lunch to use any existing SPL tool.

optional feature X using `javapp`. As *import*s cannot be annotated in Java, we assigned feature X to the class declaration A in Figure 4.14b. However, in case class B belongs to X which is unselected, Figure 4.14b violates SC: it is an error in Java to *import* a non-existent class. Our solution was to use the fully qualified name instead as shown in Figure 4.14c.[12]



```
#ifdef X
package p;

import q.B;

class A {
  B b;
}
#endif
```

(a)

```
package p;

import q.B;

@Feature(X)
class A {
  B b;
}
```

(b)

```
package p;

@Feature(X)

class A {
  q.B b;
}
```

(c)

Figure 4.14: Translation `javapp` to `@Feature` Annotations

Note: There are no "standard" tools, perhaps other than CPP, for SPL construction. All new SPL prototypes, including `X15`, require translations (often manual) of existing Java codebases to a form that the prototype can be used. There is no "free lunch" for any tool.

4.7 Related Work

4.7.1 A Survey of SPL Tools

Future tools for Java SPLs should have the following properties:

1. Support the refactoring of SPL codebase $\mathbb{P}$,
2. Do not create a separate code base for P_C,

[12]Even variability-aware compilers require source adjustments to be used; there is no free lunch.

3. Propagate text edits from P_C back to $\mathbb{P}$,
4. Propagate refactorings of P_C back to $\mathbb{P}$.

because refactorings are central to Java program development; and manual propagation of changes is laborious and error prone [13, 25].

If SPL tools created a separate codebase for P_C, it is possible to automatically propagate edits in P_C to $\mathbb{P}$. But *not* the edits made by refactorings. Why? Recall the Rule of Liebig et al [88]: *An $\mathcal{R}$-refactoring of an SPL fails if $\mathcal{R}$ fails on any product of that SPL.* Refactoring P_C as an isolated codebase will not account for modifications of other products of the SPL where that refactoring's precondition fails. Thus, unless a separate codebase for P_C also keeps track of all other products in $\mathbb{P}$, back-propagating of edits will fail.

Table 4.4 categorizes the properties of `X15` with eight well-known SPL tools (AHEAD [13], CIDE [76], Choice Calculus tools [131, 148], DeltaJ [84], DOPLER [44], Gears [25], FeatureHouse [6], and pure::variants [109][13]). `X15` is unique among existing SPL tools in that it supports all key properties.

4.7.2 Variation Control Systems

Variation Control Systems (VarCSs) are tools that project a reconfigurable codebase $\mathbb{P}$ to produce a separate codebase called a 'view'. The view is edited and its changes are back-propagated to $\mathbb{P}$ by an update tool. AHEAD and Gears are VarCSs among many others [148].

[13] pure::variants has a tool that updates SPL products when $\mathbb{P}$ is changed [30]; this is forward-propagation $\mathbb{P} \rightarrow P_C$ of changes, not back-propagation $P_C \rightarrow \mathbb{P}$ of `X15`.

Tool	Supports OO of refactorings Java SPL codebases	Tools to project editable views	Does NOT create a separate codebase for debugging and execution	Back-propagates edits	Feature-aware back-propagation
AHEAD	✗	✗	✗	✓	✗
Choice Calculus	✗	✓	✗	✓	✓
CIDE	✗	✗	✗	✗	✗
DeltaJ	✗	✗	✗	✗	✗
DOPLER	✗	✗	✗	✗	✗
Gears	✗	✗	✗	✓	✗
FeatureHouse	✗	✗	✗	✗	✗
Pure::Variants	✗	✗	✗	✗	✗
X15	✓	✓	✓	✓	✓

Table 4.4: Comparing Capabilities of SPL Tooling

The most advanced VarCS tools [131, 148] are based on the Choice Calculus [52] and rely on the *edit isolation principle* (EIP), which says that all edits made to a view are guaranteed not to effect code that was hidden by projection. `X15` follows the EIP as long as refactorings are not performed; refactorings violate EIP. We showed that propagation tools for text edits are inadequate to deal with the changes refactorings make to SPL products. Never-the-less, empirical results by Stanciulescu et al. show VarCS tools are feasible to edit and maintain real-world SPLs [131]. VarCS ideas can offer additional improvements to `X15`.

4.7.3 Other Java Variabilities

The Choice Calculus [52] is a formal model of variability-aware (not just feature-aware) languages. A capability that the ChoiceCalculus has (and we might add so does CIDE [76]) that `X15` does not is the removal of method parameters that are *Feature*-annotated [52, 76, 88]. Consider the Java code of

Figure 4.15a. Parameter a is $Feature$-annotated, suggesting that it is removed if X is not a feature of the target configuration. Figure 4.15b shows the projected result when $\neg X$ holds. There are SPL tools that support such variability [52, 76, 88].

```
(a) void m( @Feature(X) A a ) {...}

(b) void m( ) {...}
```

Figure 4.15: Parameter Removal by Projection

`X15` presently ignores *Feature* annotations on parameters of methods and generics. We are unconvinced that parameter projection is a good idea as it encourages unscalable SPL designs: if method m has 2 parameters in some SPL programs, 3 in others, and 4 in the remainder, it quickly becomes confusing to know which version to use and when. If there are many such methods, to keep all of these variations straight, the SPL codebase becomes impossible to understand. There is no technical reason that precludes parameter projection in `X15` other than increased complexity; we leave its necessity for others to decide and add.

