X15: A Tool For Refactoring Java Software Product Lines

Jongwook Kim
Iona College
jkim@iona.edu

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

ABSTRACT

X15 is the first tool that can apply common *Object-Oriented* (OO) refactorings to Java *Software Product Lines* (SPLs). **X15** is also the first tool that programmers can write custom scripts (to call refactorings programmatically) to retrofit design patterns into Java SPLs. We motivate and illustrate **X15**'s unique capabilities in this paper.

ACM Reference format:
Jongwook Kim, Don Batory, and Danny Dig. 2017. X15: A Tool For Refactoring Java Software Product Lines. In *Proceedings of 21st International Conference on Software Product Line, Seville, Spain, 25–29 September, 2017 (SPLC'17),* 4 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Refactoring is a cornerstone of modern Java software development [11]; it should be a cornerstone of modern Java SPL development too. Surprisingly this is not the case as of 2017.

In 2015, the first refactoring engine for SPLs appeared, called **Morpheus** [17]. It supported SPLs coded in C with embedded C *Preprocessor* (CPP) directives, and offered three refactorings: function inline, lift function, and rename.

Why did it take until 2015? There are many reasons; one is that **Morpheus** requires a *variability-aware compiler* (VAC). A VAC integrates CPP directives into the grammar of a host language, in this case C. Let's call this new language C&CPP. Writing a VAC for C&CPP is not simple. And there is the non-trivial challenge to write a refactoring engine for C&CPP.

Now imagine the difficulty of building a VAC for Java&JPP where JPP stands for the *Java Preprocessor* [3, 10, 13],[1] and then writing a refactoring engine for it (where Java easily supports an order of magnitude more refactorings than the C language). We suspect such a compiler+refactoring tool will never be built because of its difficulty.

X15 is the first refactoring engine for Java SPLs. It uses a standard Java compiler and Java custom annotations to express CPP-like directives. **X15** leverages a radically new refactoring engine for Java, **R3** [16], which currently supports over 30 OO refactorings. **R3** enables programmers to write refactoring scripts – programmatic invocations of refactorings – to retrofit classical design patterns [12]. A robust set of benchmarks showed that **R3** performed at least 10× faster than the *Eclipse Java Development Tools* (JDT) refactoring engine [16]. **X15** inherits these capabilities and is minimally slower than **R3**.

This paper explains the unique capabilities of **X15**. Its technical innovations are explained in our SPLC 2017 paper [15].

2 BACK-PROPAGATION CONSIDERED HARMFUL

All SPL tools that we are aware of perform a "projection" operation Π_c on an SPL codebase $\mathbb{P}$ to produce a separate codebase P_c for the product with configuration c. That is, Π_c removes code in $\mathbb{P}$ that is irrelevant to c (or composes required code fragments in $\mathbb{P}$ to produce P_c [2, 5, 18]):

$$\Pi_c(\mathbb{P}) = P_c \tag{1}$$

Today's SPL tools expect P_c to be compiled, run, and edited to make repairs. With back-propagation tools, modifications to P_c can be propagated back automatically to $\mathbb{P}$, thereby updating all products that share features/code with c. Without such tools, changes ~~have to~~ *must* be propagated manually.

Refactorings break this paradigm. Figure 1(a) shows an SPL codebase $\mathbb{P}$ expressed in terms of CPP. When $\mathbb{P}$ is projected (preprocessed) for configuration c with feature F=true, a separate codebase P_c for product c is produced, Figure 1(b). Refactoring P_c, such as renaming variable i to j, modifies P_c to Figure 1(c). When changes to P_c are back-propagated to $\mathbb{P}$, the binding of reference j breaks when feature F is unselected (Figure 1(d)).

```
#ifdef F
int i = 0;
#else
int i = 1;
#endif

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

(a) SPL Codebase $\mathbb{P}$

```
int i = 0;




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

(b) Product P_c with Feature F∈c

```
int j = 0;




int m() { j++; }
```

(c) Rename i to j in P_c

```
#ifdef F
int j = 0;
#else
int i = 1;
#endif

int m() { j++; }
```

(d) Back-propagation of P_c to $\mathbb{P}$

Figure 1: Refactoring-Unaware Back-Propagation.

In a nutshell, refactorings of SPL products are *not edits*. As the above example shows, if a programmer wants to $\mathcal{R}$-refactor the codebase of an SPL product P_c, a corresponding $\mathcal{R}$-refactoring must be applied to $\mathbb{P}$ to update all references to refactored entities. This requirement is captured by the following algebraic identities:

$$\mathcal{R}(P_c) = \mathcal{R}\big(\Pi_c(\mathbb{P})\big) = \Pi_c\big(\mathcal{R}(\mathbb{P})\big) \tag{2}$$

[1] Or choose your preferred preprocessor.

