

# Variation-Preserving Refactoring in Feature-Oriented Software Product Lines

Sandro Schulze  
University of Magdeburg  
sansschul@ovgu.de

Thomas Thüm  
University of Magdeburg  
tthuem@ovgu.de

Martin Kuhlemann  
University of Magdeburg  
mkuhlema@ovgu.de

Gunter Saake  
University of Magdeburg  
saake@ovgu.de

## ABSTRACT

A software product line (SPL) is an advanced concept to manage a family of programs under one umbrella. As with stand-alone programs, maintenance is an important challenge within SPL engineering. One pivotal activity during software maintenance is refactoring; that is, restructuring a program's source code while preserving its external behavior. However, for SPLs, this definition is not sufficient because it does not take into account the behavior of a *set* of programs. In this paper, we focus on the specific requirements for applying refactorings in feature-oriented SPLs. We propose *variant-preserving* refactoring for such SPLs to ensure the validity of all SPL variants after refactoring. Furthermore, we present a first approach how the traditional refactoring definition can be extended so that it can be applied to SPLs based on feature-oriented programming. Finally, we state our experiences of applying such refactorings for the removal of code clones in feature-oriented SPLs and discuss the generalizability for other SPL implementation techniques.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D2.2 [Software Engineering]: Design Tools and Techniques

## General Terms

Design, Languages

## Keywords

Software Product Lines, Refactoring, Feature-Oriented Programming, Code Clones

## 1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VaMoS '12 January 25-27, 2012 Leipzig, Germany  
Copyright 2012 ACM 978-1-4503-1058-1 ...\$10.00.

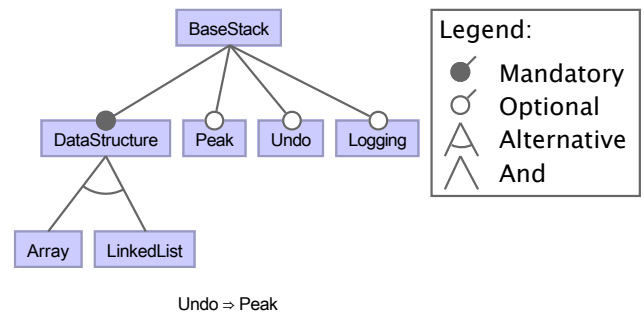


Figure 1: A feature model specifying valid feature combinations of a Stack product line.

A software product line (SPL) enables the programmer to manage a set of programs by maintaining a common code base [2]. As a result, an SPL allows for structured reuse of code to create highly customized software systems. To this end, a product line can be divided into *features*, where a feature is an increment in functionality that is visible for end-users [2]. A programmer can select a subset of all features to generate a program (variant) that is tailor-made to his requirements.

A *feature model* can be used to manage commonalities and variabilities in an SPL [23]. In Figure 1, we give a feature model for an exemplary product line of stacks (Stack SPL). Features can be *optional* or *mandatory* and beyond that, they can be grouped to express additional dependencies. For instance, the features *Array* and *LinkedList* are alternative features that is, both features are mutually exclusive and thus cannot be selected for the same program. Finally, *cross-tree constraints* can be used to indicate dependencies between features across the feature model. In our example, we define the constraint that feature *Undo* requires feature *Peak*. Consequently, feature *Undo* can only be selected if feature *Peak* is selected, too.

Since SPLs evolve even more than stand-alone programs, maintenance of SPLs is an important issue. For instance, new or changed requirements may induce a change to the feature model or source code of an SPL. Furthermore, poor code quality caused by improper design decisions may require changes to the SPL. For instance, replicated code fragments (*code clones*) are often considered harmful and may be removed in many cases [7, 12]. *Refactoring* is useful for

code clone removal since it guarantees that changes to source code are behavior-preserving. Initially, refactoring has been defined by Opdyke as a maintenance process that changes the internal structure of a program but does not affect its external behavior [19]. Based on this work, Fowler presented numerous refactorings in a catalogue-like manner [11]. However, we and others argue that refactoring in SPLs goes beyond traditional (object-oriented) refactoring approaches and thus, has to be revisited for product lines [8, 16].

In this paper, we propose an extension of existing refactoring techniques for *feature-oriented programming (FOP)*, a compositional approach for product line development. In particular, we make the following contributions:

- We point out limitations of traditional object-oriented refactorings for SPLs.
- We propose a definition for refactoring of SPLs in a *variant-preserving* way.
- We provide exemplary refactorings for feature-oriented SPLs in a catalogue-like manner.
- Additionally, we discuss the generalizability of our definition and the actual refactoring towards annotative SPL implementation techniques.

The paper is structured as follows. In Section 2 we introduce important concepts of feature-oriented programming for SPL development. Afterwards, we point out why traditional refactoring is not sufficient for software product lines. In Section 4, we present our definition of variant-preserving refactoring and discuss some special cases. Afterwards, we present some exemplary refactorings for FOP in a catalogue-like manner in Section 5. Additionally, we discuss the generalizability of our approach in Section 6. We finish the paper with related work (Section 7) and a conclusion (Section 8).

## 2. FEATURE-ORIENTED PROGRAMMING

Feature-oriented programming (FOP) is a compositional approach that enables SPL implementation by feature-oriented software development [2]. In this section, we introduce main characteristics of this approach. Furthermore, we explain the relation between FOP and collaboration-based design.