Java annotations have room for improvement. Cazzola et al. [31] presented @Java, an extension to Java language, that can annotate finer-grained code fragments such as blocks and expressions that cannot be annotated by Java. The `atjava` tool translates @Java annotations to Java-compilable code and then inserts custom attributes into bytecode instead of the translated code. @Java could improve `X15` when `atjavac` (i) provides the start and end

of each annotated code fragment, (ii) preserves the original @Java annotation's value expression (i.e., feature expression in `X15`), and (iii) keeps the annotated expression if it exists. `atjavac` currently supports (i) and (iii), and can be customized to do (ii).

4.7.4 Variability-Aware Compilers

Conditional compilation in Java has taken two forms: One is OO language-extensions to support type safe variability, such as [8, 51, 70]. These latter papers are elegant proposals to extend OO languages with conditionals to enable static variability and type safety using generics.

The other uses preprocessors, such as [72,108,126], which leads to work on VACs [22, 52, 77, 88, 149]. Developing tools to parse C-with-CPP source to analyze the impact of feature variability is difficult [33,62,77], but unavoidable if CPP-infused SPL codebases are to be analyzed. It may be years, if ever, before a VAC for C++ appears. Most of the effort in developing VACs deals with the artificial complexity that CPP constructs add to host languages [59, 61]. And using these VACs is *not* without effort – the codebase must use disciplined annotations [89].

In contrast to the above research, `X15` requires *no changes* to Java or its compiler. `X15` directly supports feature-variability for view editing, view compilation, and view refactoring, capabilities that existing SPL tools lack.

4.7.5 Refactoring Variability-Aware Codebases

Schulze et al. [120] report experiences on integrating FeatureHouse [6] with refactorings, such as pull-up, but also refactorings that partition large features into a composition of smaller features. The authors report the difficulties on refactoring SPLs when physical feature modularity is used. A deliberate design decision of ours was to use an annotative (or implicit feature modularity) approach to avoid these implementation difficulties. `X15` relies on pure Java, not a custom extension of Java. We argued that the mathematics of (4.6) applies to all feature-based SPL implementations — including those that rely on special languages to support feature modularity. But to do so requires building a custom compiler and a custom refactoring engine, which is daunting.

There are other useful kinds of feature 'refactoring'. Schulze et al. [121] presented module refactorings such as rename, merge, and remove for DOP SPLs. Code smells were proposed to identify refactoring opportunities in DOP [116]. These are potential future extensions of `X15`.

Kuhlemann et al. [85] proposed *Refactoring Feature Modules* (RFMs). Just as we use the term feature modules to mean building-blocks of SPL products, an RFM is a feature module *or* a single product refactoring (not a refactoring script). An RFM refactoring is feature-unaware and is applied to a feature-unaware product to adapt it for use in a legacy application. Although RFMs have a name that is suggestive of our work, it does not deal with feature-aware refactorings. Nevertheless, subsequent refactoring an SPL program for

adaption is a good idea because it separates the concerns for SPL product development and creation from later adaptation.

Aspect-aware refactorings [2, 66, 98, 151] are a counterpart to feature-aware refactorings. The technical issues and solutions explored were specific to AspectJ (e.g., pointcuts and wild-cards), and are distant topics to OO refactoring feature-based Java SPLs.

Chapter 5

Conclusions and Future Work

OO refactoring technology is now more than 25 years old. Most researchers, ourselves included, tacitly assumed that few significant advances in tooling classical Java refactorings were possible after this time. But looking closer, motivated by practical needs and applications for refactoring, reveals that significant advances are not only possible but are necessary. This thesis identified three core problems in software refactoring tools:

- There are many program transformation tools with distinguished merit, but none (to us) seems practical in the long run for refactoring scripts. Writing customized refactorings is a task that is largely reserved to tool experts due to overly complex programming interfaces and paradigms. We presented a refactoring tool for undergraduate students to write high-order refactorings (e.g., design patterns) as parameterized refactoring scripts in Java.
- Most research on code transformation mentions that many infrastructures or tools could be used to implement refactoring or transformation scripts. Notably none reported performance of refactoring engines. Our

experiments show that refactoring performance is critical for future refactoring engines. We built a radically different refactoring engine based on AST rendering to tackle issues of performance, expressivity, and scalability for large programs. Our engine runs 10$\times$ faster than the Eclipse refactoring engine.

- Existing refactoring tools for OO languages are unaware of software variability. We presented an approach to refactor Java SPL products and to back-propagate the changes to the SPL codebase. Our tool is based on a custom variability mechanism using innate Java annotations and IDE code folding/commenting.

Our tools `R2`, `R3`, and `X15` have addressed these problems.

5.1 R2: Practical Scripting of Refactorings

Retrofitting design patterns into a program using refactorings is tedious and error-prone. The burden can be alleviated, either partially or fully, by refactoring scripts. Today's IDEs offer poor or no support for scripts, or require a background and understanding of IDE internals that students and most programmers will never have. Proposed DSLs that can be used for scripting require knowledge of yet another programming language or the need to code primitive refactorings.

Our solution `R2` uses (1) Java as a scripting language, (2) `R2` objects are class, method, and field declarations of a Java program, and (3) `R2` methods

are native JDT refactorings, primitive transformations, or our scripts. We used `R2` to automate 18 out of 23 classical design patterns, where each `R2` script is a compact Java method. The `R2` idea is also portable to other Java IDEs such as IntelliJ IDEA, NetBeans, and Visual Studio; it is not limited to Eclipse (or to Java, for that matter). Our user study shows that `R2` (and `R3`) refactoring scripts (1) improve the success rate of retrofitting design patterns by up to 3.5×, (2) are reusable on non-trivial programs, and (3) safer than a manual process for even relatively small programs.

