Refactoring and Retrofitting Design Patterns in Java Software Product Lines

Jongwook Kim
University of Texas at Austin
Austin, TX 78712, USA
jongwook@cs.utexas.edu

Don Batory
University of Texas at Austin
Austin, TX 78712, USA
batory@cs.utexas.edu

Danny Dig
Oregon State University
Corvallis, OR 97333, USA
digd@eecs.oregonstate.edu

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.

X15 introduces our 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. X15 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 X15 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, 43, 45]. Amortizing the cost to design and maintain these commonalities makes it economical to create SPLs. Programs of an SPL are distinguished by features – 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 in a configuration; otherwise the declarations and blocks are erased [2, 47]. The Linux Kernel is a huge SPL, consisting of 8M LOC and over 10K features [34, 43]. It uses the C-preprocessor to remove 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(feature_expression) statements. But removing declarations such as classes, 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. One way is to preprocess a Java SPL codebase P for a given configuration C to produce its codebase PC, although Java does not officially have a preprocessor, there are unofficial ones [23, 40, 44]. Another way is to copy and assemble code fragments from P to produce PC [3, 5, 24, 30].

Either way, a separate codebase is created for PC, followed by run-edit-debug cycles to improve, tune, and repair PC. This is a common way to develop SPL programs. It also exposes four key limitations in today’s SPL tooling.

First, existing SPL tools rely on preprocessors. This makes it very hard to analyze an SPL codebase. Special-purpose and difficult to write variability-aware compilers are needed [6, 15, 25, 32, 51].

Second, refactoring is a centerpiece in modern software development [7, 35]. Until recently, existing prototypes [3, 5, 24, 41] and commercial tools [30] for Java SPL development provided no support to refactor SPLs. Only last year, an engine for refactoring C-language SPLs appeared, offering the inline, rename, and extract refactorings [32]. Reason#1: existing tools rely on preprocessors that lack type information needed for precondition checks and code transformations. Reason#2: some tools require extensions to the Java language, which is unlikely. Reason#3: refactoring engines and SPL tools are difficult to build.

Third, given an edited program PC, how are its changes back-propagated into P, the SPL codebase? AHEAD [5] maintained links from PC 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. But neither tool could correctly propagate changes from PC to P made by refactorings. While renaming a field in PC is easy, references to that field may reside in the SPL codebase that are not in PC. Thus, existing tools will rename some references to a field, but not all, breaking the SPL codebase.

Fourth, refactoring technology has been slow to evolve. Scripting is a key functionality that is missing in major IDEs [8, 21, 46], where scripts are programmatic sequences of refactorings. Most design patterns in the Gang of Four text [17] can be expressed as scripts [27, 28, 49]. 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”.

This paper presents X15, the first feature-aware refactoring-
ing engine for Java, that solves all four problems. We show how a modification of standard IDE code folding allows us to project program $P_X$ with configuration $C$ as a ‘view’ of an SPL codebase $P$. A programmer can edit and refactor view $P_X$; behind the curtains $X15$ applies corresponding edits and feature-aware refactorings to $P$. Further, $X15$ 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-unaware refactoring of an SPL program with a feature-aware refactoring of the entire SPL codebase followed by a projection of that SPL program,
- The $X15$ tool for editing, projecting, and refactoring Java SPLs that implements this theorem,
- The extensions $X15$ makes to $R3$ to encode feature-aware preconditions Java SPL refactorings,
- Case studies that apply 2,316 refactorings to retrofit 61 pattern instances in 8 Java SPLs and show $X15$ 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 ideas. We then reveal the novelties of $X15$: how SPLs are encoded, how editable views are created, the theorem that makes refactoring SPLs possible, and how $X15$ extends $R3$.

2. STANDARD SPL TOOLS AND IDEAS

Every SPL has a Feature Model (FM) [2]. It is a hierarchy of features that defines 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 to must feature Y).

An 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 translated to a propositional formula $\phi$ [2], 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 $\phi \land X \land Y \land \neg Z$ is submitted to a SAT solver. If satisfiable, at least one program in the SPL has this combination of features. Otherwise, the selection $(X, Y, \neg Z)$ 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.