### 2.1 Software Product Line Implementation

Implementing a software product line with FOP is mainly characterized by decomposing a program into modular implementation units. Several languages and tools exist that support feature-oriented product-line implementation such as AHEAD [6], FeatureHouse [3], or FeatureC++ [5]. The core idea of the decomposition is that all artifacts (e.g., source files, config files) that belong to a certain feature are modularized into one cohesive unit. This unit is called a *feature module* and we can map this module directly to its corresponding feature in the feature model. A feature by our means is an user-visible increment in functionality [10]. In FOP, this increment is realized by adding new structures such as classes or refining existing structures such as extending a method.

In Figure 2, we show an exemplary feature-oriented implementation for our Stack SPL. Feature *BaseStack* is the base implementation of the SPL where class `Stack` is declared initially. Moreover, feature *Peak* realizes an increment in functionality by adding a new method `peak()`. Finally, feature *Undo* introduces a new field, a new method, and, beyond that, extends the method `pop()`.

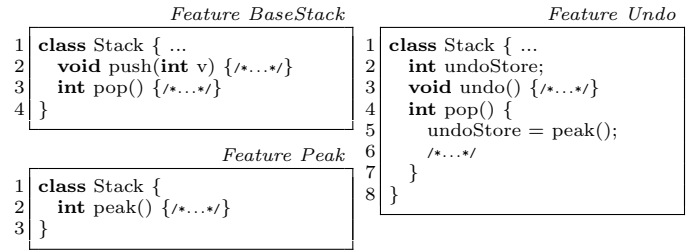


Figure 2: Feature-oriented implementation of the Stack product line containing features *Peak* and *Undo*.

### 2.2 Collaboration-Based Design

To understand why refactoring in FOP is different from object-oriented refactoring, it is important to know how features and classes are related in feature-oriented programs. Technically, FOP is based on collaboration-based design and mixins [22, 24, 9]. Collaboration-based design aims at implementing object interdependencies. In other words, with collaboration-based design, an object (or class) can have different *roles* in different collaborations. A *collaboration* is the implementation of a feature.

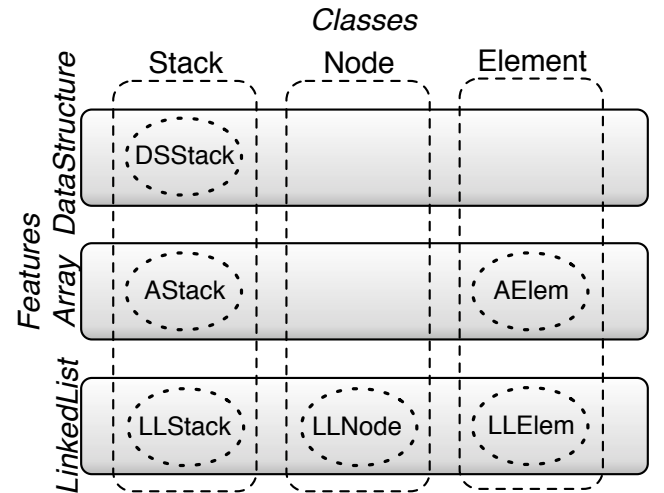


Figure 3: Collaboration diagram of the stack SPL with three classes and three features (excerpt).

For better illustration, we show an exemplary *collaboration diagram* of our Stack SPL in Figure 3. The diagram consists of three different classes (`Stack`, `Node`, `Element`) and features (`DataStructure`, `Array`, `LinkedList`). Each class can play a role with respect to a certain feature or not. In FOP, a role is characterized as an increment in functionality such as extending or adding a method. For instance, class `Stack` has a role for each feature of the diagram (`DStack`, `AStack`, `LLStack`). In contrast, class `Element` has only two roles (`AElem`, `LLElem`) because it does not participate in feature `DataStructure`. Likewise, in the exemplary implementation of our Stack SPL, class `Stack` has a role in feature *Peak*, where it implements a new method `peak()` and thus, extends the functionality (cf. Figure 2). In the following Section, we figure out how collaboration-based design affects refactoring of feature-oriented software product lines.

### 3. LIMITATIONS OF TRADITIONAL REFACTURING

Originally, refactoring is applied for the purpose of improving the structure of source code. This, in turn, requires at least a vague idea of what makes source code good or bad. For object-oriented programs, Fowler proposed a comprehensive overview of *code smells* (e.g., design flaws, replicated code) and how to remove these smells by application of refactorings [11]. Although such an overview of code smells does not exist for FOP, many situations such as evolution or increasing maintainability require refactoring. Particularly, we argue that we can even adopt code smells for FOP as in traditional refactoring. For instance, there may be a method in one feature that is more frequently used by another feature. Hence, it would be useful to move this method into the latter feature.

Unfortunately, it is not as easy to adopt object-oriented refactoring techniques for feature-oriented SPLs<sup>1</sup>. One reason is that in FOP, classes *collaborate* with other classes to implement a feature, which introduces an additional dimension for refactoring. In Figure 4, we show the resulting dimensions that have to be considered for refactoring. For dimension  $D2$ , refactoring takes place within one or between different classes of one feature. Hence, the dependencies, specified in the corresponding feature model, between the target feature (i.e., the feature that is subject to the refactoring) and other features do not directly affect the actual refactoring. Rather, these dependencies have to be considered after refactoring such as updating references to methods that have been moved or renamed. From our perspective, this relates to the traditional object-oriented refactoring, because such updates to references are common as well.

In contrast, dimensions  $D1$  and  $D3$  involve multiple features and thus all of these features and their dependencies, revealed by the feature model, have to be considered for the refactoring of the feature-oriented SPL. Hence, we have to consider additional information such as constraints or dependencies between features for applying a refactoring. Otherwise, the refactoring may lead to unwanted behavior of single variants or decreased variability.