Practical issues still remain. We found that refactoring scripts place a heavy demand on the correctness, expressiveness, and speed of IDE-provided refactorings:

- Correctness of IDE-supplied refactorings remains a serious problem. One of our experiments executed 96 JDT refactorings and introduced 100 errors (in Juno 4.2.2) that we had to fix manually. It took two years for a version of JDT (Luna 4.4.1) to resolve these bugs.
- IDE-supplied refactorings should be expressive and easy to understand. Odd or limited refactorings (as discussed in Section 2.2) preclude or otherwise distort elegant scripts. An expressive basis set of primitive refactorings to be supported by IDEs remains an open problem. Our work takes this first step to eliminate the rough edges of JDT refactorings by generalizing them enough to be useful in scripting design patterns.
- Refactoring speed is important as programmers expect instantaneous

results. In our experiments, many executions are over 20 seconds; the largest is 10 minutes.

5.2 R3: 10× Speed Improvement

We built a new refactoring engine that executes R2 scripts almost instantaneously. Also, we showed how *classical* Java refactorings (e.g., move, rename, change-method-signature) and refactorings that are essential to script the creation and removal of Gang-of-Four design patterns, can be implemented by a novel combination of databases and AST pretty-printing. Our tool R3:

- improves correctness by (1) applying more precise preconditions, (2) passing all available JDTRE regression tests, and (3) fixing the bugs that we discovered so far,
- like R2, expresses eighteen Gang-of-Four design patterns which can be fully or partially automated as refactoring scripts, and
- executes refactoring scripts 10× faster than JDTRE on average.

Whether off-the-shelf JDT (or other IDE refactoring engines) will ever match these capabilities remains to be seen. Standard OO refactoring engines leave a *lot* to be desired – slow speed, no support for scripting, and overly complex code bases.

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 [15, 54] and there are many refactorings that cannot simply be "pretty-printed", such as refactoring sequential legacy code into parallel code [41]. 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.

5.3 X15: Refactoring Java Software Product Lines

`X15` is a tool that not only brings critical refactoring support to Java SPLs, it also solves four other vexing problems:

- propagation of edits and refactorings of SPL programs back to the SPL codebase,
- scripting refactorings to automatically retrofit SPL codebases with design patterns,
- *not* requiring language extensions to Java or a special variability-aware compiler; a standard Java compiler will suffice, and
- efficiency: `X15` is only 1.8 seconds slower than `R3` for the largest SPL application in our experiments.

`X15` leverages practical experiences in years of Java SPL construction to eliminate artificial complexities and ambiguities in SPL design. It also leverages a theorem for refactoring feature-based SPLs that reveals a key distributivity property (refactorings distribute over feature module compositions)

that makes `X15` concepts and implementation clean. We believe that `X15` significantly advances and simplifies the state-of-the-art in SPL tooling.

5.4 Lessons Learned

We spent two years on `R2`. Our initial goal was to create a refactoring engine that undergraduates could understand and use. It is well-known that PTSs are difficult to use, and are typically used only by their authors. For example, Semantic Design's DMS (Design Maintenance System) [22] is an impressive, industry-hardened tool for large-scale program transformations. DMS has its own proprietary programming language called *Parlanse*. The learning curve for undergraduates to become proficient in Parlance (or more generally another language whose programming paradigm is different from Java) makes DMS (and lesser tools) unappealing to the masses.

In general, we found PTSs are intimidating and are not for casual users. To us, Eclipse is no different. Although the Eclipse is typical of the state-of-the-art refactoring tools, it is not a system that can be easily given to undergraduates to pick up, use, and modify. JDT refactorings use the LTK framework to provide language-neutral refactoring APIs. LTK consists of a refactoring core, UI components, and incorporates the JDT's UI and language-specific support. We were not aiming our effort at people with hard-core interests writing arbitrary program transformations; the LTK framework and DMS engine are for them.

During our two years, we committed ourselves to use JDTRE, at least

initially, to better understand the problems of contemporary refactoring engines. Consequently, we were convinced that further development of `R2` within Eclipse, using the JDT and LTK, was not the way to go. At the same time, we extensively studied the JDTRE source to learn how a classical refactoring engine is implemented, for example, by separating precondition and postcondition checks, pre-computing code generation through AST operations, and supporting other functionalities such as preview. The knowledge and experience we gained from JDTRE analyses were essential to our future work; they helped us design and build a more reliable, expressive, and faster refactoring engine, which is `R3`.

We also recognized that designing a scripting interface is not easy at all. One problem is that we want to encode constraints into a more abstract, understandable and reusable form for students as opposed to, say, coding them in Java as is currently done in the JDTRE. (Needless to say, the analyses needed to evaluate preconditions for each refactoring are the most difficult tasks to implement a refactoring engine.) Also, refactoring APIs should be both user-friendly and functional almost as much as GUI. Please note that there is considerable evidence that refactoring tools are significantly underused. Different studies have offered reasons, often pointing to high complexity of user interfaces for refactoring tools [100,143]. Despite a large amount of time and effort, we still do not know how to efficiently locate a field variable in an anonymous class declaration by writing a script. Nevertheless, we believe that `R2` APIs are well-designed to allow students to write programs that sequence

refactoring steps to mechanize orchestrated program changes, such as design patterns. Our empirical studies proved that undergraduates can script their own refactorings at a significantly higher success rate, comparing with using JDT refactorings. Manual interactions with IDE refactorings require not only to determine a precise sequence of refactorings to invoke but to make additional decisions (e.g., when refactorings violate preconditions, but refactorings can repair these exceptions), all of which is hidden by `R2`.

