Refactoring and Retrofitting Design Patterns in Java Software Product Lines

ABSTRACT
A centerpiece of modern software development is refactoring. Software Product Lines (SPLs), a major software development paradigm, lack tools to refactor Java SPL codebases.

R4 is a new set of design guidelines, techniques, and language constructs to (1) express feature-based Java SPLs using only Java custom annotations, (2) view programs of the SPL, (3) edit views, automatically propagating edits back to the SPL codebase, (4) verify refactoring preconditions are satisfied by the target program as well as all SPL programs, (5) refactor programs, automatically applying a corresponding refactoring to the SPL codebase, and (6) retrofit design patterns into the SPL codebase by executing refactoring scripts. R4 implements a core theorem on refactoring feature-based Java SPLs. Case studies apply 2,316 refactorings to retrofit 64 pattern instances in 8 public Java SPLs and show that R4 is as efficient, expressive, and scalable as a state-of-the-art feature-unaware refactoring engine.

1. INTRODUCTION
An SPL is a family of programs with commonalities [2, 42, 44]. Amortizing the cost to design and maintain these commonalities makes it economical to create SPLs. Programs of an SPL are distinguished by features—in increments in program functionality. Each program in an SPL is defined by a unique set of features called a configuration [2].

A common way to code SPLs is to use #if-#endif preprocessing: declarations and code blocks are labeled with feature presence conditions and are included when particular feature(s) are present in a configuration; otherwise the declarations and blocks are erased [2, 46]. The Linux Kernel is a huge SPL, consisting of 16M LOC and over 6000 features [15, 34, 42]. It uses the C-preprocessor to remove declarations, code, and files to produce the C codebase for a configuration.

The presence or absence of a feature in Java can be encoded by a global static boolean variable; the Java compiler can evaluate feature predicates to remove unreachable code in if(-url_expression) statements. But removing entire class, field and method declarations is not yet possible with existing Java constructs. So Java SPLs are hacked in some manner to achieve this additional and essential effect.

One way is to preprocess a Java SPL codebase P for a given configuration C to produce its codebase P_C; although Java does not officially have a preprocessor, there are unofficial ones [22, 40, 43]. Another way is to copy and assemble code fragments from P to produce P_C [3, 5, 23, 30]. Either way, a separate codebase is created for P_C, followed by run-edit-debug cycles to improve, tune, and repair P_C. This is a common way to develop SPL programs. It also exposes three key limitations in today’s SPL tooling.

First, given an edited program P_C, how are its changes back-propagated into P, the SPL codebase? Manually propagating changes does not scale. AHEAD [5] maintained links from P_C to P and had the first tool to back-propagate changes. Gears [30], a commercial tool for SPL development, has a similar (but in our opinion more elegant) preprocessing solution. All solutions for back-propagation that are known to us rely on preprocessors.

Second, refactoring is a centerpiece in modern software development [6, 35]. Programmers routinely rename program elements and move them using refactorings to improve program structure and clarity. Unfortunately, the same cannot be said for Java SPL development. Despite the significance of refactoring, existing prototype [3, 5, 23, 41] and commercial tools [30] for Java SPL development offer no help to refactor SPLs; only recently has a refactoring engine for C-language SPLs appeared [32]. The main reason is that existing tools rely on preprocessors which lack type information needed for precondition checks and code transformations. Another reason is that some require extensions to the Java language, which is unlikely. Further, refactoring engines and SPL tools are notoriously difficult to build.

Third, refactoring technology has been slow to evolve. Scripting is a key functionality that is missing in major IDEs [7, 20, 45], where scripts are programmatic sequences of refactorings. Most design patterns in the Gang of Four text [16] can be expressed as scripts [27, 28, 48]. R3 [29] is a Java refactoring engine that enables programmers to write and execute such scripts. R3, however, cannot refactor Java SPLs as it is feature ‘unaware’.

In short, Java SPL development is difficult because of (1) the lack of tools to propagate program changes back into an SPL codebase, (2) the inability to refactor SPLs, and to a lesser extent (3) the inability to script refactorings to retrofit design patterns into SPL codebases automatically.
This paper presents R4, the first feature-aware refactoring engine for Java, that solves all three problems. We show how a modification of standard IDE code folding allows us to project program P with configuration C as a ‘view’ of an SPL codebase P. A programmer can edit and refactor view P; behind the curtains R4 applies corresponding edits and feature-aware refactorings to P. R4 leverages scripts of R3 to allow programmers to automatically retrofit design patterns into SPL codebases.

This paper makes several novel contributions:

- New constraints (guidelines) to eliminate semantic ambiguities that arise in annotated Java SPL designs,
- A core theorem that equates a feature-aware refactoring of an SPL program with a feature-aware refactoring of the entire SPL codebase,
- The R4 tool for editing, projecting, and refactoring Java SPLs that implements this theorem,
- The extensions R4 makes to R3 to support feature-aware precondition checks in refactoring SPLs,
- Case studies that apply 2,316 refactorings to retrofit 64 pattern instances in 8 Java SPLs and show R4 is as efficient, expressive, and scalable as a state-of-the-art feature-unaware refactoring engine.

