Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line

Wolfram Fenske,* Jens Meinicke,*,† Sandro Schulze,* Steffen Schulze,* Gunter Saake*
*University of Magdeburg, Germany
†Carnegie Mellon University, USA
{wfenske,meinicke,sanschul,saake}@ovgu.de

***Abstract*—A common and simple way to create custom product variants is to copy and adapt existing software (a. k. a. the *clone-and-own* approach). Clone-and-own promises low initial costs for creating a new variant as existing code is easily reused. However, clone-and-own also comes with major drawbacks for maintenance and evolution since changes, such as bug fixes, need to be synchronized among several product variants. *Software product lines (SPLs)* provide solutions to these problems because commonalities are implemented only once. Thus, in an SPL, changes also need to be applied only once. Therefore, the *migration* of cloned product variants to an SPL would be beneficial. The main tasks of migration are the identification and extraction of commonalities from existing products. However, these tasks are challenging and currently not well-supported. In this paper, we propose a step-wise and semi-automated process to migrate cloned product variants to a feature-oriented SPL. Our process relies on clone detection to identify code that is common to multiple variants and novel, variant-preserving refactorings to extract such common code. We evaluated our approach on five cloned product variants, reducing code clones by 25 %. Moreover, we provide qualitative insights into possible limitations and potentials for removing even more redundant code. We argue that our approach can effectively decrease synchronization effort compared to clone-and-own development and thus reduce the long-term costs for maintenance and evolution.**

I. Introduction

Clone-and-own is a quick way to create customized variants of a software product. The basic idea is to copy an existing, well-tested product and adapt it to a new set of requirements [11], [39], [40]. A major drawback of clone-and-own is that it leads to a high proportion of *code clones* [37]. Changes (e. g. bug fixes) to these clones must be carefully synchronized to keep all variants consistent. Clone-and-own is often implemented using the branching and merging capabilities of a version control system [9], [12], [44], [45], [50]. However, these systems lack the necessary support for mapping changes to features or tracking which variants implement which features – a shortcoming that research has only recently started to address (e. g., [3], [33], [34]). Thus, practitioners still face the time-consuming and error-prone tasks of identifying changes and merging them into the proper variants. This makes long-term maintenance of cloned products expensive.

In a *software product line (SPL)*, commonalities and differences of a set of product variants are managed in terms of *features* [4]. In this context, a feature represents an increment in functionality that is important to some stakeholder. Since features in an SPL are implemented only once, but shared amongst many variants, changes to the implementation only have to be performed once as well. Consequently, the effort for synchronizing product variants in an SPL is minimal.

Cloned product variants can be *migrated* to an SPL in order to benefit from systematic reuse of source code and ease of maintenance. However, migration takes considerable time and effort and may ultimately fail, which makes it a risky and unforeseeable process [8], [23]. Some migration approaches have been proposed [1], [38], [51], [52], but they focus mostly on models and lack specifics on how to migrate the source code. Furthermore, it is not clear how to combine these migration approaches with other development activities, for instance, to release important bug fixes while the migration process is ongoing.

This paper is an initial study of a novel approach to migrate multiple cloned product variants into an SPL using *feature-oriented programming (FOP)* [7], [35] as a variability mechanism. We demonstrate its feasibility on a small but realistic case study. Our approach relies on code clone detection to identify code that is common to multiple variants and on *variant-preserving refactoring* [43] to consolidate such clones. As an important concept, we propose *preparatory refactorings,* that is, refactorings that align variant-specific divergencies that would otherwise prevent the consolidation of code clones. Unlike previous "big-bang" approaches (e. g., [1], [52]), our approach favors small, incremental and easy-to-understand steps. In particular, we do not propose full automation but rather support developers in automating the tedious and error-prone tasks, leaving the overall design decisions in their hands. In summary, we contribute the following:

- A novel concept to migrate cloned product variants to a feature-oriented SPL in a step-wise manner;
- Refactorings to reliably consolidate code that has been cloned across multiple product variants;
- A concept and a refactoring to compensate for variant-specific differences that would prevent the extraction of cloned code;
- Tool support that integrates clone detection and our refactorings into FeatureIDE[1] [46].