`R3` (and `X15` which is based on `R3`) renders ASTs to produce refactored code, which is a radically different approach of implementing a refactoring engine. To do so, `R3` builds a distinct database for pretty-printers to reference when rendering ASTs. For the sake of the database (which is further optimized for our purpose), checking preconditions became much simpler and faster in `R3`. Also, the removal of AST operations contributed to the speed-up of `R3` refactorings. However, as `R3` supports scripting "a series of" refactorings and performs projection "only once" after executing the entire scripts, the implementation complexity of (1) recording the changes made by a number of refactorings into the database and (2) pretty-printing refactored code at a single phase by calculating all changes in the database was higher that we expected. An example is explained in Appendix D.

5.5 Future Work

There remain interesting research problems for future work. First, we need more fine-grained refactorings to achieve the goal of greater automation

in software design. Fowler introduces over 90 refactorings in his webpage; only one third of them is currently supported by IDEs including `R2` and `R3`. Not only are additional primitive refactorings needed, but also technical definitions in terms of precondition definitions and code transformation rules. Contemporary texts on refactorings (and design patterns) lack precision in their descriptions, offering vague explanations instead.

Second, `R3` does not take advantage of the latest technologies in software parallelism and/or multi-core architecture. `R3` builds the entire database at one time. We wonder how much faster `R3` refactorings could be when building database tables in parallel and locally updating only database tables affected by code changes. We expect to have a combination of parallelism and database localization in the next generation of `R3`. `R3` recently improved its projection speed 10-fold simply by using `StringBuilder` that outperforms `String` in string concatenation.

Third, the real issue of the state-of-the-art refactoring engines is unreliability. We revealed that the current refactoring tools are infested with bugs, which are rarely detected by regression tests. Even worse is that none of them are aware of flaws in code transformation that we reported recently [83].

Fourth, we plan a user study with `X15` to measure productivity when we refactor SPLs using `X15` comparing with manually edits to back-propagate changes to SPLs. We see improving refactoring engines as a significant, interesting and intellectual challenge, because the behavior preservation property of refactoring engines is within reach provided that there are pioneers to make

it so.

Fifth, we believe that `R3` and `X15` provide a basis for a fundamental advance in teaching classical refactorings to undergraduate and graduate students. No more will it be necessary to offer vague descriptions of refactorings and patterns; now it will be possible to have students write their own refactorings, and compose them to write their own design pattern variants. Our experience in teaching has clearly revealed that the lack of precision and the lack of a clear understanding of what refactorings and patterns do are primary sources of confusion and errors. By presenting tools, such as `R3` and `X15`, we believe a new dimension of understanding can be reaped by students and practitioners alike, leading to more use of refactoring tools and better tools in the future.

Appendices

Appendix A

Visitor Variants

Given a Visitor-retrofitted program in Figure A.1a, a programmer may break the pattern structure inadvertently by introducing method $B.accept$ in Figure A.1b that shares run-time polymorphism with *accept* methods but lacks invoking the corresponding *visit* method. When another programmer makes a common change to *visit* methods in the $Visitor$ class, the change will not be executed when dangling $B.accept(Visitor)$ is invoked. In order to avoid such mistakes in the follow-up maintenance, refactoring engines would need to introduce *accept* and *visit* stubs that preserve the Visitor structure. Also, *super*-delegates are required to relay calls to super *accept* methods.

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

class B extends A {



}

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

  void super_mθ() {
    super.accept(Visitor.instance);
  }
}

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

  void visit(A a) {}

  void visit(C c) {
    c.super_mθ();
  }
}
```

(a) Original Visitor

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

class B extends A {
  void accept(Visitor v) {
    //missing a call to visit
  }
}

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

  void super_mθ() {
    super.accept(Visitor.instance);
  }
}

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

  void visit(A a) {}

  void visit(C c) {
    c.super_mθ();
  }
}
```

(b) Broken Visitor Structure

Figure A.1: An example of Visitor Pattern

Appendix B

Feature-Aware Preconditions for Design Pattern Refactorings

Execution Flow. Figure B.1a shows a feature-unaware class A. It is illegal to inline method n due to the *return* statement inside n, as the i++ statement of method m would never be executed. In contrast, inlining is allowed in class A of Figure B.1b, provided that feature $BLUE$ implies *not* RED. Although this example may seem artificial, we did need this check for the `inverseVisitor` script of Section 4.6.

```
class A {
  int i = 0;

  void m() {
    n();
    i++;
  }

  void n() {
    i = 1;
    return;
  }
}
```

(a) Without Features

```
class A {
  int i = 0;

  void m() {
    if(BLUE) n();
    if(RED) i++;
  }

  void n() {
    i = 1;
    return;
  }
}
```

(b) With Features

Figure B.1: Inlining Constraint

Variable Capture. Renaming field $B.j$ to $B.i$ in Figure B.2a intercepts the binding to inherited variable $A.i$. In Figure B.2b, capture does not

arise if features $BLUE$ and RED are mutually exclusive [88].

```
class A {

  int i = 0;
}

class B extends A {

  int j = 1;


  void m() {
    i++;
  }
}
```

(a) Without Features

```
class A {
  @Feature(BLUE)
  int i = 0;
}

class B extends A {
  @Feature(RED)
  int j = 1;

  @Feature(BLUE)
  void m() {
    i++;
  }
}
```

(b) With Features

Figure B.2: Variable Capturing Constraint

Explicit Super Invocation. Default constructors are needed in class inheritance hierarchies. Suppose that feature $BLUE$ is unselected in Figure B.3b. Java generates an error for the code as class A has no default constructor.

```
class A {
  A(int i) {}

  A() {}
}

class B extends A {
}
```