A standard analysis of an SPL codebase is Safe Composition (SC), the verification that every program of an SPL is type safe, i.e., compiles without error [2, 26, 48]. Suppose statement $\text{int } i = 2$; is introduced by feature $F$, variable $x$ is defined by feature $X$, and $y$ by $Y$. This relationship is expressed by the constraint $\psi := (F \Rightarrow X \land Y)$. That is, if the “$x = y$” statement appears in a program, too must the definitions for $x$ and $y$.

Again, let $\phi$ be the prop formula of the SPL’s FM. If $\phi \land \neg \psi$ is satisfiable, then there exists at least one program in the SPL that does not satisfy $\psi$ and hence will not compile [12].

Similarly, dead code is source that appears in no SPL program. Let $\delta$ be the presence condition of code fragment $\ell$. If $\phi \land \delta$ is unsatisfiable, then $\ell$ is dead code.

An SC tool calls an SPL codebase $P$ for all distinct $\psi$ and $\delta$ predicates 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, 24, 25, 48] provide more details.

We are now ready to present the novel ideas of $X15$.

3. $X15$

3.1 Encoding Java SPLs with $X15$

$X15$ 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 $\varPhi$ 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 $\{X, Y, \neg Z\}$. 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. 3a, where #ifndef introduces at most one declaration of method $m$ in any program; Fig. 3b shows the cascading if-else statements that we use to encode the same variability using only one declaration for $m$.

Figure 1: The Feature Annotation Type.

![Figure 1: The Feature Annotation Type.](image1)

Figure 2: Feature Annotations.

![Figure 2: Feature Annotations.](image2)

---

1 In Java, each package-level annotation is placed in a package-info.java file.
2 To write @Feature(X) instead of @Feature(Feature.X), include import static Feature.*; in all files.
3.2 SPL Codebase Projections

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

$$\Pi_C(P) = P_C$$  (1)

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

$X15$ uses two projection operations, $\Pi_{C_0}^P$ and $\Pi_{C_2}^P$, that both satisfy (1). The first, $\Pi_{C_0}^P$, folds lines of code in $P$, exposing only the source of $P_C$ [11]. This is the source of $P$ that $X15$ 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 $\text{GREEN}=\text{false}$, and Fig. 3c shows $P_C$ with unfolded code when $\text{GREEN}=\text{false}$.

Note: Variation Points (VPs) are locations where different SPL programs may differ in source. Knowing the placement of VPs is vital to comprehend the behavior of all variants of packages, classes, methods, etc.; they must be designed with each other in mind, sharing the same VP structure [2].

The second, $\Pi_{C_2}^P$, does something similar: instead of code folding regions, it simply comments them out. $\Pi_{C_2}^P(P)$ 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, $X15$ uses $\Pi_{C_2}^P$ to provide an editable view of $P_C$, and $\Pi_{C_2}^P$ 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.\(^3\)

We expect the semantics of methods and classes, etc. to vary slightly across SPL programs. What we do not want is a declaration to mean one thing in some programs and something radically different (e.g., have a fundamentally different type) in others. We call this semantic inconsistency, an ugly relative of name variability. 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). 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 $P$ [48]. If this is not the case, we know at least one program of the SPL will not compile. The compilability of $P$ is a precondition for safe composition [2,48], discussed earlier.

Henceforth, we require the following SPL sanity constraints:

1. Absence of name variability,
2. Absence of semantic inconsistencies, and
3. Compilability of the Java SPL codebase $P$.

These constraints not only simplify SPL designs by eliminating “artificial complexity” [9], 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 $X15$, as described so far, is not incremental. $X15$ relies on Java custom annotations, not explicit preprocessor directives. $X15$ does not need a variability-aware compiler; a standard Java compiler will do. $X15$ does not need a non-refactoring edit back-propagation tool; all non-refactoring

\(^3\)There are many SPLs where products are customized by cloning and then have their own edit histories that give rise to name variability [13]. To us, this is a bad smell or bad SPL design.
edits are made directly to \( \mathbb{P} \). And \( \text{X15} \) does not produce a separate codebase for an SPL program \( \mathbb{P}_C \); it uses a code folded view of \( \mathbb{P} \).