We evaluated our approach on a case study of five Android programs with a total of 21k lines of code. This case study

[1] http://fosd.net/fide

978-1-5090-5501-2/17 © 2017 IEEE

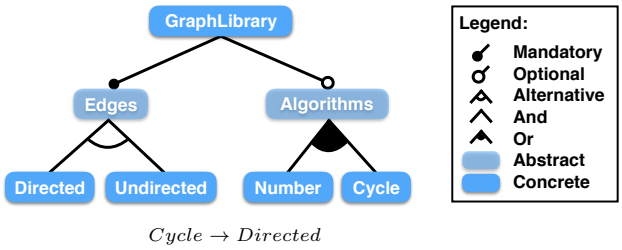


Figure 1. Feature model of a graph product line.

shows that our approach is feasible and provides directions for future enhancements.

II. Background

In this paper, we use feature models for representing the variability of an SPL on the requirements level, and feature-oriented programming (FOP) on the implementation level. Our refactorings have to take both levels into account in order to preserve the behavior of all products of the SPL. During refactoring, we focus on source code that is similar in the variants being migrated. We use code-clone detection to identify such code. We provide the necessary background on these topics in this section.

A. Variability Modeling

Commonalities and differences of the products within an SPL are expressed as features. Features are arranged in a *feature model* to expresses which feature combinations form valid products [16]. A feature model is a tree structure in which nodes represent features and edges describe how a child feature is related to its parent. As an example, we show the feature model of an SPL for graph algorithms in Figure 1. The root, *GraphLibrary*, is a *concrete* feature since it implements basic functionality of all instances of this SPL. By contrast, its children, *Edges* and *Algorithms*, are *abstract*, i. e., they only help organization but contain no implementation of their own [47]. *Edges* is *mandatory* (every graph needs edges), whereas *Algorithms* is *optional* (not all instances of this SPL contain algorithms). Children can be grouped in several ways. For instance, edges can either be directed or undirected, not both, not neither. Hence, *Directed* and *Undirected* form an *alternative*. More complex relationships can be expressed by adding formulae in propositional logic, called *cross-tree constraints*. The example model contains one such constraint, stating that the *Cycle* algorithm requires *Directed* edges.

A selection of features is called a *configuration*. The (potentially large) set of configurations that conform to the feature model and its cross-tree constraints is called the set of *valid configurations*. The actual products that are built from a given SPL correspond to a subset of all valid configurations. We call them the *existing configurations*.

B. Variability Implementation

Similarly to single-system development, features in an SPL are primarily implemented in a host language. In our case, the

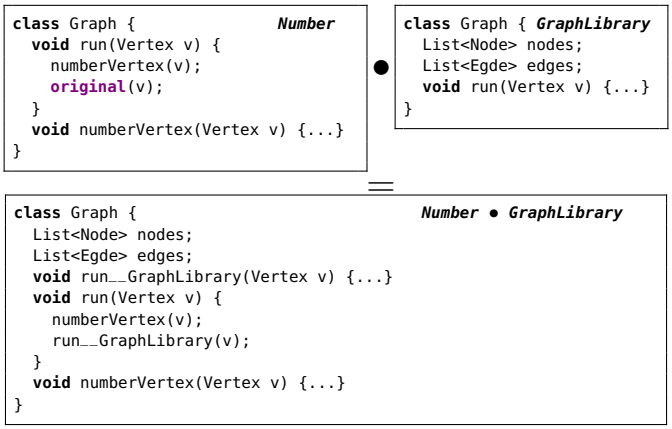
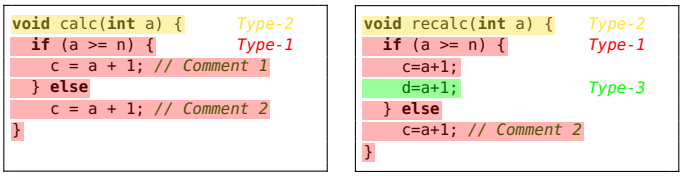


Figure 2. Composition of features *Number* and *GraphLibrary*, producing *Number* • *GraphLibrary*.

host language is Java. To achieve variability, this host language is combined with a variability mechanism. We use FOP, a composition-based mechanism [7], [35]. Composition-based mechanisms aim for physical separation of concerns [18]. All code (and non-code) artifacts that implement a particular feature are modularized in one cohesive unit, thus separated from the implementation of other features. In FOP, this unit is called a *feature module*.