We begin with a gentle overview of standard SPL tools and concepts. We then reveal the novelties of R4: how SPLs are encoded, how editable views are created, the theorem that makes refactoring SPLs possible, and how R4 extends R3.

2. STANDARD SPL TOOLS AND CONCEPTS

Every SPL has a Feature Model (FM) [2]. It is a hierarchy of features that define containment relationships (distinguishing mandatory from optional features, and alternative from multi-choice features) and cross-tree constraints (such as if feature X is selected, so must feature Y).

An FM can be translated to a propositional formula [2]. A SPL configuration tool reads a FM and allows users to select desired features and deselect unwanted features to specify a program. The tool’s responsibility is to guarantee that the selection is legal w.r.t. the FM. To do so, the FM is mapped to formula φ, conjoined with the set of desired features and the negation of undesired features. If features X and Y are selected and Z is deselected, the formula φ ∧ X ∧ Y ∧ ¬Z is submitted to a SAT solver. If satisfiable, at least one program in the SPL has this combination of features. Otherwise, the selection {X, Y, ¬Z} is illegal. A configuration lists every feature (as being selected or not) and this combination is legal w.r.t. the SPL’s FM. A configuration file, typically a CPP #include file, is produced by the configuration tool. The program for this configuration file is produced by C preprocessing the SPL codebase.

Using the C-preprocessor causes all sorts of problems, such as those listed in the Introduction. It also makes it considerably harder to analyze the SPL codebase [23, 24]. Such an analysis is Safe Composition (SC), the verification that every program of an SPL is type safe, i.e., compiles without error [2, 26, 47]. Suppose statement “x=y;” is introduced by feature F, variable x is defined by feature X, and y by Y. This relationship is expressed by the constraint ψ := (F ⇒ X ∧ Y). That is, if the “x=y;” statement appears in a program, so too must the definitions of x and y.

Again, let ψ be the prop formula of the SPL’s FM. If ψ ∧ ¬ψ is satisfiable, then there exists at least one program in the SPL that does not satisfy ψ and hence will not compile [11]. Similarly, dead code is source that appears in no SPL program. Let ψ be the presence condition of code fragment ℓ. If ψ ∧ ¬ψ is unsatisfiable, then ℓ is dead code.

An SC tool calls an SPL codebase P for all distinct ψ constraints and verifies that no program in the SPL violates each constraint. We say P satisfies SC or P is dead code free if there are no violations. Others [2, 23, 24, 47] provide more details.

We are now ready to present the novel ideas of R4.

3. R4

3.1 Encoding Java SPLs with R4

R4 requires every Java SPL to use the custom annotation type Feature, which defines an SPL configuration. 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). Fig. 1 shows a Feature declaration with three features {X, Y, Z} where X and Y are selected and Z is not. The specified configuration is {X, Y, ¬Z}. Feature.java is generated by a SPL configuration tool, mentioned earlier.

Every Java declaration (class, method, field, etc.) and package in P has an optional Feature annotation with a boolean expression of Feature variables. If the expression is true for a configuration, the declaration (or entire package) is present in the program; otherwise it is removed. If a declaration has no Feature annotation, it is included in every program of the SPL.

Fig. 2a illustrates three declarations: Graphics, Square, and Picture. Graphics belongs to every program of the SPL as it has no Feature annotation. Square is added by feature X. Picture is added by feature Y.

Methods and fields are annotated similarly. Fig. 2b shows a declaration of three integer variables (i, j, k), all belonging to feature X; the Feature annotation is for the entire line. If we wanted variables i and j to be introduced by feature X, and k by feature Z, we would use Fig. 2c.

Feature variations in executable code are written using if(feature_expression) statements. For example, it is not uncommon in SPLs to have different bodies for a single method. Suppose features X and Y are never both selected. In preprocessor-based tools, one might use Fig. 4a, where #if!¬Fendif introduces at most one declaration of method m

---

1In Java, each package-level annotation is placed in a package-info.java file.
2To write @Feature(X) instead of #Feature(Feature.X), include import static Feature.*; in all files.
in any program; Fig. 4b shows the cascading if-else statements that we use to encode the same variability using only one declaration for m.

![Figure 3: Code Folding in R4.](image)

**3.2 SPL Codebase Projections**

Let C be the configuration of program P_c. Let P be the SPL codebase and P_c(P) be its C-projection that yields P_c:

\[ P_c(P) = P_c \]  

(1)

Think of P_C as an operation that alters the text of P to produce the text of P_C.

R4 uses two projection operations, P_C^2 and P_C^1, that both satisfy (1). The first, P_C^2, folds lines of code in P, exposing only the source of P_C [10]. This is the source of P that R4 allows programmers to modify. Further, code folding provides the important functionality that (a) shows where variation points in P_C exist and (b) allows programmers to inspect, not edit, their folded contents. Fig. 3a shows P, Fig. 3b shows P_C with folded code when GREEN = false, and Fig. 3c shows P_C with unfolded code when GREEN = false.