SPLC'17, 25–29 September, 2017, Seville, Spain

ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

That is, $\mathcal{R}$-refactoring codebase P_c must equal a c-projection of $\mathcal{R}$-refactored codebase $\mathbb{P}$. More on this in Section 3.3.

3 FEATURES AND CAPABILITIES OF X15

X15 has 3 key advances that make the refactoring of Java SPLs possible:

(1) Encoding variability in Java using custom annotations,
(2) Editing and viewing individual SPL products without creating separate codebases, and
(3) Implementing the identities in equation (2).

Each is discussed in turn in the following subsections.

3.1 Variability using Java Custom Annotations

X15 uses Java custom annotations in a simple and intuitive way to encode feature variability.

Figure 2(a) shows a **X15** configuration file, where `Feature` is a reserved word in **X15**. Each feature of the SPL is declared as a unique `static boolean` variable. A value is assigned to each feature of an SPL to define an SPL configuration. In this example, feature BLUE is selected and RED is not.

```java
package p;

@interface Feature {
    static boolean BLUE = true;
    static boolean RED = false;

    boolean value();
}
```
(a)

```java
package p;

class Graphic {
    void resize(int rate) {
    }
}
```
(b)

```java
package p;

import static p.Feature.*;

@Feature(RED)
class Square extends Graphic {
    int length = 100;

    void resize(int rate) {
        length *= rate;
    }
}
```
(c)

```java
package p;

import static p.Feature.*;

class Rectangle extends Graphic {
    @Feature(BLUE)
    int height = 10, width = 10;

    @Feature(BLUE)
    void resize(int rate) {
        height *= rate;
        if (RED) {
            width *= rate;
        }
    }
}
```
(d)

Figure 2: An X15-annotated Java SPL.

Next, every Java package, class, interface, field, and method declaration is annotated with a `Feature` expression to define its *presence condition* (in terms of features) when that declaration is to appear in an SPL product [1]. The class `Square` in Figure 2(c) appears in every SPL product whose configuration includes feature RED. If an annotation is absent, the declaration belongs to all SPL products. The `Graphic` class of Figure 2(b) is an example.

Statement blocks are variability-aware if they are enclosed in an `if (feature_expression)` statement. In Figure 2(d), the statement "`width *= rate;`" appears in an SPL product whose configuration includes feature RED.[2] **X15** infers `Feature` annotations to simplify variability specification. The precise set of rules **X15** follows is given in Appendix A.

Of course, **X15** needs a feature model for each SPL. **X15** currently uses `guidsl` of the AHEAD tool suite [5]; other classical feature modeling tools could have been used [9].

3.2 Projection as Code Folding

All SPL tools have a projection operation (Π_c) that removes code from $\mathbb{P}$ that is irrelevant to a product with configuration c. **X15** uses code folding, a standard IDE functionality [7], to display (project) the code of a product. In Figure 3, code folding hides code fragments of unselected features.

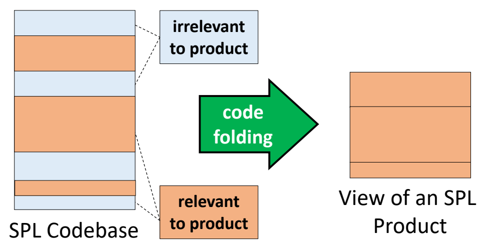

Figure 3: Code Folding reveals P_c.

How is this done? The standard Java compiler generates ASTs that include subtrees for Java annotations. **X15** determines the current configuration c by harvesting boolean values from the AST of declaration `@Feature`. Knowing c, **X15** evaluates the boolean expressions of `@Feature` annotations; if `true` the annotated AST is pretty-printed, otherwise, the contents of the AST are hidden.

In this way, **X15** folds declarations and statements that are irrelevant to P_c. An extra benefit, that we will see later, is that *variation points* (VPs) within an SPL codebase are clearly marked by code fold markers (⊕ meaning folded code and ⊖ indicating expanded code). An **X15** programmer will see where VPs exist in a codebase, and can view – but *not* edit – code-folded regions.

X15 has a second projection operation that comments-out irrelevant contents of $\mathbb{P}$ – this version of P_c is not viewable by a programmer, but is consumed directly by the Java compiler to produce byte codes for P_c. This allows **X15** users to debug a code-folded view of P_c, using the commented-out version compiled by `javac`.