For instance, if we move a method from feature *Array* to feature *LinkedList* (both are alternative features) in our Stack SPL (cf. Figure 1), all variants with the feature *LinkedList* retain their behavior. By contrast, variants with feature *Array* and without *LinkedList* may exhibit inadvertently changed behavior due to dangling references, because the method is not accessible anymore for these variants. Consequently, we cannot guarantee unchanged behavior for all variants when applying the refactoring.

According to traditional refactoring, where the preservation of the external behavior is a mandatory requirement, for refactoring in FOP, preserving the variability as well as the unchanged behavior of each variant is mandatory. We comprise both aspects within the term *variant-preserving refactoring*, which is introduced in the following Section.

### 4. VARIANT-PRESERVING REFACTURING

In this section, we give a concrete definition for the term variant-preserving refactoring. Furthermore, we introduce

<sup>1</sup>We use the terms feature-oriented program and feature-oriented SPL synonymously throughout this paper.

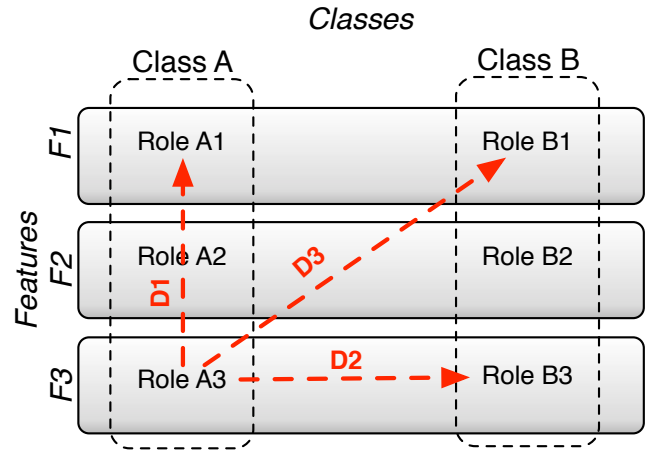


Figure 4: Different dimensions of refactoring in feature-oriented programming.

some specific aspects that have to be considered for refactoring in FOP.

#### 4.1 Definition

As indicated in the previous section, refactoring of feature-oriented programs can involve two parts of the product line. First, the feature model that reflects the domain knowledge (*problem space*) and, second, the source code that implements the functionality of the SPL (*solution space*). We define a *variant-preserving refactoring* as follows:

DEFINITION 1. A change to the feature model or the implementation of features or both is called variant-preserving refactoring if the following two conditions hold:

1. Each valid combination of features remains valid after the refactoring, whereas the validity is specified by the feature model.
2. Each valid combination of features that was compilable before can still be compiled and has the same external behavior after the refactoring.

With the first condition, we address refactorings that are applied to the feature model. In particular, this condition ensures that all combinations of features specified as valid in the before-edit version of the feature model are also valid in the after-edit version. Note that by our definition new variants are allowed as long as they do not affect existing feature combinations. With the second condition, we target refactorings that operate on the implementation of features. Following our definition, such refactorings must not lead to changed behavior of any program of the SPL. Here, we focus on the second condition only, while refactorings for feature models are discussed elsewhere [23, 8].

#### 4.2 Aspects of Refactoring in FOP

Although the definition of variant-preserving refactoring given above is sufficient to cope with the specific FOP characteristics, we identified two specific aspects that require additional treatment. In the following, we explain these cases and discuss, how they affect the application of refactoring in FOP.

**Root Feature as a Target.** The first case we consider is the refactoring of a code fragment to the root feature (e.g., feature *BaseStack* in Figure 1) of a feature-oriented SPL. Theoretically, using this feature as the target feature for a certain refactoring is often possible, because the root feature is selected per default for each variant of the product line. However, in practice this procedure comes at costs of intrinsic implications.

First, using the root feature as target feature decreases the cohesion of the source code that implements a certain feature. Typically, FOP aims at implementing a certain feature in a cohesive unit, i.e., the corresponding feature module. However, taking off code fragments from a feature module (and moving it to the root feature) breaks with the modular implementation and thus may decrease the cohesion within the feature module. Specifically, this refactoring is a kind of generalization, because the refactored code is moved to the most general (and meaningless) feature of the SPL.

Furthermore, preserving the variability can be difficult in certain cases if the root feature is the target feature for refactoring.

**Intra- vs. Inter-Feature refactoring.** Another aspect that must be considered thoroughly are the features that are involved in the actual refactoring process. We differentiate between two kinds of refactoring: First, if the refactoring affects more than one feature we call this an *inter-feature refactoring*. Amongst others, moving a method from one feature to another one is an example for such a kind of refactoring. Second, if the refactoring affects only one or more classes of the same feature, we call this an *intra-feature refactoring*. For that kind of refactoring, extracting a method (affects one class) or moving a method between classes in *one* feature are exemplary refactorings. As a result of this distinction, we can establish a relation between the different possible dimension of refactoring in Figure 4 and the two kinds of refactoring mentioned above: While inter-feature refactoring takes place along the dimensions *D1* and *D3*, intra-feature refactoring takes place along dimension *D2* (including refactoring within one class).

From our point of view, this has the following implications: For inter-feature refactoring, the programmer has to ensure that the actual refactoring does not violate the conditions of Definition 1. Consequently, if we want to restructure the SPL across feature boundaries, we have to apply FOP-specific refactorings (as introduced in the following section) to preserve the variability of the SPL.