**Note:** Variation points (VP) are locations where different SPL programs may differ in source. Knowing the placement of VPs is vital to guarantee a consistent behavior of all variants of packages, classes, and methods; they must all be designed consistently, sharing the same VP structure [2].

The second, P_C^1, does something similar: instead of code folding regions, it simply comments them out. P_C^2(P) is fed to the Java compiler to produce bytecode for P_C; it is this compiled version that allows programmers to execute, debug, and step-through a code folded version of P_C.

In summary, R4 uses P_C^2 to provide an editable view of P_C, and P_C^1 to provide the source of P_C necessary for compilation, execution, and debugging.

**3.3 Improved SPL Design Techniques**

We have authored many Java SPLs. In doing so, we came to the realization that an SPL design is a “master plan” that all SPL programs must conform. Henceforth when we say “program” we mean a product of an SPL.

We realized that an SPL design should follow an element naming convention: *All programs of an SPL use the same name for the same element*. 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 doubles the information a programmer needs to remember: s/he has to know when to use “dd” and when to use “d”. A decent-sized SPL can have thousands or tens of thousands of declarations. To remember all type, method, and variable names is difficult enough, but complicating this knowledge with name variability is untenable. Eliminating *name variability* was a key requirement for (our) sanity in SPL designs.\(^3\)

An ugly relative of name variability is semantic inconsistency: we do not want d to mean one thing in some programs and something radically different (e.g., have a fundamentally different type) in others. Every declaration should have a consistent meaning across programs of an SPL, otherwise the scalability problems similar to name variability arise. More on this in Section 4.4.

An important consequence of eliminating name variability and semantic inconsistency is that the codebase P of an SPL always compiles. (In our case, P compiles by ignoring Feature annotations, discussed in the next section). The compiled version of P need not correspond to an SPL program—it is only a check that *every* reference can be bound to *some* declaration in P [47]. If this is not the case, there may be one program of the SPL that does not compile. The compilability of P is a precondition for safe composition [2, 47], discussed earlier.

Henceforth, we require the following *SPL sanity constraints*:

- **S1.** Absence of name variability,
- **S2.** Absence of semantic inconsistencies, and
- **S3.** Compilability of the Java SPL codebase P.

These constraints not only simplify SPL designs by eliminating “artificial complexity” [8], they also provide an unambiguous way to understand the result of refactoring a Java SPL codebase, a topic which we explore in Section 4.4.

**3.4 Perspective**

Readers who are familiar with today’s SPL tools may recognize R4, as described so far, is not incremental. As said in the Introduction, existing tools rely on preprocessing so that a program P_c projected from P is literally a different codebase [3, 5, 23, 30]. Special (preprocessing) tools are needed to propagate edits of P_c back to P. R4 is different: it relies on

\(^3\)There are many SPLs where products are customized by cloning and then have their own edit histories that give rise to name variability [12]. To us, this is a bad smell or bad SPL design.
Java custom annotations and an extension of standard code-folding tools to provide editable views of SPL programs.

What existing tools lack is the ability to refactor SPL codebases. This is the next contribution of R4 that we discuss, starting with a core theorem.

4. THEOREM FOR REFACTORING SPLS

Feature modules are abstractions that exist in the SPL problem space [2]; theorems derived in the problem space can be mapped to an SPL solution space in many ways (e.g., preprocessor or annotation-based implementations) that preserve theorem validity. This is how we developed R4: we proved a theorem about refactoring feature modules and mapped it to R4’s Java-annotation-based implementation.

Let \( R \) be a feature-unaware refactoring. A programmer applies \( R \) to an SPL program \( P_c \) to produce \( P^R_c \):

\[
R(P_c) = P^R_c \tag{2}
\]

Equation (2) does not say how the SPL codebase \( P \) should be modified to project \( P^R_c \). Fig. 5 depicts the core theorem of SPL refactoring as a commuting diagram [38]. Namely: \( C \)-projecting \( P \) to produce \( P_c \) and then applying \( R \) is equivalent to a feature-aware \( R^C \)-refactoring of \( P \), yielding \( P^R_c \), and \( C \)-projecting it to produce \( P^R_c \). \( R \) and \( R^C \) are identical in terms of their code transformations, but differ in their preconditions. In the following, we prove the theorem of Fig. 5 and explain how \( R \) and \( R^C \) differ.