Finally, customer-specific adaptations of SPL products are occasionally useful. The question is: should **X15** create a separate codebase for a product (thereby opening a can of worms requiring back-propagation) or should customer-specific adaptations be integrated into the SPL codebase $\mathbb{P}$ as special features? Currently, we opt for the latter solution.

3.3 Implementing Equation (2)

X15 follows a common path in SPL tool development: feature algebras axiomatize the semantics of feature composition operations. Tools are then developed to implement these algebras [2, 5, 18].

We started with a known feature algebra [4] and recognized a new axiom that distributes refactorings over sums of features. This axiom was used in the proof of Equation (2) [15].

X15 implements Equation (2) in the following way: every **X15** user is always modifying the entire SPL codebase $\mathbb{P}$. Views merely

[2] Due to the nesting of `Feature` references, the presence condition for statement "`width *= rate;`" is RED ∧ BLUE. That is, statement "`width *= rate;`" appears iff its method `resize` is present (BLUE is selected) and if RED is also selected (the statement variability qualifier).

restrict edits to a computable subset of $\mathbb{P}$. When a programmer invokes a refactoring $\mathcal{R}$, the preconditions of $\mathcal{R}$ are applied to all SPL products, using standard SAT analysis techniques (see Section 4.1). If no violations occur, the code transformation of $\mathcal{R}$ is applied to $\mathbb{P}$ to produce $\mathcal{R}(\mathbb{P})$. By Equation (2), a c-projection of $\mathcal{R}(\mathbb{P})$ yields the desired result, $\mathcal{R}(\mathsf{P}_\mathsf{c})$. The variability-aware refactoring preconditions that **X15** uses is documented in [15].

Of course, programmers can refactor $\mathbb{P}$ directly, not requiring a view of a product.

4 DEMONSTRATION

X15 offers four user-interface operations:

(1) checking dead code,
(2) checking safe composition,
(3) code folding to view an SPL product, and
(4) executing **X15** refactorings on an SPL.

The operations are integrated with **X15**'s Eclipse JDT plug-in (Figure 4).

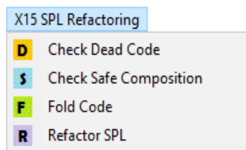

Figure 4: User Interface Operations of X15.

4.1 X15 Analyses

X15 relies on a standard analysis to locate *dead code* – code fragments that belong to no SPL product [1]. **X15** displays a pop-up for any dead code instance found.

X15 uses a similar standard analysis to guarantee all products of an SPL are type-safe (i.e., compilation is error-free) [8, 19]. **X15** displays a pop-up, much like dead code, for any violation found.

Both are SAT analyses [1], which are leveraged by **X15** to verify refactoring preconditions.

4.2 Viewing an SPL Product

As said earlier, **X15** uses code folding to view an SPL product. In Figure 5(a), an `if` statement is folded in class `Rectangle` because its expression RED is `false`. Figure 5(b) shows an expanded view of `Rectangle`.

For the same reason that RED is unselected, class `Square` is folded in Figure 5(c). Figure 5(d) reveals the folded code. **X15** users can fold (or unfold) by clicking either markers ⊕ or ⊖ on the left-hand size of a code fragment. Folded code cannot be edited. Of course, occasionally when a programmer needs to edit folded code, s/he can exit the code-fold mode of **X15** and edit the entire SPL codebase as necessary.

4.3 Refactoring of SPL Products

X15 inherits the **R3**'s ability to allow programmers to write refactoring scripts that automatically retrofit design patterns into a codebase [16]. Figure 6 is an **X15**/**R3** script that creates a visitor given one of the methods that are to be moved into the created visitor class.[3]

Figure 7 shows the SPL codebase of Figure 2 where a Visitor pattern is retrofitted for the method `resize` [12]. The Visitor design

Figure 5: Code Folding as A View of an SPL Product.

```
// member of RMethod class
RClass makeVisitor(String N) {
  RPackage pkg = this.getPackage();
  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 6: R3 makeVisitor Method.

pattern moves all methods with the same signature in a class hierarchy to a visitor class, leaving delegates behind. The moved methods are renamed to `visit` and delegates are renamed to `accept`.

Figure 7(d) shows a visitor that has three `visit` methods which are moved from classes `Graphic`, `Rectangle`, and `Square`, respectively. Note that method `resize` of class `Square` is also moved to the Visitor although it does not exist (i.e., its code was folded) in the SPL product (Figure 5(d)). That shows **X15** applies refactorings to the entire SPL codebase, not a single product as explained in Section 3.3.

5 RECAP

Modern OO design is at least 20 years old [6]. Refactorings and design patterns have been an integral part of modern OO design for at least that long. SPL development is also about 20 years old, but it lacks tools to refactor OO SPL codebases.

X15 is a step forward toward improved tools for OO SPL development. It follows and leverages a history of prior work that is

[3] **R3** scripts can be used without change with **X15**. That is, a script that can be applied to a Java program can also be applied to an **X15** Java SPL.