(a) Without Features

```
class A {
  A(int i) {}
  @Feature(BLUE)
  A() {}
}

class B extends A {
}
```

(b) With Features

Figure B.3: Non-default Constructor Constraint

Existence of Type Creation. A singleton design pattern refactoring introduces a single static instance of a class A, and replaces the only constructor call to A in a program with a reference to this instance. The program in

Figure B.4b satisfies the singleton constraint provided that features $BLUE$ and RED are mutually exclusive.

```
class A {

  A a = new A();


  public static final
    A instance = new A();

  A(){
    /*do something*/
  }
}
```

(a) Without Features

```
class A {
  @Feature(BLUE)
  A a = new A();

  @Feature(RED)
  public static final
    A instance = new A();

  A(){
    /*do something*/
  }
}
```

(b) With Features

Figure B.4: Singleton Constraint

Appendix C

X15 Design Rules

It is well-known CPP encourages bad SPL coding practices and significantly complicate tooling [22, 52, 76, 77, 88, 149]. Modern OO languages, like Java, renounce preprocessors (although unofficial preprocessors exist [72, 108, 126]). SPL tooling for modern languages should follow modern design practices, and to this end, we propose rules to eliminate unnecessary design complexity. We consider violations of these rules 'bad smells' in SPL design.

Our first rule is a common-sense naming convention: *All programs of an SPL use the same name for the same declaration.* Here is why: Let d be a declaration that appears in many programs of an SPL. Suppose d is given the name "*dd*" in some SPL programs and "*d*" in others. This is *name variability*: it *doubles* the information a programmer must remember. S/he has to know when to use "*d*" and when to use "*dd*". A decent-sized SPL can have thousands or tens of thousands of declarations. To remember all type, method, and variable names is hard enough, but complicating this knowledge with name variability is untenable. If there are k declarations and each declaration has two names (with conditions on when to use each), name variability magnifies design complexity exponentially ($O(2^k)$). *Name variability is unnecessary and*

harmful to SPL designs.

Our second rule is similar: We expect the semantics of declarations to vary slightly among SPL programs, but every declaration should have a consistent meaning. We do *not* want a declaration to mean one thing in some programs and something radically different (e.g., have a fundamentally different type) in others. This is *semantic inconsistency*: For method m to mean one thing in some programs, and something vastly different in others, doubles the amount of information a programmer needs to remember. For the same motivations of name variability, *semantic inconsistency is unnecessary and harmful to SPL designs.*

Our third rule is that the codebase $\mathbb{P}$ of an SPL must compile if any analysis is to be done. Simply: one cannot check SC or refactoring preconditions if a codebase has type errors; doing so compromises the validity of semantic analyses [37, 75]. A type-error-free compilation of $\mathbb{P}$ need *not* correspond to a valid SPL program – it is merely a check that *every* reference is bound to *some* declaration in $\mathbb{P}$ [139]. If this is not so, at least one program of the SPL does not compile. Type-error-free compilation of $\mathbb{P}$ is a precondition for SC.

Our fourth rule is that all products of SPL satisfy SC.

Henceforth, we follow four *Design Rules* for writing, analyzing, and refactoring a Java SPL codebase:

DR1. Absence of name variability,

DR2. Absence of semantic inconsistencies, and

DR3. Compilability of the Java SPL codebase $\mathbb{P}$.

DR4. Compilability of all SPL products.

These rules have practical value. Suppose product P_C requires particular variables and methods to have specific names so that P_C integrates neatly with some legacy application. Instead of complicating an SPL codebase $\mathbb{P}$ and violating **DR1**, separate concerns by maintaining a simple codebase $\mathbb{P}$ that does not violate **DR1**, and use post-processing to refactor P_C for legacy conformance. Doing so no longer precludes using P_C without these adaptations or the need to adapt P_C to other legacy applications [85].

Just because bad SPL practices are used today (e.g., clone-and-own) does not mean that their practice should be perpetuated. The four design rules improve tooling and can be applied to non-`X15` SPLs as well:

DR1: Absence of name variability. Java with $Feature$-annotations makes it hard to give variables different names in different configurations. It is not impossible, but requires egregious hacks. Name variability can be easier to accomplish in CPP, but this requires variability-aware compilers for Java [78], which to our knowledge do not yet exist. To violate **DR1** easily will require unlikely additions to Java, complicating its compiler and supporting tools.

DR2: Absence of semantic inconsistency. Java with $Feature$-annotations makes it hard for a variable to be an instance of one Java type in some configurations and a different Java type in others. Figure C.1a shows a

common CPP idiom that violates **DR2**: field *global* has type *int* when feature X is defined, otherwise it is a *bool*. It is easy to do with CPP, but would require egregious hacks in `X15`. Our solution is either to use different variable names or give *global* a single type. Figure C.1b shows another common CPP idiom for initializing a variable; we express same idea in a slightly more verbose way in Figure C.1c. This variability does not come free in C-with-CPP: tools are more complicated. We cannot imagine that a future version of Java will allow for such variability – doing so will significantly complicate the Java compiler and its supporting tools.[1] We hope/believe that future SPLs will be developed with modern OO languages that reject such artificial complexities.

```
#ifdef X
  int global;
#else
  bool global;
#endif
```

(a)

```
#ifdef X
  int global = 1
#else
  int global = 0;
#endif
```

(b)

```
int global;
{
  if(X)
    global = 1;
  else
    global = 0;
}
```

(c)

Figure C.1: C-Preprocessor vs Java SPL Idioms

The remaining rules, **DR3: Compilability of the Java SPL codebase** $\mathbb{P}$ and **DR4: Compilability of all SPL products** guarantee that the validity of semantic analyses is not compromised. Admittedly ignoring semantic correctness can indeed simplify tooling, as early SPL tools testify [13].