However, for intra-feature refactoring, we can apply the traditional, object-oriented refactorings and add only additional checks (taking all features into account) after the refactoring to detect and update incorrect references. We can do this, because intra-feature refactoring *does not* affect the variability directly and thus is inherently variant-preserving. Nevertheless, we have to take feature semantics into account to some extent. First, if we check the pre-conditions (i.e., conditions that have to be fulfilled to apply a refactoring), we probably must take into account the relation between features or roles respectively to decide whether the refactoring is applicable or not. Second, after the application of an intra-feature refactoring (using traditional refactoring techniques), we possibly have to update references (to fields or methods) in more features than the target feature. For instance, assume that we want to move a method `foo()` from class *A* to a class *B* for Feature *F3* in

our example in Figure 4. Consequently, we can apply the *Move Method* refactoring proposed by Fowler [11]. In case that the corresponding role *B3* does not exist, we have to introduce it, that is, creating the class *B* for Feature *F3*. Additionally, we may have to update references to the original method in class *A* to the new class *B* to avoid dangling method references.

As conclusion, we argue that refactoring feature-oriented SPLs includes both, FOP-specific as well as traditional refactorings, depending on the features involved in the refactoring process. The challenging task is to identify which refactoring approach to chose in which situation and how to deal with additional mechanics for traditional refactorings.

## 5. A CATALOGUE FOR FOP REFACTORING

In this section we present four refactorings for feature-oriented software product lines in a catalogue-like manner. Note that we focus on inter-feature refactorings (i.e., dimensions *D1* and *D3* in Figure 4) within our catalogue, because these refactorings require considerable changes with respect to the original object-oriented refactorings. Consequently, the presented refactorings are rather extensions of the object-oriented ones proposed by Fowler [11] by taking feature semantic into account. Furthermore, we present the FOP refactorings in the same way as Fowler so that programmers can easily understand their mechanics and application (assuming that they have knowledge of the original refactorings). We are also inspired by Monteiro who did this in a similar way for aspect-oriented programming [18].

The first refactorings we present are an adaptation of the *Pull Up Field* and *Pull Up Method* refactoring of Fowler. However, for FOP we consider features as source and target of the refactoring rather than classes (although the latter are also taken into account). Consequently, we consider the *refinement hierarchy* instead of *class hierarchy* for the application of these refactorings.

### *Pull Up Field to Parent Feature*

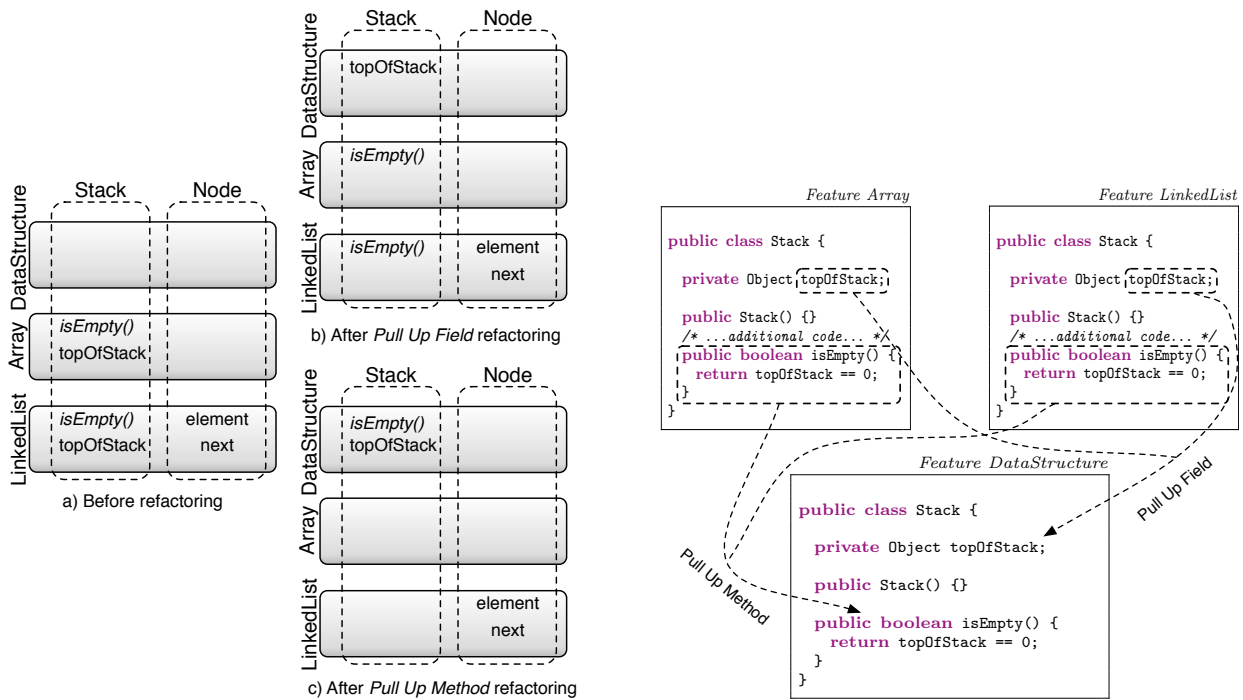
**Typical situation:** A feature has sub features, that are similar to some extent. Specifically, certain fields could be duplicates. Hence, the programmer can reduce duplication by generalizing these fields.

**Recommended action:** Move up fields that are identical (e.g., type and initialization) across sub features and used in the same way in the respective features (e.g., referenced by the same methods).

**Pre-conditions:** First, features, containing the identical field(s) have the same, direct parent feature. Second, the field is contained in the same class in the respective features.