What existing tools lack is the ability to refactor SPL codebases. This is the next contribution of \( \text{X15} \) that we discuss, starting with a core theorem.

4. THEOREM FOR REFACTORIZING 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 \( \text{X15} \): we proved a theorem about refactoring feature modules and mapped it to \( \text{X15} \)'s Java-annotation-based implementation.

Let \( \mathcal{R} \) be a feature-unaware refactoring. A programmer applies \( \mathcal{R} \) to an SPL program \( \mathbb{P}_C \) to produce \( \mathbb{P}_C^R \):

\[
\mathcal{R}(\mathbb{P}_C) = \mathbb{P}_C^R \tag{2}
\]

Equation (2) does not say how the SPL codebase \( \mathbb{P} \) should be modified to project \( \mathbb{P}_C^R \). Fig. 5 depicts the core theorem of SPL refactoring as a commuting diagram [38]. Namely: \( \mathcal{C} \)-projecting \( \mathbb{P} \) to produce \( \mathbb{P}_C \) and then applying \( \mathcal{R} \) is equivalent to a feature-aware \( \mathcal{R}' \)-refactoring of \( \mathbb{P} \), yielding \( \mathbb{P}_C^{R'} \), and \( \mathcal{C} \)-projecting it to produce \( \mathbb{P}_C^R \): \( \mathcal{R} \) and \( \mathcal{R}' \) 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 \( \mathcal{R} \) and \( \mathcal{R}' \) differ.

4.1 Feature Modules and Their Sums

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

\[
\mathbb{P} = \sum_{i \in \mathcal{F}} \mathbb{F}_i \tag{3}
\]

And the \( \mathcal{C} \)-projection of \( \mathbb{P} \), where \( \mathcal{C} \subseteq \mathcal{F} \), yields \( \mathbb{P}_C \) which is the sum of all feature modules in \( \mathcal{C} \):

\[
\Pi_{\mathcal{C}}(\mathbb{P}) = \sum_{i \in \mathcal{C} \cap \mathcal{F}} \mathbb{F}_i = \sum_{i \in \mathcal{C}} \mathbb{F}_i = \mathbb{P}_C \tag{4}
\]

4.2 Code Transformation of Feature Modules

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

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

The insight behind (5) is simple: provided that one observes the sanity rules \( \text{S1-S3} \) of Section 3.3 -- \textit{common refactorings are largely oblivious to feature module boundaries}. That is, when a program \( \mathbb{P} = \mathbb{A} + \mathbb{B} \) is \( \mathcal{R} \)-refactored, one expects both \( \mathbb{A} \) and \( \mathbb{B} \) to be modified by \( \mathcal{R} \), i.e., \( \mathbb{P}_R^c = \mathbb{A}_R^c + \mathbb{B}_R^c \).

\[^3\text{Without loss of generality, feature interactions can be treated as separate features that are summed like other features [5].}\]

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

Two identities follow from (5). First, \( \mathcal{R} \)-refactoring a \( \mathcal{C} \)-projection of \( \mathbb{P} \) equals \( \mathbb{P}_C^R \):

\[
\mathcal{R}(\Pi_{\mathcal{C}}(\mathbb{P})) = \mathcal{R}(\mathbb{P}_C) \quad \text{// by (4)}
\]

\[
\mathbb{P}_C^R = \mathbb{P}_C^R \quad \text{// by (2)} \tag{6}
\]

Second, \( \mathcal{C} \)-projecting an \( \mathcal{R}' \)-refactored codebase \( \mathbb{P} \) equals \( \mathbb{P}_C^{R'} \):

\[
\Pi_{\mathcal{C}}(\mathcal{R}'(\mathbb{P})) = \Pi_{\mathcal{C}}(\mathcal{R}'(\sum_{i \in \mathcal{F}} \mathbb{F}_i)) \quad \text{// by (3)}
\]

\[
\mathbb{P}_C^{R'} = \mathbb{P}_C^{R'} \quad \text{// by (2)} \tag{7}
\]

Equations (6) and (7) prove the theorem of Fig. 5 when the code transformations of \( \mathcal{R} \) and \( \mathcal{R}' \) are identical. We will see in the next section that the preconditions for \( \mathcal{R} \) and \( \mathcal{R}' \) are \textit{not} the same, which is their distinction.