Products are instantiated by *composing* the feature modules that correspond to a product's configuration. We use FeatureHouse as a composer [5]. In Figure 2, we show how the features *GraphLibrary* and *Number* of the graph SPL from Figure 1 are composed. In this example, two features, *GraphLibrary* and *Number*, define a class `Graph`. The composition result is shown in the bottom half of the figure. Except for `run` and `run__GraphLibrary`, the composition is simply the union of all fields and methods of the constituent class definitions. The `run` methods are composed differently because the definition in *Number* uses the keyword `original`. This keyword instructs FeatureHouse to reference a method with same signature (i. e. the same name an parameter types) from a feature that is composed earlier. In our example, this is the `run` method in *GraphLibrary*. Thus, FeatureHouse copies *GraphLibrary*'s defintion into the composition result and renames it to `run__GraphLibrary`, copies *Number*'s `run`, and replaces `original` with a call to the renamed method.

The *composition order* defines how the composition of feature modules proceeds. The standard is to compose a child feature after its parent, and the left subtree before the right one. If needed, custom orders can be specified. The feature module that first defines a class is said to *introduce* the class. Feature modules composed later may *refine* this definition. Apart from adding fields and methods, or refining methods via `original`, as shown in the example, refinements can also override (completely replace) previous method definitions. Since our migration process involves moving code between features, we must take theses effects into account. For instance, let there be two features, f_1 and f_2, both defining a method `m`



(a) Original code **(b) Code clone**

Figure 3. Examples of code clones of Type-1 to Type-3. The red areas form Type-1 clones. Red and yellow together form a Type-2 clone. Red, yellow and green together form a Type-3 clone.

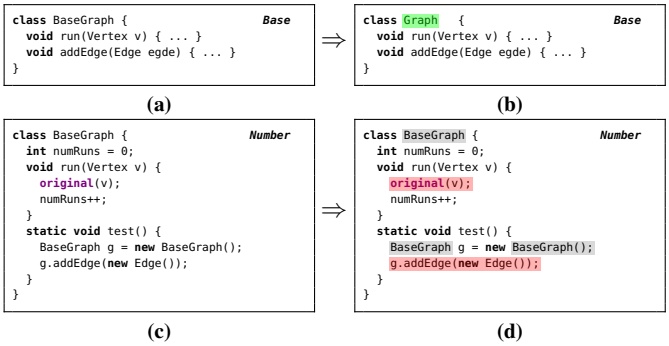


Figure 4. Example of an OOP RENAME refactoring producing wrong results when applied to FOP code. Only the code in feature *Base* is renamed correctly (b), but the code in *Number* is not, leading to dangling references (red highlights in (d)).

with the same *signature*. If f_2's `m` does not contain an `original` call, then f_2's definition will override the one from f_1 in the composition $f_2 \bullet f_1$.

C. Code Clones

A code clone is a piece of source code that occurs in the same or similar form in multiple locations [37]. Usually, a clone results from copying a piece of code from one location to another, possibly customizing the copy afterwards [17].

Code clones differ in their degree of textual similarity, which impacts the refactorings required to consolidate them. According to similarity, clones are categorized into four types, namely, *Type-1* to *Type-4* clones [37]. In Figure 3, we show examples of clone Type-1 to Type-3, which are most relevant to our discussion. The original code fragment is depicted in Figure 3 (a), whereas the cloned fragment is shown in (b). Type-1 clones, highlighted in red in Figure 3, are exact copies of each other, except for changes in whitespace or comments. Type-2 clones subsume all Type-1 clones, but also allow for renaming (for instance, function or type names may differ). In Figure 3, the red and yellow areas together form a Type-2 clone. Beyond simple renaming, even statements may have been added, deleted or changed with respect to the original code, resulting in a Type-3 clone, encompassing the red, yellow, and green areas in Figure 3. Finally, Type-4 clones, also called *functional clones*, implement similar functionality but share little or no textual similarity [37].

There are different approaches (based on, for instance, tokens, trees or graphs) and tools to detect code clones automatically [37]. Generally, Type-1 and 2 clones are well-supported, but few tools detect Type-3 or 4 clones.