**Mechanics:**

1. Check whether the candidate fields are used in the same way (e.g., qualifier).
2. Check whether the pre-conditions, mentioned above, are fulfilled.
3. In the case, that the class containing the candidate fields has no role in the target feature (i.e., is not implemented in the target feature), create the class for the target feature.
4. If the candidate fields have different names, rename the respective fields so that they all have the name that should appear after the refactoring in the target



**Figure 5:** On the left side, we depict the corresponding collaboration diagram before and after *Pull Up Field* and *Pull Up Method* refactoring. On the right side, we show excerpts of the implementation of features *Array*, *LinkedList* (before refactoring) and feature *DataStructure* (after refactoring). The refactored code fragments are marked by a dotted rectangle.

feature.

5. Create a new field in the respective class of the target feature.
6. If the original fields were private and the refactoring is applied along dimension D3 (cf. Figure 4), we have to change the access modifier of the new field to **protected** so that the sub features can access it.

**Example:** In the left part of Figure 5, we illustrate this refactoring with collaboration diagrams and an excerpt of the implementation of our Stack SPL, respectively. The two alternative features *Array* and *LinkedList* each contain a field `topOfStack` in class *Stack* that represents the first element of the stack. Since both fields as well as the features they belong to fulfill the (pre-)conditions, we apply the pull up field refactoring. First, we create a new field in the class *Stack* in the direct parent feature *DataStructure* (cf. Figure 1). Afterwards, we delete the original fields in feature *Array* and *LinkedList*. On the right side of Figure 5 we show the corresponding code fragments for the features involved in the refactoring. Since the refactoring takes place along dimension D1, we do not have to change the access modifier of the field. The reason is that we move the field to another feature but it remains within the same class *Stack*. Hence, it is accessible within the target feature even though it is private. For a comprehensive overview of access modifiers in feature-oriented programming, we refer to the work of Apel et al. [4].

### *Pull Up Method to Parent Feature*

**Typical situation:** Methods that are (semantically or syntactically) identical occur in different (sibling) features.

**Recommended action:** Move the subject method up to the parent feature of the two sibling features that contain the identical method.

**Pre-conditions:** First, the features, containing the identical method(s) have the same parent feature. Second, the subject method is implemented by the same class within the respective features.

**Mechanics:**

1. Identify the identical methods.
2. Check whether the pre-conditions, mentioned above, are fulfilled.
3. If the candidate methods have different signatures, change the signatures so that they comply with the one you want to have in the target feature.
4. In the case that the class containing the candidate methods has no role in the target feature create the class for the target feature.
5. Create a new method in the corresponding class in the target feature and copy the body of one of the candidate methods to it (and adjust the new method if necessary). If fields are accessed directly within the method (i.e., not passed as parameter), move the field(s) to the target feature using *Pull Up Field to Parent Feature*. Check for dangling references regarding the moved field and update them.
6. Delete the remaining methods in the sub-features.

**Example:** Similar to the *Pull Up Field* refactoring, we illustrate this refactoring by means of our Stack SPL. The two features *Array* and *LinkedList* contain a method `isEmpty()` (in class *Stack*) that checks, whether the current stack is empty or not. Both methods are identical and the two fea-

tures have a common parent feature, namely *DataStructure* (cf. Figure 1). Hence, the pre-conditions for the refactoring are fulfilled. Next, we create a new method in class *Stack* of feature *DataStructure* and copy the body of one of the methods of the two candidate features. Since the original method(s) reference a field, we also have to apply *Pull Up Field*. We show the corresponding collaboration diagrams and exemplary implementation in Figure 5. Finally, we delete the original methods in features *Array* and *LinkedList*.

Next, we present FOP-specific adaptations of the *Move Method* and *Move Field* refactoring, respectively. Specifically, we focus on moving a method/field along dimension D1 (cf. Figure 4), because this refactoring affects only one class. For dimension D3, the refactorings would be more complex, because we have to consider access modifiers and accessors of fields that are moved across classes. Hence, we limit our considerations to dimension D1 only and set aside the discussion on how to move units along dimension D3 for future work.

### Move Method Between Features

**Typical situation:** A method uses or is used by another feature more than by the one it is currently contained in (e.g., indicated by caller/callee relations). Hence, by moving this method to the feature where it is referenced more frequently (or even exclusively), we can increase feature cohesion. Alternatively, we can apply this refactoring to move a method, which is replicated throughout the source code, in a common feature without decreasing the overall variability of the SPL.

**Recommended action:** Move the method from its current role to the role of the target feature.

**Pre-conditions:** The method must be available for all variants as before the refactoring. For instance, if a feature *F1* references a method of feature *F2* and this method is moved to another feature, the moved method must be accessible for feature *F1* in all variants containing *F1*. A programmer has to ensure this *before* the actual refactoring takes place.

#### Mechanics:

1. Examine the source feature for code fragments (e.g., methods, fields) that are only used by the source method. Decide whether they are about to be moved as well.
2. Check whether sub-features or parent features of the source feature exist that contain other declarations of the source method. If there are such features this may hinder to move the source method (e.g., if these features occur in variants without the target feature).
3. If the class that contains the source method has no role in the target feature create the class for the target feature.
4. Declare the method in the target feature (in the respective class).
5. Move the code from the source method to the target. Adjust it if necessary (e.g., references to fields/methods).
6. Decide whether to turn the source method into a delegating method or delete it at all. The latter is especially useful if you have only few or no references to the source method.
7. In case that you delete the source method, replace references to this (outdated) method with references to

the target method.