The theorem is important: it tells us how to “back project” code changes on views (namely \( \mathbb{P}_C \)) to \( \mathbb{P} \) -- whether they are text edits or modifications made by refactorings. That is, if \( \text{X15} \) presents view \( \mathbb{P}_C \) of \( \mathbb{P} \), a programmer can invoke a refactoring \( \mathcal{R} \) to get \( \mathcal{R}(\mathbb{P}_C) = \mathbb{P}_C^R \). But behind the curtains, \( \text{X15} \) is really applying \( \mathcal{R}' \) to \( \mathbb{P} \), and taking its \( \mathcal{C} \)-projection to present \( \mathbb{P}_C^{R'} \) to the programmer (again, \( \mathcal{R} \) and \( \mathcal{R}' \) perform identical code transformations).

Note: There are situations where to correctly edit \( \mathbb{P}_C \), programmers must edit \( \mathbb{P} \). Example: a programmer wants to provide a new body to an existing method. To do so, s/he must edit the \( \mathbb{P} \) definition to achieve the desired projection. \( \text{X15} \) offers a GUI button for users to toggle between editing \( \mathbb{P}_C \) and \( \mathbb{P} \), should the need arise.

4.3 Refactoring Preconditions

Equation (7) does not take into account preconditions for refactorings, which determine whether a refactoring can be applied to all SPL products. We start with the rule of Liebig, et al. \([32]\): A refactoring \( \mathcal{R} \) of a product line fails if \( \mathcal{R} \) fails on any program of that product line. We explain in the next section why we must qualify this rule.

Example: A programmer wants to edit the base program \( \mathbb{P}_{\text{base}} \) of an SPL whose codebase \( \mathbb{P} \) is Fig. 7. Method

![Figure 6: Rename-Method Refactoring.](image-url)
bar is invisible to the programmer as it belongs to feature X which is unselected. If the programmer tries to rename foo to 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 foo to bar in Pbase is legal.

SPL programmers must realize that refactoring an SPL codebase has more constraints than just refactoring a single program P. We report precondition failures of a refactoring R by citing a condition or SPL configuration where it fails. This is done by ‘lifting’ a refactoring precondition to a SC ψ constraint and verifying all SPL programs satisfy ψ. Examples of such constraints are given in Section 5.5.

So the difference between a refactoring R on a single program P and the corresponding refactoring R’ on P is lifting preconditions ρ of R to determine if there exists any program in the SPL that fails to satisfy ρ.

### 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 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 remains as b.m(d).

Our sanity constraints S1–S3 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, X15 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 S1–S3 of Section 3.3 are satisfied and the SPL codebase satisfies SC, a refactoring R of an SPL fails if R fails on any program of the SPL.

### 5. IMPLEMENTATION

#### 5.1 R3

R3 [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 R3 works in this section.

R3 is a Java package that presents Java declarations (class, method, field, etc.) of a target program as objects which a user can retrieve and manipulate. Methods of R3 objects are (a) refactorings such as rename and move, (b) retrievals of R3 objects such as get the member methods of a given class, and (c) creations of other R3 objects such as add a new field to a given class. R3 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 R3 methods, does not modify the target program’s ASTs, but instead modifies the R3 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 R3 tuple of the new “owner” type declaration. In general, a refactoring script to install design patterns is a database transaction – it alters the R3 database.

R3 integrates the database and AST pretty-printing to produce a refactored program. Fig. 9 illustrates the R3 lifecycle: Source is parsed into an AST that references tuples in the R3 database. A refactoring script renames variables a,b,c to 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 10x increase in R3’s performance over the Eclipse refactoring engine in terms of refactoring execution speed.