4.1 Feature Modules and Their Sums

Let \( F \) denote the set of all features of an SPL and let \( F_i \) be the feature module for feature \( i \). Feature modules are composed by the + operation [3,5]. The SPL codebase \( P \) is the sum of all feature modules:

\[
P = \sum_{i \in F} F_i \tag{3}
\]

And the \( C \)-projection of \( P \), where \( C \subseteq F \), yields \( P_c \) which is the sum of all feature modules in \( C \):

\[
\Pi_C(P) = \sum_{i \in C \cap F} F_i = \sum_{i \in C} F_i = P_C \tag{4}
\]

4.2 Code Transformation of Feature Modules

From our experience in developing SPLs, the following distributivity identity suggested itself: **with respect to code transformations and not their preconditions**, an \( R \)-refactoring of a sum of feature modules \( A \) and \( B \) equals the sum of the each \( R \)-refactored feature module:

\[
R(A + B) = R(A) + R(B) \tag{5}
\]

The insight behind (4) is simple: provided that one observes the sanity rules S1-S3 of Section 3.3 — common refactorings are largely oblivious to feature module boundaries. That is, when a program \( P = A + B \) is \( R \)-refactored, one expects both \( A \) and \( B \) to be modified by \( R \), i.e., \( P^R = A^R + B^R \).

**Example:** Method \( m \) in Fig. 6 is defined in class/feature \( A \). Class/feature \( B \) has a call to \( m \). When \( m \) is renamed to \( n \), both features \( A \) and \( B \) are modified to \( A^R \) and \( B^R \).

Without loss of generality, feature interactions can be treated as separate features that are summed like other features [5].

...
Example: A programmer wants to edit the base program \( P_{\text{base}} \) of an SPL whose codebase \( P \) is Fig. 7. Method \( \text{bar} \) is invisible to the programmer as it belongs to feature \( X \) which is unselected. If the programmer tries to rename \( \text{foo} \) to \( \text{bar} \), the rename fails. Reason: there is at least one program in the SPL (any configuration with \( X \)) where the rename refactoring fails, even though renaming \( \text{foo} \) to \( \text{bar} \) in \( P_{\text{base}} \) is legal.

SPL programmers must realize that *refactoring an SPL codebase has more constraints* than just refactoring a single program \( P_2 \). We report 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 \( \psi \) constraint and verifying all SPL programs satisfy \( \psi \). Examples of such constraints are given in Section 5.5.

So the difference between a refactoring \( \mathcal{R} \) on a single program \( P_2 \) and the corresponding refactoring \( \mathcal{R}' \) on \( P \) is lifting preconditions \( \rho \) of \( \mathcal{R} \) to determine if there exists any program in the SPL that fails to satisfy \( \rho \).

4.4 Eliminating Ambiguous Designs

We said earlier that refactorings are largely oblivious to feature module boundaries. Here is a case where they are not and ‘accidental’ or ad hoc polymorphism and subtype polymorphism collide [39]. Ad hoc polymorphism arises when methods can be applied to arguments of different types, but behave differently. Example: classes HandSketch and Chess both have a \( \text{draw()} \) method with very different semantics.

Consider the class hierarchy of Fig. 8. Two mutually-exclusive features, blue-hatched and red-solid, both introduce a method \( m(d) \). These methods never appear in the same SPL program, so they need not have the same semantics – this is ad hoc polymorphism. But any Java programmer who reads the SPL codebase would instinctively expect these methods to be related via subtype polymorphism.

What happens when method \( B.m(d) \) is moved to class \( D \)? With an ad hoc interpretation, the method is moved and the red call is updated to \( d.m(b) \). With a subtype interpretation, the method can be moved only if *it leaves a delegate behind* and all calls remain as \( b.m(d) \).

Our sanity constraints \( S_1-S_3 \) of Section 3.3 eliminate this ambiguity about which interpretation to use. If the \( m \) methods are semantically different, the constraints say they must be given different names, and the result of a move is consistent with that of an ad hoc interpretation. If they are semantically related, the constraints say they have the same name, and the result of a move is consistent with a subtyping interpretation. Either way, \( R_4 \) handles both interpretations by asking users to follow the sanity constraints listed earlier.

In effect, we qualify Liebig’s rule [32]: *Given the sanity constraints \( S_1-S_3 \) of Section 3.3 are satisfied and the SPL codebase satisfies \( SC \), a refactoring \( \mathcal{R} \) of an SPL fails if \( \mathcal{R} \) fails on any program of the SPL.*

5. IMPLEMENTATION

5.1 \( R_3 \)

\( R_3 \) [29] is a Java refactoring engine that refactors programs by pretty-printing Abstract Syntax Trees (ASTs). Unlike standard engines that modify ASTs, pretty-printing never changes ASTs; it only displays a view of ASTs. We briefly explain how \( R_3 \) works in this section.

\( R_3 \) is a Java package that presents Java declarations of a target program as objects which a user can retrieve and manipulate. Methods of \( R_3 \) objects are (a) refactorings such as rename and move, (b) retrievals of other \( R_3 \) objects such as get the member methods of a given class, and (c) creations of other \( R_3 \) objects such as add a new field to a given class. \( R_3 \) objects are tuples in a non-persistent main-memory database and are harvested from the ASTs of the program.

A refactoring script, a programmatic invocation of \( R_3 \) methods, does not modify the target program’s ASTs, but instead modifies the \( R_3 \) database. For example, the rename-method refactoring updates the name field of that method’s tuple. The move-method refactoring updates the “owner” field of that method’s tuple to point to \( R_3 \) tuple of the new “owner” type declaration. In general, a refactoring script to install design patterns is a database transaction – it alters the \( R_3 \) database.