[1]Semantic inconsistencies arise in bad OO designs. Example: superclass *Game* has a method *draw*() to mean make next move and subclass *USFootball* has method *draw*() to mean run a draw play, violating subtype polymorphism [107]. We are unaware of tools that can detect such errors; eliminating semantic inconsistencies of this sort are the responsibilities of SPL designers.

However, semantic correctness is something that future SPL users should refuse to surrender.

Appendix D

A Challenge of R3 Implementation

When a new refactoring is added to R3, we need to precisely define an additional rule for both database update and projection to render the related program elements correctly without side effects on existing refactorings, which is challenging. For example, move-instance-method refactoring changes the owner class of a method. Literally, it is implemented as an owner pointer update of the target method in R3 database. When R3 supported additional pull-up/push-down refactorings which are also changes of owner pointers, we had to define two code transformation rules for a "moved" method to render differently. Further, a moved method renders its AST in a single destination type only whereas push-down can replicate a method in multiple subclasses as shown in Figure D.1.

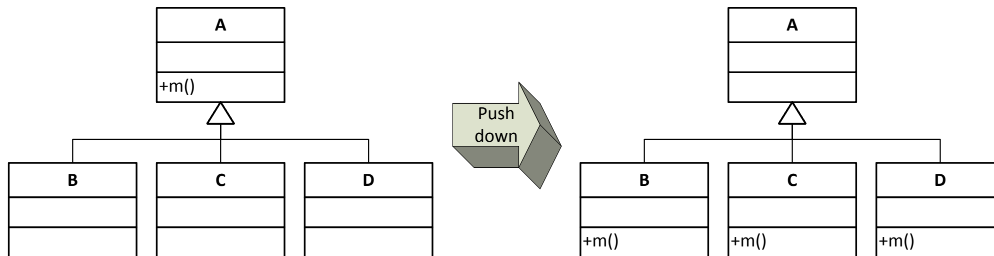


Figure D.1: Push Down m() to Multiple Subclasses.

The move-instance-method and pull-up/push-down refactorings of R3

are differentiated by (1) the use of a via-parameter and (2) whether the destination type is either a parent or a child class (for pull-up or push-down). Since move-via-field incorporates adding a parameter to be used as a via-parameter internally (see Section 3.2.3.4), move-instance-method always requires a valid destination parameter for both move-via-parameter and move-via-field refactorings while pull-up/push-down does not. Also, when the current owner of a method is its parent class, `R3` applies a special rule for *super* keyword transformation that replaces *super* with *this* if the *super* prefix is used to reference a member of the parent (not ancestor) class as shown in Figure D.2a-b.

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

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

(a) Before Pull-Up
(d) After Push-Down

```
class A {
  void n() {}
  void m() {
    this.n();
  }
}

class B extends A {
  void n() {}
}
```

(b) After Pull-Up
(c) Before Push-Down

Figure D.2: Pull-Up/Push-Down m() Refactoring

A symmetrical *this* transformation (replacing *this* with *super*) for push-down is applied when a target method uses *this* to reference a current member method/field which is overridden/hidden in the destination subclass as shown in Figure D.2c-d.

Bibliography

[1] Advanced Software Engineering (CS561) Course at the Oregon State University. `classes.engr.oregonstate.edu/eecs/fall2014/cs561/`.

[2] Prasanth Anbalagan and Tao Xie. Automated Inference of Pointcuts in Aspect-Oriented Refactoring. In *ICSE*, 2007.

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

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

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[6] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[7] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In *RTA*, 2007.

[8] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized Types for Java. In *POPL*, 1997.

[9] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, September 2005.

[10] Don Batory, Peter Höfner, and Jongwook Kim. Feature Interactions, Products, and Composition. In *GPCE*, 2011.

[11] Don Batory, Peter Höfner, Bernhard Möller, and Andreas Zelend. Features, Modularity, and Variation Points. In *FOSD*, 2013.

[12] Don Batory, Eric Latimer, and Maider Azanza. Teaching Model Driven Engineering from a Relational Database Perspective. In *MODELS*, 2013.

[13] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004.

[14] Marcy Baughman. The Influence of Scientific Research and Evaluation on Publishing Educational Curriculum. *New Directions for Evaluation*, 117:85–94, 2008.

[15] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When Does a Refactoring Induce Bugs? An Empirical Study. *Source Code Analysis and Manipulation*, 2012.

[16] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, October 2014.

[17] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. *Recommending Refactoring Operations in Large Software Systems*. RSSE, 2014.

[18] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures. *Journal of Systems and Software*, April 2011.

[19] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering*, July 2014.

[20] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory. In *ICSM*, 2010.

[21] Gabriele Bavota, Sebastiano Panichella, Nikolaos Tsantalis, Massimiliano Di Penta, Rocco Oliveto, and Gerardo Canfora. Recommending Refactorings based on Team Co-Maintenance Patterns. In *ASE*, 2014.

[22] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *ICSE*, 2004.

[23] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Ed.)*. Addison-Wesley, 2004.

[24] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of Similarity Metrics for Refactoring Detection. In *MSR*, 2011.

[25] BigLever Gears Tool. `www.biglever.com/solution/product.html`.

[26] Marat Boshernitsan and Susan L. Graham. iXj: Interactive Source-to-Source Transformations for Java. In *OOPSLA Companion*, 2004.

[27] Gilad Bracha. Newspeak. `bracha.org/Site/Newspeak.html`, 2010.