**Example:** As an example, we move a method from the optional feature *Undo* to the optional feature *Peak* of our Stack SPL. In Figure 6, we show the corresponding collaboration diagrams and code fragments that are affected by this refactoring. In particular, we move the method `undo()` from feature *Undo* to feature *Peak*. Although this may be questionable from a design point of view, with this example we are able to appropriately illustrate the mechanics of the refactoring. The pre-condition is fulfilled, since feature *Undo* requires feature *Peak* (cf. Figure 1). Hence, all variants that contain *Undo* contain *Peak* as well. While working through the mechanics, mentioned above, we come to the point where we have to care about moving local objects from *Undo* to *Peak*, namely the field `undoStore` (cf. Figure 6). Since we have to move this field as well, we apply *Move Field Between Features* refactoring. You find the respective explanation for this refactoring below. Afterwards, we can proceed by deleting the original method and replace the references of the method `undo()`. Note that we do not depict these last steps in Figure 6.

### Move Field Between Features

**Typical situation:** A field is used by another feature (or a certain role in it respectively) more than by the role on which it is defined.

**Recommended action:** Move the field from its current role to the role of the target feature.

**Pre-conditions:** The field must be available for all variants as before the refactoring.

#### Mechanics:

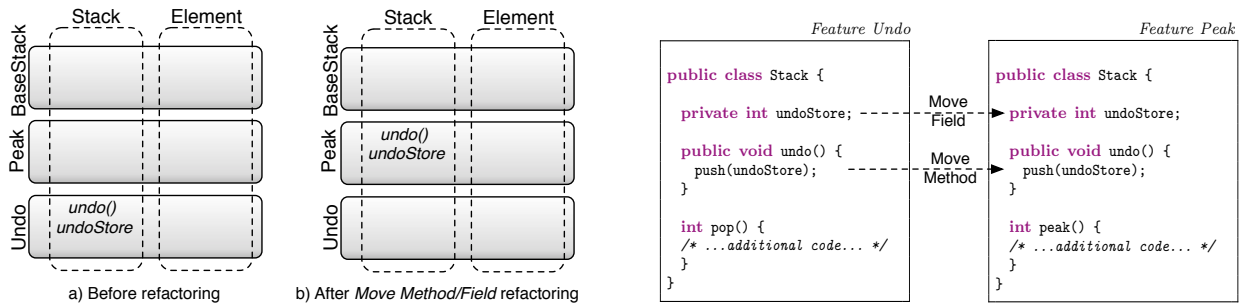
1. Create a field in the target feature (within the envisaged class) and provide getter/setter methods if necessary.
2. If the class that contains the source method has no role in the target feature create the class for the target feature.
3. Delete the field from the source feature (to say, the respective role).
4. Replace references to the source field with references to the respective accessor method on the target (e.g., the getter method).

**Example:** In our example in Figure 6, we have to move the field `undoStore` from feature *Undo* to feature *Peak*, because the method that uses the field is moved as well (using the *Move Method Between Features* refactoring). Therefore, we create the field in feature *Peak*. Since the field is not used outside its new role, we don't provide a getter method. Consequently, there are no references that we have to replace either.

## 5.1 Experiences.

In a recent study, we analyzed the occurrence of code clones in feature-oriented SPLs [20]. Based on the results, we exemplarily performed refactorings to remove code clones from an analyzed SPL called *TankWar*. In the following, we will share our experiences on applying variant-preserving refactoring to the *TankWar* SPL.

**The TankWar product line.** *TankWar* is a shoot 'em up game, developed as SPL by students of the university of Magdeburg. Furthermore, *TankWar* is of medium size (approx. 5000 SLOC) and consists of 38 features. The code clone analysis we conducted revealed that *TankWar* contains



**Figure 6:** On the left side, we depict the corresponding collaboration diagram before and after the *Move Method* and *Move Field* refactoring. On the right side, we show excerpts of the implementation of features *Undo* (before refactoring) and feature *Peak* (after refactoring).

a considerable amount of code clones ( $\sim 20\%$ ). Furthermore, we observed that a majority of these clones exist between alternative features, which inherently have a common parent. Additionally, most of these clones occur on method or constructor level. Hence, we selected this SPL to perform refactorings for code clone removal.

**Code clone removal using FOP refactoring.** We mainly applied the *Pull Up Method to Parent Feature* to remove clones. To this end, the first and maybe most important task was to check the defined pre-condition that determines whether we can perform a variant-preserving refactoring. The information we needed to check and fulfill this pre-conditions (regarding the features), we obtained from *FeatureIDE*<sup>2</sup>, an Eclipse plug-in for feature-oriented software development. FeatureIDE provides not only an editor to create feature models but also capabilities to formulate queries to a specific feature model to gather knowledge about feature relations and dependencies. The query is implemented as a method within FeatureIDE and can be expressed in natural language as *Given a Feature X, which features are in all variants that contain feature X?*. Hence, providing the features that contain the candidate methods (for refactoring) as input for the query, we obtain a list of features that are possible refactoring targets. Due to this capabilities, we could check the pre-conditions automatically by only providing the features that contain the candidates for refactoring.

However, the actual application of refactorings and checking the correctness of the SPL program thereafter, we had to perform manually. To this end, we performed the mechanisms, proposed for the respective refactoring, step-by-step. After each refactoring we ran a subset of all variants that were possibly affected by the refactoring. All of the tested variants exhibited the same behavior as before the refactoring. Hence, we assume that the refactorings are indeed variant-preserving. All in all, we were able to reduce the amount of clones to 12 % by applying FOP-specific, variant preserving refactorings.

## 6. DISCUSSION

In this section, we discuss the generalizability of our approach as well as possible automation of certain steps to improve the applicability of our approach.