\( R_3 \) integrates the database and AST pretty-printing to produce a refactored program. Fig. 9 illustrates the \( R_3 \) lifecycle: Source is parsed into an AST that references tuples in the \( R_3 \) database. A refactoring script renames variables \( a, b, c, x, y, z \) via tuple modifications. A pretty print of the AST retrieves database tuples to display the new variable names yielding the refactored source.

A consequence of the above and other improvements has lead to a \( 10 \times \) increase in \( R_3 \)’s performance over the Eclipse refactoring engine in terms of refactoring execution speed.

5.2 Refactoring Scripts

Although refactoring preconditions are altered, \( R_4 \) scripts to retrofit design patterns into programs are identical to those of \( R_3 \). In effect, refactoring scripts are feature-unknown; \( R_4 \) makes \( R_3 \) refactorings feature-aware with its precondition checks. Fig. 10 is a makeVisitor script that retrofits a Visitor pattern into an existing program. makeVisitor invokes non-trivial refactorings such as change-method-signature (Line 10) and move-instance-method (Line 11) as well as a Singleton design pattern script (Line 5).

To illustrate, Fig. 11a shows a class hierarchy, rooted by the abstract class Graphic, that consists of four classes each with a distinct draw() method. (For now, ignore the coloring/shading of classes). Fig. 11b sketches the Draw-
Visitor class that is produced by invoking `makeVisitor` on any `draw()` method. The method on which `makeVisitor` is invoked is called a *seed*. Another script, `undoVisitor`, removes an existing Visitor by moving Visitor methods back to their original classes, invoking a different set of refactorings.

Figure 10: R4 `makeVisitor` Method.

In R4, each class has a Feature assignment. The shading in Fig. 11a indicates that Graphic belongs to Base and each other class belongs to a different feature. When R4 moves a declaration, it also moves its Feature annotation. In the case of the move-and-delegate refactoring, which `makeVisitor` uses, the created delegate has the same Feature annotation as its delegated method. Fig. 11b shows the preservation of Feature assignments of moved methods by their coloring/shading.

Figure 11: `makeVisitor` Pattern.

5.3 R4 Pipeline

Fig. 12 shows the execution pipeline of R4. The only differences with R3 are the addition of steps α, β, and γ.

Figure 12: R4 Pipeline.

(C) As Eclipse does not provide AST pretty-printers, R4 transforms the original program and

(D) produces AHEAD parse trees;

(E) R4 links database tuples with AHEAD ASTs;

(F) A design pattern script executes (F1) precondition checks and (F2) database updates;

(γ) R4 performs replacement precondition checks needed only for SPLs (see Section 5.5);

(G) The refactored or pattern-retrofitted codebase is produced by AHEAD pretty-printing.

The use of AHEAD ASTs is an artifact of R3’s implementation. Eclipse does not have the requisite AST pretty printers; AHEAD does. In a non-prototype implementation, steps (C) and (D) would be eliminated entirely.

5.4 Dead Code and Safe Composition Checks

Feature models of SPLs are rather static; they do change but slowly. As said in Section 2, R4 culls P for constraints and collects a large set of theorems to prove. From the tables of Section 6, a crude estimate is about 1 theorem for every 2 lines of source. A saving grace is that the number of unique theorems can be orders of magnitude smaller [47].

R4 leverages the stability of an SPL’s feature model by caching the results of a theorem. When a feature-aware condition is checked, R4 identifies the unique theorems to prove and looks in its theorem cache. Only when a previously unseen theorem 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. The performance of R4 and SC are detailed in Section 6.

5.5 Preconditions for SPL Refactorings

R3 supports 32 different primitive refactorings and uses 39 distinct primitive precondition checks, where each R3 refactoring uses a subset of these 39 checks. R4 supports all of R3’s primitive refactorings and preconditions.

We expected most preconditions of R3 would become feature-aware. Interestingly, only 5 of the 39 required lifting. There are three reasons: (1) Java annotations cannot be attached to arbitrary code fragments, such as a Java modifier. (2) Our sanity guidelines S1-S3 eliminate ambiguities that would otherwise complicate precondition checks and make them feature-aware. And (3) some preconditions are agnostic w.r.t. features (like Declaring Type and Constructor below). Overall, this is good: it tells us that refactoring engines for SPLs can approach the efficiency of feature-unaware refactoring engines. Our experiments in Section 6 explore this conjecture.

Examples of primitive preconditions for the move-instance-method refactoring that are unaffected by features are:

- **Method Modifier** – A method cannot be moved if it has an abstract, native, or synchronized modifier.
- **Declaring Type** – A method cannot be moved if its enclosing type is an annotation or interface.
- **Constructor** – A constructor cannot be moved.
- **Destination Parameter** – A method with a parameter of class C cannot be moved to class C if one of its calls has null as its C argument.