[28] John Brant and Don Roberts. The SmaCC Transformation Engine: How to Convert Your Entire Code Base into a different Programming Language. In *OOPSLA Companion*, 2009.

[29] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, June 2008.

[30] Danilo Bueche. Pure::variants functionality. private correspondence, 2016.

[31] Walter Cazzola and Edoardo Vacchi. @Java: Bringing a richer annotation model to Java. *Computer Languages, Systems and Structures*, 2014.

[32] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. On the Impact of Refactoring Operations on Code Quality Metrics. In *ICSME*, 2014.

[33] Shame Clifford Charles Simonyi, Magnus Christerson. Intentional Software. In *ONWARD! OOPSLA*, 2006.

[34] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental Assessment of Software Metrics Using Automated Refactoring. In *ESEM*, 2012.

[35] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, August 2006.

[36] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *GPCE*, 2006.

[37] Barthélémy Dagenais and Laurie Hendren. Enabling Static Analysis for Partial Java Programs. In *OOPSLA*, 2008.

[38] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *ESEC-FSE*, 2007.

[39] Databases and Software Engineering Workgroup. Featureide. `wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`, 2016.

[40] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M. Ali Shah. On the Existence of High-Impact Refactoring Opportunities in Programs. In *ACSC*, 2012.

[41] Danny Dig. A Refactoring Approach to Parallelism. *IEEE Software*, Jan 2011.

[42] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *ECOOP*, 2006.

[43] Danny Dig and Ralph Johnson. How Do APIs Evolve&Quest; A Story of Refactoring: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, March 2006.

[44] DOPLER: Decison-Oriented Product LIne Engineering for Effective Reuse. `ase.jku.at/modules/product-lines/index.html`, 2016.

[45] Eclipse Bug 217753. `bugs.eclipse.org/bugs/show_bug.cgi?id=217753`.

[46] Eclipse Bug 467019. `bugs.eclipse.org/bugs/show_bug.cgi?id=467019`.

[47] JDT Refactoring Bugs. `www.cs.utexas.edu/~jongwook/r2/jdtrefactoringbugs.html`.

[48] Eclipse Java Development Tools (JDT) Mars 4.5.2. `eclipse.org/jdt/`.

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

[50] Eclipse Luna. `projects.eclipse.org/releases/luna/`.

[51] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In *ECOOP*, 2006.

[52] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM TOSEM*, December 2011.

[53] David Flanagan. *Java in a Nutshell, 5th Edition*. O'Reilly Publishing, 2005.

[54] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[55] Jack R. Fraenkel and Norman E. Wallen. *How to Design and Evaluate Research in Education*. McGraw-Hill, 2009.

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

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[58] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP*, 1993.

[59] Alejandra Garrido. *Software Refactoring Applied to C Programming Language*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[60] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[61] Alejandra Garrido and Ralph Johnson. Challenges of Refactoring C Programs. In *IWPSE*, 2002.

[62] Paul Gazzillo and Robert Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *PLDI*, 2012.

[63] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling Manual and Automatic Refactoring. In *ICSE*, 2012.

[64] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic Testing of Refactoring Engines on Real Software Projects. In *ECOOP*, 2013.

[65] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[66] Jan Hannemann. Aspect-Oriented Refactoring: Classification and Challenges. In *AOSD*, 2006.

[67] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a Refactoring with Rascal and Eclipse. In *WRT*, 2012.

[68] Marc R. Hoffmann. EclEmma 2.3.2. `www.eclemma.org`, 2014.

[69] Peter Höfner and Bernhard Möller. Private correspondence, 2017.

[70] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *AOSD*, 2007.

[71] IntelliJ IDEA 15.0.6. `jetbrains.com/idea/`.

[72] Java Comment Preprocessor. `github.com/raydac/java-comment-preprocessor`.

[73] Wei Jin, Alessandro Orso, and Tao Xie. Automated Behavioral Regression Testing. In *ICST*, 2010.

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

[75] Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring References for Library Migration. In *OOPSLA*, 2010.

[76] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[77] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*, 2011.

[78] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-aware Module System. In *OOPSLA*, 2012.

[79] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[80] Jongwook Kim, Don Batory, and Danny Dig. Scripting Parametric Refactorings in Java to Retrofit Design Patterns. In *ICSME*, 2015.

[81] Jongwook Kim, Don Batory, and Danny Dig. Refactoring and Retrofitting Design Patterns in Java Software Product Lines, 2016.

[82] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving Refactoring Speed by 10X. In *ICSE*, 2016.

[83] Jongwook Kim, Don Batory, and Milos Gligoric. Move-Instance-Method Refactorings: Experience, Issues, and Solutions, 2016.

[84] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: Delta-oriented Programming for Java 1.5. In *PPPJ*, 2014.

[85] Martin Kuhlemann, Don Batory, and Sven Apel. Refactoring Feature Modules. In *ICSR*, 2009.

[86] Huiqing Li and Simon Thompson. *Implementation and Application of Functional Languages*. Springer-Verlag, 2008.

[87] Huiqing Li and Simon Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In *FASE*, 2012.

[88] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware Refactoring in the Wild. In *ICSE*, 2015.

[89] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *AOSD*, 2011.

[90] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the Linux Kernel Variability Model. In *SPLC*, 2010.

[91] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[92] Hayden Melton and Ewan Tempero. Identifying Refactoring Opportunities by Identifying Dependency Cycles. In *ACSC*, 2006.

[93] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based Analysis of Feature Models is Easy. In *SPLC*, 2009.

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

[95] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Scalable Detection of Missed Cross-function Refactorings. In *ISSTA*, 2014.