**Generalizability.** While we focussed on compositional

SPL implementation techniques, specifically on FOP, in this paper, there is also the alternative of using annotation-based techniques for the implementation of SPLs. The latter approach implements product lines by annotating code fragments that belong to certain features in the overall code base. Then, a variant can be generated by removing annotated code fragments. In contrast to compositional approaches, the features are not modularized in annotative approaches. Most notably, the C preprocessor *cpp* is a form of such an annotative approach, commonly used with C/C++ programming language. To express variability, the respective code fragment is annotated with specific preprocessor directives (*#ifdefs*) that are evaluated before compilation of the actual program.

Regarding our refactoring approach, the question is whether we can apply it to the annotation-based approach as well. From our point of view the answer is twofold: First, given our definition for variant-preserving refactoring we argue that it holds for annotation-based approaches, specifically the *cpp*, as well. The reason is that we can also build a feature model for annotative SPLs to determine which combinations of features are valid and which are not. Since our definition is based on the valid combination of features, it can be applied to annotation-based SPLs as well.

However, applying the FOP-specific refactorings we propose in this paper to annotation-based SPLs may require additional or changed mechanics. For instance, in C programs, variability is also expressed using configuration files that determine whether certain source files may be compiled together or not. Since this is an additional variability mechanism, it may influence the actual refactoring process and thus the preservation of variants. Furthermore, other preprocessor directives for file inclusion (*#include*) and macros (*#define*) exist that complicate refactorings.

We conclude, that our proposed refactorings are only partly generalizable and that we more work has to be done to fully generalize variant-preserving refactoring across SPL implementation techniques. Nevertheless, we argue that our approach can be used as a starting point.

**Automating FOP-specific refactorings.** An important aspect for the applicability of the proposed refactorings is to what extent the refactoring process can be automated. For the refactorings, presented in this paper, we identified three steps that would benefit from such automation: detection possible refactoring candidates, checking the pre-conditions, and the actual change of the source code. In

<sup>2</sup><http://www.fosd.net/featureide/>

the following, we mainly focus on step one and three, because we addressed the second step already in Section 5.1.

The first step is of particular interest for the *Pull Up* refactorings, because it is impossible to identify identical or similar code fragments manually. However, different approaches exist that can be used to guide the user to such code fragments. First of all, *clone detection*, that is, the detection of replicated code fragments, can be used to determine syntactically identical or similar code fragments. Second, with recent approaches it is even possible to detect semantically identical (but syntactically different) code fragments [13]. However, for the latter kind of similar code fragments, it is still open how to automate the syntactical unification of such code fragments.

Automating the actual refactoring, especially the propagation of source code changes, requires an appropriate representation of the underlying source code such as an *abstract syntax tree (AST)*. Since generating such an AST for each variant of an SPL is impossible, an AST for the whole SPL, which contains all information on variability, is required. However, we are not aware of any approach that can generate such an AST for feature-oriented programming and thus automating the actual refactoring is currently not possible. Nevertheless, recent approaches for annotation-based approaches provide a variability-aware AST that could be used for refactoring (especially for automating the corresponding step), which is one of our future tasks [14].

## 7. RELATED WORK

A variety of work has been done on refactoring software product lines. In this section, we account for the most important work and figure out how it differs from our approach.

**Refactoring of feature models.** Similar to us, Alves et al. identified shortcomings in traditional refactorings when applied to SPLs due to missing support of configurability [1]. As a solution, they propose a set of feature model refactorings to improve the feature model of an SPL and, beyond that, to merge multiple programs into one SPL. Likewise, Thüm et al. present an approach that supports reasoning about feature model edits [23]. To this end, they classify changes to a feature model into refactoring, generalization, specialization, and arbitrary edits. Additionally, they formalize their approach using propositional formulas. All the aforementioned approaches have in common that they focus on refactoring of feature models. In contrast, we focus on refactoring the underlying source code of an SPL with our approach. Although we take feature models into account, we use their information for evaluating pre-conditions only. Finally, we focus on feature-oriented SPLs while the work mentioned above is independent of the SPL implementation technique.

**Refactoring of feature-oriented SPLs.** Liu et al. propose a theory for feature-oriented refactoring that relates code to algebraic refactoring [17]. With their theory, they focus on decomposing programs into features. Specifically, they address the problem, that features may have different implementations in different variants of the SPL. In contrast to our work, where we address the refactoring of source code in existing feature-oriented SPLs, they address how to refactor an object-oriented legacy application into features. Kuhlemann et al. propose *refactoring feature modules (RFM)* for refactoring in feature-oriented SPLs [15]. The core idea of RFM is to encapsulate information that is necessary for per-

forming an object-oriented refactoring in a separate feature module and make it explicit in the corresponding feature model. Additionally, Kuhlemann et al. formalized their approach by means of an algebraic model [16]. In contrast to their approach, where refactorings are features in a concrete SPL, we aim at a more interactive approach. In particular, we aim at integrating the proposed refactoring into an IDE such as Eclipse so that they can be used on any SPL, depending on the implementation technique only. Finally, Borba et al. recently proposed a theory that covers both, refactoring of feature models and source code, based on product line refinement [8]. With their general, language-independent formalization they provide SPL properties (in terms of refinement) that support evolution of software product lines. In contrast to their work, we specifically focus on FOP and introduce first refactorings in a more practical way by describing rather the actual mechanisms than the underlying theory. However, integrating our ideas with their theory could be beneficial and is left for future work.