The last example is instructive. Assuming the offending call(s)-with-null are not in dead code, if such a call exists, we know at least one SPL program violates this constraint. This can be checked by a feature-unaware approach.\[5\]

\[5\] The create class refactoring has the precondition that no other class
6. EVALUATION

We evaluated R4 by answering three research questions:

- **RQ1**: Can R4 refactor Java SPLs?
- **RQ2**: Does R4 refactor at interactive speeds?
- **RQ3**: Is there a benefit to theorem caching?

### 6.1 Experimental Set-Up

We evaluated R4 by demonstrating that its scripts could refactor SPL codebases to retrofit design patterns. Our criteria for determining which patterns to apply were those that stressed the capabilities of R4 the most. We chose the makeVisitor and undoVisitor scripts, as they invoke the largest number of R4 refactorings, each using different sets of refactorings (e.g., undoVisitor uses inline and delete-class whereas makeVisitor uses Singleton and create-class). R3, and thus R4, supports 18 of the 23 design patterns in [16] and the remaining 5 patterns do not benefit from automation. We executed R4 scripts that invoked 2,316 primitive refactorings to retrofit 32 makeVisitor and 32 undoVisitor pattern instances, respectively.

There are plenty of public SPLs written in the C language [24, 32, 42]. To our astonishment, our biggest challenge was finding Java SPLs. Surprisingly, there are very few public Java SPLs; most do not include regression tests, in the same package has the same name. This preconditions could indeed be made feature-aware, but just like Destination Parameter, there is no need to do so. If such a class exists, then we know at least one SPL program violates this constraint. A feature-unaware approach suffices.

The presence of method n implies the presence for class C.

When we need to verify that the original program behavior is preserved by R4 refactorings.

We eventually found 8 Java SPLs. Three (AHEAD, Calculator, and Elevator) had regression tests that we could use. Two (Notepad and Sodoku) lacked regression tests but could be checked by manually invoking their GUIs before and after running R4 scripts to verify behavior preservation. The remaining three (Lampiro, MobileMedia, and Prevayler) also lacked regression tests. We did not know how to execute them, so we could only verify that they compiled without errors before and after refactoring.

We selected “seed” methods for makeVisitor that created the largest Visitors in terms of the number of “visit” methods collected in the Visitor class. Since the number of refactoring needed to make a Visitor is proportional to the size of Visitor (the number of “visit” methods), large-sized Visitors were appropriate for R4 evaluations.

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.

### 6.2 Results

#### 6.2.1 Table Organization

Table 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 applied to a distinct “seed” method in the Seed ID column. The third column, # of Refs, is the total number of refactorings executed to make a Visitor for the “seed” method.

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 R4 w.r.t. R3. The next six columns show the times spent on each R3 pipeline step of Section 5.3:

- **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 execute an R3 script which updates the R3 database.
- **Proj (G)**: time to write the refactored code to files.
- **Tot (R3T)**: total time in pipeline stages (B), (E), (F1), (F2), and (G).

The next three columns list the extra computations needed for feature-aware refactorings in R4:

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

By comparing the total times using R3 and R4, we estimate the overhead of feature-aware refactorings in our experiments, the subject of the last column:

8We used profiling tool VisualVM (ver. 1.3.8) [49] to measure CPU times in running the R4 scripts. We repeated each experiment five times and report the average execution time.

---

Figure 13: Inline and After Move Constraint.
• **Overhead:** the overhead percentage in terms of execution time with/without caching.

Table 2 lists the results of *undoVisitor* which has an identical tabular structure.

### 6.2.2 Answers to Research Questions

**RQ1:** Can R4 refactor Java SPLs? R4 successfully applied 64 design pattern instances on our SPLs using a total of 2,316 R4 refactorings; 32 added a visitor pattern and 32 removed a visitor. The most challenging experiments, A5 in Tables 1 and 2, executed 552 primitive refactorings, respectively. Most other experiments required fewer as they have fewer ‘visit’ methods (see Fig. 11).

Our conclusion: R4 can indeed refactor SPL codebases.

**RQ2:** Does R4 refactor at interactive speeds? To evaluate R4 refactoring speed, we used three measures:

1. Consider the execution times for R4 for all *makeVisitor* and *undoVisitor* experiments. The largest R4 experiment, Row A5, took a mere 4.8 seconds. The comparable experiment using R3 took 3.7 seconds. (For a perspective on R3’s improvement over the Eclipse refactoring engine, a comparable refactoring to A5 took Eclipse 298 seconds to execute [28].)

Row L5 also took 4.8 seconds; the comparable experiment using R3 took 3.7 seconds. The numbers for *undoVisitor* in Table 2 are similar. For less demanding scripts – remember: *not individual refactorings* – (pick any non-A or non-L row), all R4 executions complete in under 1.4 seconds; the corresponding R3 executions finish in under 1 second. On average across all experiments, R4 was 36% slower (i.e., .5 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. R4 additionally collects feature presence predicates column (γ); this adds one more second of execution time for the largest SPLs. For smaller SPLs, R4 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, an R4 database can be created with time to spare.

3. Over 80% of Eclipse refactoring execution time is consumed by checking preconditions [29]. In contrast, R3 precondition checking is almost nothing (see column (F1) and [29]). R4 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: R4 refactors SPLs at interactive speeds.