#### 5.2 Refactoring Scripts

Although refactoring preconditions are altered, X15 scripts to retrofit design patterns into programs are identical to those of R3. In effect, R3 refactoring scripts are feature-unaware: X15 makes R3 refactorings feature-aware with its precondition checks. Fig. 10 is a makeVisitor script that retfits 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 DrawVisitor class that is produced by invoking makeVisitor on

![Figure 7: Rename foo to bar Fails.](image-url)

![Figure 8: Polymorphic Ambiguity.](image-url)

![Figure 9: How R3 Works.](image-url)
any `draw()` method. The method on which `makeVisitor` is
invoked is called a *seed*. Another script, `undoVisitor`, re-
moves an existing Visitor by moving Visitor methods back to
their original classes, invoking a different set of refactorings.

```java
// member of RMethod class
RClass makeVisitor(String N) {
  RClass v = pkg.newClass(N);
  RField singleton = v.addSingleton();
  RRelativeList relatives = this.getRelatives();
  relatives.rename("accept");
  int index = relatives.addparameter(singleton);
  relatives.moveAndDelegate(index);
  v.getAllMethods().rename("visit");
  return v;
}
```

Figure 10: X15 `makeVisitor` Method.

In X15, each class has a Feature assignment. The shading in
Fig. 11a indicates that Graphic belongs to Base and each
subclass belongs to a different feature. When X15 moves a
declaration, it also moves its Feature annotation. In the
case of the move-and-delegate refactoring, which `makeVisi-
tor` uses, the created delegate has the same Feature anno-
tation as its delegated method. Fig. 11b shows the preserva-
tion of Feature assignments of moved methods by their
coloring/shading.

### 5.3 X15 Pipeline

Fig. 12 shows the execution pipeline of X15. The only
differences with R3 are the addition of steps α, β, and γ.

- Figure 12: X15 Pipeline.

(A) Eclipse parses a Java SPL codebase;
(B) X15 populates its database by traversing Eclipse ASTs;
   (a) X15 constructs feature predicates for all identifiers and
       stores them 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;
(C) As Eclipse does not provide AST pretty-printers, X15
    transforms the original program and
(D) produces AHEAD parse trees;
(E) X15 links database tuples with AHEAD ASTs;
(F) A design pattern script executes (F1) precondition checks
    and (F2) database updates;
(γ) X15 performs replacement precondition checks needed
    only for SPLs (see Section 5.5);
(G) The refactored or pattern-retrofitted codebase is pro-
duced by AHEAD pretty-printing.

The use of AHEAD ASTs is an artifact of R3’s imple-
mentation. Eclipse does not have the requisite AST pretty
printers; AHEAD does. In a non-prototype implementation,
steps (C) and (D) will 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, X15 culls pig for constraints
and collects a large set of theorems to prove. From the tables
of Section 6, a crude estimate is about 1 theorem per every 2
lines of source. A saving grace is that the number of unique
theorems can be orders of magnitude smaller [48].

X15 leverages the stability of an SPL’s feature model by
lining the results of a theorem. When a feature-aware
condition is checked, X15 identifies the unique theorems to
prove and looks in its theorem cache. Only when a previ-
ously 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 X15 and SC is 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 refac-
toring uses a subset of these 39 checks. X15 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 param-
  eter of class type C cannot be moved to class C if one
  of its calls has null as its C-type 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.\footnote{The create class refactoring has the precondition that no other class in the same package has the same name. This precondition 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.}

Here are some feature-aware preconditions:

- **Execution Flow** – Fig. 13a shows a feature-unaware class C. 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. However, in the feature-aware class C of Fig. 13b, inlining is allowed if features X and Y are mutually exclusive [32]. Although this example may seem artificial, we did need this check for the undoVisitor script.

- **Binding Resolution** – Before a method is moved, the lifted binding resolution check is performed. That is, the moved method should still be present in all programs in which it appeared before the move and all declarations referenced in its body are still present and visible, otherwise the move refactoring is rejected. Fig. 13c and 13d show 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 ∧ blue ⇒ yellow\footnote{Also, the presence of method n implies the presence of class C.} and after the move (see Fig. 13d) the check becomes red ∧ blue ⇒ yellow.

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 X15 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 5.3:\footnote{In 2004, AHEAD [5] had a larger codebase size than any existing Java SPL, but (a) it was not in a form that we could easily use – we do use one program (Mixin) from the AHEAD Tool Suite and (b) we wanted to use as many Java SPLs as we could that we did not author.}

- **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.
- **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 X15:

- **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.
- * 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.

Our conclusion: X15 can indeed refactor SPL codebases.