## 8. CONCLUSION

Refactoring of source code is a pivotal task regarding the quality and longevity of source code. While it is well-understood for conventional, stand-alone systems, only few work exists for SPLs. In this paper, we presented how existing, object-oriented refactorings can be extended and applied to feature-oriented SPLs, while preserving all variants of the SPL. To this end, we introduced the notion of variant-preserving refactoring and proposed concrete refactorings for feature-oriented SPLs. Additionally, we shared first experiences when applying such refactorings for the removal of code clones. Finally, we discussed the generalizability of our approach for annotative SPLs.

In future, we intend to extend this work by more refactorings for feature-oriented and annotative SPLs. Furthermore, we intend to provide tool support for these refactorings so that they can be applied and pre-conditions can be checked automatically. Additionally, merging our ideas with existing theories and formalizations for feature-oriented refactoring is an important future task, because it allows us to prove the variant-preserving nature of refactorings. Finally, extending our refactoring approach to annotative SPLs is a challenging but promising task that we intend to pursue in the future.

## 9. ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their detailed and valuable comments on the initially submitted version of this paper.

## 10. REFERENCES

- [1] ALVES, V., GHEYI, R., MASSONI, T., KULESZA, U., BORBA, P., AND LUCENA, C. Refactoring product lines. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering* (2006), pp. 201–210.
- [2] APEL, S., AND KÄSTNER, C. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (July 2009), 49–84. (column).
- [3] APEL, S., KÄSTNER, C., AND LENGAUER, C. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int. Conf. on Software*

- Engineering* (2009), IEEE Computer Society, pp. 221–231.
- [4] APEL, S., KOLESNIKOV, S., LIEBIG, J., KÄSTNER, C., KUHLEMANN, M., AND LEICH, T. Access control in feature-oriented programming. *Science of Computer Programming (Special Issue on Feature-Oriented Software Development)* (2010), –. to appear; submitted 24 Mar 2010, accepted 29 Jul 2010.
- [5] APEL, S., LEICH, T., ROSENMÜLLER, M., AND SAAKE, G. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering* (2005), Springer-Verlag, pp. 125–140.
- [6] BATORY, D., SARVELA, J., AND RAUSCHMAYER, A. Scaling Step-Wise Refinement. *IEEE Trans. Soft. Eng.* 30, 6 (2004), 355–371.
- [7] BAXTER, I., YAHIN, A., MOURA, L., SANT’ANNA, M., AND BIER, L. Clone detection using abstract syntax trees. In *Proc. of the Int. Conf. on Software Maintenance* (1998), IEEE Computer Society, pp. 368–379.
- [8] BORBA, P., TEIXEIRA, L., AND GHEYI, R. A theory of software product line refinement. In *Proc. Int. Colloquium on Theoretic Aspects of Computing (ICTAC)* (2010), Springer-Verlag, pp. 15–43.
- [9] BRACHA, G., AND COOK, W. Mixin-based inheritance. In *Proc. of the Eur. Conf. on Object-Oriented Programming* (1990), ACM Press, pp. 303–311.
- [10] CLEMENTS, P., AND NORTHROP, L. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2006.
- [11] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [12] GÖDE, N. Clone removal: fact or fiction? In *Proc. Int. Workshop on Software Clones (IWSC)* (2010), ACM Press, pp. 33–40.
- [13] JUERGENS, E., DEISSENBOECK, F., AND HUMMEL, B. Code similarities beyond copy & paste. In *Proc. of the Eur. Conf. on Software Maintenance and Reengineering* (2010), pp. 78–87.
- [14] KÄSTNER, C., GIARRUSSO, P., RENDEL, T., ERDWEG, S., OSTERMANN, K., AND BERGER, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2011), pp. 805–824.
- [15] KUHLEMANN, M., BATORY, D., AND APEL, S. Refactoring feature modules. In *Proc. of the Int. Conf. on Software Reuse* (2009), Springer-Verlag, pp. 106–115.
- [16] KUHLEMANN, M., KÄSTNER, C., APEL, S., AND SAAKE, G. An algebra for refactoring and feature-oriented programming. Tech. Rep. FIN-006-2011, Otto-von-Guericke University Magdeburg, 2011.
- [17] LIU, J., BATORY, D., AND LENGAUER, C. Feature oriented refactoring of legacy applications. In *Proc. Int. Conf. on Software Engineering* (2006), ACM Press, pp. 112–121.
- [18] MONTEIRO, M. P., AND FERNANDES, J. M. Towards a catalog of aspect-oriented refactorings. In *Proc. Int. Conf. on Aspect-Oriented Software Development* (2005), pp. 111–122.
- [19] OBDYKE, W. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [20] SCHULZE, S., APEL, S., AND KÄSTNER, C. Code Clones in Feature-Oriented Software Product Lines. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering* (2010), ACM Press, pp. 103–112.
- [21] SIEGMUND, N., ROSENMÜLLER, M., KÄSTNER, C., GIARRUSSO, P., APEL, S., AND KOLESNIKOV, S. Scalable prediction of non-functional properties in software product lines. In *Proc. of the Int. Software Product Line Conf.* (2011), IEEE Computer Society, pp. 160–169.
- [22] SMARAGDAKIS, Y., AND BATORY, D. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), 215–255.
- [23] THÜM, T., BATORY, D., AND KÄSTNER, C. Reasoning about Edits to Feature Models. In *Proc. Int. Conf. on Software Engineering* (2009), IEEE Computer Society, pp. 254–264.
- [24] VANHILST, M., AND NOTKIN, D. Using role components in implement collaboration-based designs. In *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (1996), ACM Press, pp. 359–369.