**RQ3:** Is there a benefit to theorem caching? To answer this question, we used two measures:

1. The average overhead for checking feature-aware preconditions in the *makeVisitor* experiment was 41% without caching SAT solutions. With caching, the average overhead dropped to 32%. For a perspective, experiment L5 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 0.07 seconds.

2. Table 3 shows the time and number of SAT problems for dead code and SC checks on the SPLs in Table 1. F satisfying SC and being dead code free is a precondition for R4 refactorings and it is meaningless to perform refactorings on un compilable source. Again, we took two different approaches (non-caching and caching) to measure how much time R4 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 2 shows the result of our *undoVisitor* experiment. Although the total # of refactorings needed for *undoVisitor* is equal to that of *makeVisitor*, the set of refactoring types and corresponding R4 scripts are different and the number of SAT problems to solve for *undoVisitor* is slightly larger than that of *makeVisitor*. On average, the overhead for feature-awareness in *undoVisitor* refactorings was 38% without caching and 32% with caching.

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 uncomplicated. And the theorems to be proven are also simple. Having said this, our observations are consistent with prior work that SAT problems for feature models are ‘easy’ [36].

Our conclusion: theorem caching is beneficial.

### 6.3 Threats to Validity

We would have preferred all SPLs to have regression-tests, with larger codebases, and with feature models with huge sets of products – characteristics of large SPLs. Such SPLs were simply unavailable to us.

The applications in Table 1 use *javapp* to specify features [22]. 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. Fig. 14a shows a compilation unit belonging to feature X using *javapp*. As *imports* cannot be annotated in Java, we assigned feature X to the class declaration A in Fig. 14b. However, in case that class B belongs to X, Fig. 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 Fig. 14c.

![Figure 14: Translation *javapp* to *@Feature* Annotations.](image)

### 7. RELATED WORK

Kim et al. [29] report a user study that showed undergraduate students could write R3 scripts (R4 scripts are no different) and writing scripts significantly improved the success rate over that of manual refactoring. As our work leverages R3, R4 inherits its benefits.

Conditional compilation in Java has taken at least two forms: preprocessors, such as [22, 40, 43], and OO language-extensions to support type safe variability, such as [4, 13, 21]. These latter papers are elegant proposals to extend OO languages with conditionals to enable static variability and type safety using generics. Although this remains an active and basic area, work on variability-aware compilers seems more
### Table 1: makeVisitorResults

<table>
<thead>
<tr>
<th>Seed ID</th>
<th># of Refs</th>
<th>R3 Time (seconds)</th>
<th>R4 Time (seconds)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>54</td>
<td>0.080</td>
<td>0.042</td>
<td>2.16</td>
</tr>
<tr>
<td>A2</td>
<td>56</td>
<td>0.080</td>
<td>0.020</td>
<td>1.801</td>
</tr>
<tr>
<td>A3</td>
<td>56</td>
<td>1.712</td>
<td>0.087</td>
<td>1.919</td>
</tr>
<tr>
<td>A4</td>
<td>124</td>
<td>0.000</td>
<td>0.030</td>
<td>2.079</td>
</tr>
</tbody>
</table>

### Table 2: undoVisitorResults

<table>
<thead>
<tr>
<th>App</th>
<th>Non-caching</th>
<th>Caching</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.228 [9]</td>
<td>0.238 [15]</td>
<td>0.42 (65)</td>
</tr>
<tr>
<td>E</td>
<td>0.385 [5]</td>
<td>0.42 (5)</td>
<td>0.47 (4)</td>
</tr>
<tr>
<td>F</td>
<td>0.470 [0]</td>
<td>0.50 (1)</td>
<td>0.55 (1)</td>
</tr>
<tr>
<td>G</td>
<td>0.590 [0]</td>
<td>0.60 (1)</td>
<td>0.66 (1)</td>
</tr>
<tr>
<td>H</td>
<td>0.700 [0]</td>
<td>0.70 (1)</td>
<td>0.75 [1]</td>
</tr>
<tr>
<td>I</td>
<td>0.810 [0]</td>
<td>0.81 (1)</td>
<td>0.91 [1]</td>
</tr>
<tr>
<td>J</td>
<td>0.920 [0]</td>
<td>0.92 (1)</td>
<td>1.02 [1]</td>
</tr>
</tbody>
</table>