RQ2: Does X15 refactor at interactive speeds? To answer this question, we used three measures:
1. Consider the execution times for X15 for all makeVisitor and undoVisitor experiments. The largest X15 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 28.6 seconds to execute [28].)

Row L5 also took 5.5 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 X15 executions complete in under 1.4 seconds; the corresponding R3 executions finish in under 1 second. On average across all experiments, X15 was 36% slower (i.e., 0.5 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. X15 additionally collects feature presence predicates column (α); 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, an X15 database can be created with time to spare.

3. Over 50% of Eclipse refactoring execution time is consumed by checking preconditions [29]. In contrast, R3 precondition checking is almost nothing (see column (F1) and [29]). 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: X15 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. P satisfying SC and being dead code free is a precondition for X15 refactorings and it is meaningless to perform refactorings on uncompilable source. 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 1.03x and SC by 15x.

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 X15 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 [24]. 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-url)

7. RELATED WORK

Kim et al. [20] report a user study that showed undergraduate students could write B3 scripts (X15 scripts are...
no different) and writing scripts significantly improved the success rate over that of manual refactoring. As our work leverages R3, X15 inherits its benefits.

Conditional compilation in Java has taken at least two forms: preprocessors, such as [23,40,44], and Object-Oriented (OO) language-extensions to support type safe variability, such as [4,14,22]. 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 active today [6,15,25,32,51]. In contrast, X15 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.

The ChoiceCalculation [15] is a formal model of variability-aware (not just feature-aware) languages. It permits variations, such as alternate variable names, that X15 and our sanity conditions S1-S3 preclude. Among the capabilities of projection that X15 does not support is the removal of method parameters that are Feature-annotated [15,24,32]. 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.

\( \text{(a) void m (Feature(X) A a ) \{ \ldots \}} \)

\( \text{(b) void m ( ) \{ \ldots \}} \)

Figure 15: Parameter Removal by Projection.

X15 does not support method parameter projection; it simply ignores 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 S1-S3: 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 a. There is nothing in X15 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 very difficult and beyond the capabilities of most researchers [10,25], 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 [18,19]. 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 [16] is considerably larger than that for C [18]. Morpheus [32], the first refactoring tool for C-with-CPP, offers three refactorings (rename, lift, and inline).

The authors report the difficulties to refactor SPLs when physical feature modularity is used. Using the annotative (or implicit feature modularity) approach of X15 was a deliberate design decision of ours to avoid these difficulties. X15 focusses on classical OO feature refactorings and design patterns. But as noted in [42], there are other ‘feature’ refactorings (such as feature partitioning) that X15 does not yet address and is left for future work.

Aspect-aware refactorings [1,20,37,52] 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. X15 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 X15 is between 20%-50% slower than R3, a state-of-the-art feature-unaware refactoring engine (which is 10x 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.

X15 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 and ambiguities in SPL design. It also leverages a theorem for refactoring feature-based SPLs that reveals a fundamental distributivity property (refactorings distribute over feature module compositions) that makes X15 concepts and implementation clean. X15 is a mere 10K Java LOC.

X15 may help to remedy the awful situation where there are few examples of public Java SPLs to analyze. X15 is an advance in Java SPL tooling and theory, and should encourage the deployment of Java-based SPLs in the future.

Another important source of complexity in C-based SPLs are violations of our sanity constraints S1-S3. Example: Fig. 16a shows a common CPP idiom that violates S2: 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 hope/believe that future SPLs will be developed with modern OO languages that eschew such artificial complexities.

Kuhlemann 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 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.

Schultze et al. [42] report experiences on integrating FeatureHouse [3] with refactorings – such as pull up, but also refactorings that partition large features into a composition of smaller features. The authors report the difficulties to refactor SPLs when physical feature modularity is used. Using the annotative (or implicit feature modularity) approach of X15 was a deliberate design decision of ours to avoid these difficulties. X15 focusses on classical OO feature refactorings and design patterns. But as noted in [42], there are other ‘feature’ refactorings (such as feature partitioning) that X15 does not yet address and is left for future work.

Figure 16: C-Preprocessor vs Java SPL Idioms.

Figure 16 shows a common CPP idiom that violates S2: 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 hope/believe that future SPLs will be developed with modern OO languages that eschew such artificial complexities.
9. REFERENCES