[96] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Vector Abstraction and Concretization for Scalable Detection of Refactorings. In *FSE*, 2014.

[97] Robert C. Miller and Brad A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX*, 2001.

[98] Miguel P. Monteiro and João M. Fernandes. An Illustrative Example of Refactoring Object-oriented Source Code with Aspect-oriented Mechanisms. *Software: Practice and Experience*, April 2008.

[99] MoveInstanceMethodProcessor.java. `git.eclipse.org/c/jdt/eclipse.jdt.ui.git/plain/org.eclipse.jdt.ui/core%20refactoring/org/eclipse/jdt/internal/corext/refactoring/structure/MoveInstanceMethodProcessor.java`.

[100] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. In *ICSE*, 2009.

[101] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*, 2014.

[102] NetBeans 8.1. `netbeans.org/`.

[103] Bill Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

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

[105] org.eclipse.jdt.ui.tests.refactoring. `git.eclipse.org/c/jdt/eclipse.jdt.ui.git/tree/org.eclipse.jdt.ui.tests.refactoring/`.

[106] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[107] Polymorphism (Computer Science). `en.wikipedia.org/wiki/Polymorphism_(computer_science)`.

[108] Prebop Preprocessor. `prebop.sourceforge.net/`.

[109] pure::variants User Guide. `www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`, 2016.

[110] R2 Design Pattern Scripts. `www.cs.utexas.edu/~jongwook/r2/designpatternscripts.html`.

[111] R3 Manual. `www.cs.utexas.edu/users/jongwook/r3/release`.

[112] Dialect User's Guide, 1990.

[113] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[114] Cagri Sahin, Lori Pollock, and James Clause. How Do Code Refactorings Affect Energy Usage? In *ESEM*, 2014.

[115] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending Move Method Refactorings Using Dependency Sets. In *WCRE*, 2013.

[116] Ina Schaefer and Ferruccio Damiani. Pure Delta-oriented Programming. In *FOSD*, 2010.

[117] Max Schaefer and Oege de Moor. Specifying and Implementing Refactorings. In *OOPSLA*, 2010.

[118] Ina Schäfer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

[119] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Transactions on Software Engineering*, November 2012.

[120] Sandro Schulze, Malte Lochau, and Saskia Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *FOSD*, 2013.

[121] Sandro Schulze, Oliver Richers, and Ina Schaefer. Refactoring Delta-oriented Software Product Lines. In *AOSD*, 2013.

[122] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems. In *GECCO*, 2006.

[123] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting Performance via Automated Feature-interaction Detection. In *ICSE*, 2012.

[124] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending Automated Extract Method Refactorings. In *ICPC*, 2014.

[125] Julio Sincero, Horst Schirmeier, Wolfgang Schrder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In *SPLC-OSSPL*, 2007.

[126] SLASHDEV Preprocessor. `www.slashdev.ca/javapp/`.

[127] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering*, February 2013.

[128] Quinten David Soetens, Javier Pérez, and Serge Demeyer. An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings. In *ICSM*, 2013.

[129] Software Design (CS373S) Course at the University of Texas at Austin. `www.cs.utexas.edu/users/dsb/CS373S/`.

[130] SPLC Product Line Hall of Fame. `splc.net/fame.html`.

[131] Stefan Stanciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wasowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *ICSME*, 2016.

[132] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A Refactoring Constraint Language and its Application to Eiffel. In *ECOOP*, 2011.

[133] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs Under Constrained Accessibility. In *ECOOP*, 2009.

[134] Friedrich Steimann and Jens von Pilgrim. Constraint-Based Refactoring with Foresight. In *ECOOP*, 2012.

[135] Kathryn T Stolee and Sebastian Elbaum. Refactoring Pipe-like Mashups for End-User Programmers. In *ICSE*, 2011.

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

[137] System Generation. `en.wikipedia.org/wiki/System_Generation_(OS)`.

[138] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. Recommending Refactorings to Reverse Software Architecture Erosion. In *CSMR*, 2012.

[139] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *GPCE*, 2007.

[140] Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. In *ASE*, 1999.

[141] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing Duplicated Code with Linked Editing. In *VLHCC*, 2004.

[142] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, May 2009.

[143] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. In *ICSE*, 2012.

[144] Mark G. J. van den Brand, Magiel Bruntink, G. R. Economopoulos, Hayco de Jong, Paul Klint, A. Taeke Kooiker, Tijs van der Storm, and Jurgen J. Vinju. Using The Meta-Environment for Maintenance and Renovation. In *CSMR*, 2007.

[145] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *CC*, 2001.

[146] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a Scripting Language for Refactoring. In *ICSE*, 2006.

[147] VisualVM 1.3.8. visualvm.java.net/.

[148] Eric Walkingshaw and Klaus Ostermann. Projectional Editing of Variational Software. In *GPCE*, 2014.

[149] Lucas Wanner, Liangzhen Lai, Abbas Rahimi, Mark Gottscho, Pietro Mercati, Chu-Hsiang Huang, Frederic Sala, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Puneet Gupta, Rajesh Gupta, Ranjit Jhala, Rakesh Kumar, Sorin Lerner, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steve Swanson, Dennis Sylvester, and Yuanyuan Zhou. NSF Expedition on Variability-Aware Software: Recent Results and Contributions. *Information Technology*, 2015.

[150] Feature model. en.wikipedia.org/wiki/Feature_model.

[151] Jan Wloka, Robert Hirschfeld, and Joachim Hänsel. Tool-supported Refactoring of Aspect-oriented Programs. In *AOSD*, 2008.