### Table 3: Dead Code and Safe Composition Check Results

<table>
<thead>
<tr>
<th>App</th>
<th>Dead Code</th>
<th>Safe Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.178 [14]</td>
<td>1.130 (16)</td>
</tr>
<tr>
<td>B</td>
<td>0.110 [24]</td>
<td>1.00 (39)</td>
</tr>
<tr>
<td>C</td>
<td>0.228 [9]</td>
<td>0.238 [15]</td>
</tr>
<tr>
<td>D</td>
<td>0.285 [7]</td>
<td>0.32 [1]</td>
</tr>
<tr>
<td>E</td>
<td>0.385 [5]</td>
<td>0.42 (5)</td>
</tr>
<tr>
<td>F</td>
<td>0.470 [0]</td>
<td>0.50 (1)</td>
</tr>
<tr>
<td>G</td>
<td>0.590 [0]</td>
<td>0.60 (1)</td>
</tr>
<tr>
<td>H</td>
<td>0.700 [0]</td>
<td>0.70 (1)</td>
</tr>
<tr>
<td>I</td>
<td>0.810 [0]</td>
<td>0.81 (1)</td>
</tr>
<tr>
<td>J</td>
<td>0.920 [0]</td>
<td>0.92 (1)</td>
</tr>
</tbody>
</table>
active today [25, 32, 50]. In contrast, R4 requires no changes to Java and directly supports feature-variability for view editing, view compilation, and view refactoring, a combination of capabilities that elude preprocessor-based tools.

Among the capabilities of projection that R4 does not support is the removal of method parameters that are Feature-annotated [23, 32]. Consider Fig. 15a. Parameter a is Feature-annotated, meaning that it is removed if X is not a feature of the target configuration. Fig. 15b shows the projected result when ¬X holds.

(b) void m ( Feature X A a ) ( ... )
(b) void m () ( ... )

Figure 15: Parameter Removal by Projection.

R4 does not support method parameter projection; it simply ignores any Feature annotations on parameters of methods or generics. We are unconvinced that parameter projection is a good idea. It is a ugly variant of our sanity checks $S_1$-$S_3$: if method m has 2 parameters in some SPL programs, 3 in others, and 4 in the remainder, it quickly becomes untenable to know which version to use and when – especially if there are many methods like m. There is nothing in R4 that precludes parameter projection other than increased complexity; we leave its necessity for others to decide.

In Section 6 we said there are few large public SPL codebases. Those that are available are written in C with CPP directives. Developing tools to parse C-with-CPP source to analyze the impact of feature variability is extraordinarily difficult and beyond the capabilities of most researchers [9, 24], but are unavoidable if these codebases are to be analyzed. Most of the effort in parsing C-with-CPP deals with the artificial complexity that CPP brings to C [17, 18]. And using these tools is not without effort – the codebase must use disciplined annotations [33]. Further, as OO languages expose more program structure than C, the number of OO refactorings [14] is considerably larger than that for C [17]. Morpheus [32], the first refactoring tool for C-with-CPP, offers three refactorings (rename, lift, and inline).

Figure 16: C-Preprocessor vs Java SPL Idioms.

Another important source of complexity in C-based SPLs are violations of our sanity constraints $S_1$-$S_3$. Example: Fig. 16a shows a common CPP idiom that violates $S_2$: field global has type int when feature X is defined, otherwise it is a bool. Our solution is either to use different variable names or give global a single type. Fig. 16b shows another common CPP idiom for initializing a variable; we express same idea in a slightly more verbose way in Fig. 16c. We believe that future SPLs will be developed with modern OO languages that eschew such artificial complexities.

Kühlmann et al. [31] proposed Refactoring Feature Modules (RFMs). Just as we use the term feature modules to mean building-blocks of SPL programs, an RFM is a feature module or a single program refactoring (e.g., not a refactoring script). An RFM refactoring is feature-unaware and is applied to a feature-unaware program to it to the interface needed by a legacy application. Although RFMs have a name that is suggestive of our work, it does not deal with feature-aware refactorings.

Aspect-aware refactorings [1, 19, 37, 51] are a counterpart to feature-aware refactorings in this paper. The technical issues and solutions explored were specific to AspectJ (e.g., pointcuts and wild-cards), and are not the topics of our work: refactoring feature-based Java SPLs, back-propagation of Java program edits, and refactoring scripts.

8. CONCLUSIONS

Refactoring is a staple of Java program development. It should be a staple of Java SPL development too. R4 is a tool that not only brings critical refactoring support to Java SPLs, it also solves four other vexing problems: (1) propagation of edits and refactorings of SPL programs back to the SPL codebase, (2) scripting refactorings to automatically retrofit SPL codebases with design patterns, (3) not requiring language extensions to Java or a special variability-aware compiler; the standard Java compiler will suffice, and (4) efficiency: although R4 is between 20%-50% slower than R3, a state-of-the-art feature-unaware refactoring engine (which is 10× faster than the Eclipse refactoring engine), our experiments showed that a factor of 50% slower is barely more than a second for large refactoring tasks.

R4 leverages practical experiences in years of Java SPL construction (which we called ‘sanity constraints’ that might otherwise be ‘best practice’ techniques) to eliminate artificial complexities in SPL design. It also leverages a theorem for refactoring feature-based SPLs that reveals a fundamental distribution property (refactorings distribute over feature module compositions) that makes R4 concepts and implementation clean. R4 is a mere 10K Java LOC.

R4 may remedy the awful situation where there are few examples of public Java SPLs to analyze. R4 is an advance in Java SPL tooling and theory, and should encourage the deployment of Java-based SPLs in the future.

9. REFERENCES

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