```java
package p;

class Graphic {
    public void accept(int rate, Visitor visitor) {
        visitor.visit(this, rate);
    }
}
```
(a)

```java
package p;

import static p.Feature.*;

class Rectangle extends Graphic {
    @Feature(BLUE)
    int height = 10, width = 10;

    public void accept(int rate, Visitor visitor) {
        visitor.visit(this, rate);
    }
}
```
(b)

```java
package p;

import static p.Feature.*;

//[Variation_Point_Starts]
@Feature(RED)
class Square extends Graphic {
    int length = 100;

    public void accept(int rate, Visitor visitor) {
        visitor.visit(this, rate);
    }
}
//[Variation_Point_Ends]
```
(c)

```java
package p;

import static p.Feature.*;

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

    void visit(Graphic graphic, int rate) {
    }

    @Feature(BLUE)
    void visit(Rectangle rectangle, int rate) {
        rectangle.height *= rate;

        //[Variation_Point_Starts]
        if (RED) {
            rectangle.width *= rate;
        }
        //[Variation_Point_Ends]
    }

    //[Variation_Point_Starts]
    @Feature(RED)
    void visit(Square square, int rate) {
        square.length *= rate;
    }
    //[Variation_Point_Ends]
}
```
(d)

Figure 7: A Visitor Pattern.

detailed in our SPLC 2017 paper [15]. A demo video of **X15** in action is available at our project webpage [20].

Acknowledgments. We gratefully acknowledge support for this work by NSF grants CCF-1212683, CCF-1439957 and CCF-1553741. We also thank the referees for their helpful comments.

REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines.* Springer, 2013.

[2] S. Apel, C. Kästner, and C. Lengauer. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *ICSE*, 2009.

[3] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized Types for Java. In *POPL*, 1997.

[4] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *GPCE*, 2011.

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

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1998.

[7] Code Folding. https://en.wikipedia.org/wiki/Code_folding.

[8] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

[9] Databases and S. E. Workgroup. Featureide. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/, 2016.

[10] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and Generalized Constraints for C# Generics. In *ECOOP*, 2006.

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

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

[13] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *AOSD*, 2007.

[14] J. Kim, D. Batory, and D. Dig. Refactoring and Retrofitting Design Patterns in Java Software Product Lines, 2016.

[15] J. Kim, D. Batory, and D. Dig. X15: A tool for refactoring java software product lines. In *SPLC*, 2017.

[16] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving Refactoring Speed by 10X. In *ICSE*, 2016.

[17] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware Refactoring in the Wild. In *ICSE*, 2015.

[18] I. Schäfer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *SPLC*, 2010.

[19] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, 2007.

[20] X15 Project. http://www.cs.utexas.edu/users/jongwook/x15/.

A ANNOTATION RULES OF X15

Figure 8 lists the declaration types that can be annotated in Java. **X15** allows all annotations except those on formal parameters of methods [14].

```
Package (declared in package-info.java)
Type declaration (i.e., class, interface, enum)
Annotation declaration
Method declaration
Constructor declaration
Field declaration (including enum constant)
Formal and exception parameter
Local variable (including loop variable of for statement
  and resource variable of try-with-resources statement)
```

Figure 8: Java Entities that Allow Annotations.

X15 infers annotation(s) for unannotated entities. (Inferred annotations are not added to the source code.) The purpose of inferencing is to avoid annotating every declaration repeatedly when successive annotations are identical. Here are the rules for applying annotations in **X15**. A rule with a lower # has higher priority.

Rule #1. Use the annotation(s) of the current declaration.

Rule #2. If Rule #1 is not applicable, use the annotation(s) of the closest preceding declaration in the same scope.

Rule #3. If Rule #2 is not applicable, use the annotation(s) of the closest enclosing declaration.

Rule #4. If Rule #3 is not applicable, use a default feature, which is `BASE`, whose contents is present in every SPL product.

X15 uses `if(feature_expression) {block}` to conditionally include the codeblock `block` in a product; `feature_expression` must be `true` for inclusion to occur; otherwise `block` is erased along with the `if` statement. This action is independent of the above rules.