III. Problem Statement

In previous work, we defined *variant-preserving migration* as the transformation of a family of related software products to an SPL in such a way that for each of the original products, it is possible to create an instance of the SPL that has the same external behavior [13]. Existing approaches to variant-preserving migration focus on aspects such as requirements, development models, or feature location (e. g. Alves et al. [1], Xue et al. [51]–[53]). By contrast, this paper focuses on the implementation-level aspects, i. e., on finding commonalities and differences in the source code and increasing the amount of systematic source code reuse. Specifically, we propose an approach to migrate product variants written in Java to an SPL that uses FOP for implementing variability.

Due to the clone-and-own origin of the product variants that we migrate, we expect a large amount of code sharing (a. k. a. *code clones*) among them. Thus, we consider migration primarily as a code-clone problem: If we can reduce the amount of code clones across product variants, thereby increasing the degree of systematic reuse, we will also decrease the maintenance overhead caused by change synchronization. Consequently, the main building blocks of our approach are code clone detection and *variant-preserving refactoring* [43] for identifying and consolidating commonalities between product variants.

Next, we explain the variant-preserving refactorings we propose for consolidation. Afterwards, we describe how we integrate these refactorings with code clone detection into a variant-preserving migration process.

IV. Variant-Preserving Refactorings

A refactoring is a "change to the internal structure of a program without changing its external behavior" [14], [32], [36]. Several catalogs of refactorings have been proposed for object-oriented programming (OOP) or aspect-oriented programming (AOP) (e. g., [14], [19], [31]). However, these refactorings do not take variability into account and, thus, are generally not behavior-preserving when applied to SPLs. As an example, we show in Figure 4 the result of applying the Eclipse RENAME refactoring for Java to a small graph product line with two features, *Base* and *Number*. While the goal was to rename class `BaseGraph` to `Graph`, Eclipse's (OOP) RENAME refactoring only renamed the code in the feature module of *Base* (see Figure 4, top row), but not in *Number* (bottom row). As a result, feature *Number* contains several errors, highlighted in gray and red in Figure 4 (d). In particular, the class in *Number* is still named `BaseGraph` (instead of `Graph`) and thus no longer refines the class in *Base*. Even worse, the calls to `original` and `addEdge` are turned into dangling references (highlighted in red), which will cause compilation errors in variants that contain the *Number* feature.

In previous work, we proposed *variant-preserving refactoring* in order to extend the notion of behavior-preservation to SPLs [43]. In essence, a variant-preserving refactoring ensures that all potential products within an SPL remain compilable and keep their previous behavior. Moreover, all configurations that were valid before the refactoring must remain valid afterwards. For the example in Figure 4, a variant-preserving RENAME would also change the class name in feature *Number* from `BaseGraph` to `Graph` (highlighted in gray in Figure 4 (d)) and thus eliminate the dangling references (highlighted in red).

Next, we describe two variant-preserving refactorings for FOP, PULL UP TO COMMON FEATURE and RENAME, thus generalizing and improving upon our previous work [41], [43]. Compared to their OOP counterparts, PULL UP and RENAME, we contribute the following:

1) We extend the preconditions and mechanics of both refactorings so they become variant-preserving.
2) For the PULL UP TO COMMON FEATURE refactoring, we provide an algorithm to identify the "common feature", that is, the feature into which the respective code fragment can be moved in a variant-preserving manner. If no suitable feature exists, the algorithm tries to create a new one.

Although we use these refactorings to eliminate code duplication, we anticipate they will also be useful for general FOP programming, outside of a migration context.

A. Pull Up To Common Feature

The PULL UP refactoring, described by Fowler et al. (see chapter 11 in [14]), is used to move identical definitions of class members (i. e., fields, methods and constructor bodies) from a set of subclasses into a common superclass. As PULL UP replaces multiple replicated definitions with a single definition, it is an effective means to remove code clones from a single software product. While removing clones is our goal as well, we want to remove them from multiple software products. To this end, we propose an extension of PULL UP for FOP, called PULL UP TO COMMON FEATURE. The basic idea of our refactoring is to move identical definitions of class members from several *source features* into a single, common *target feature,* which is located higher up in the feature hierarchy than the source features. Hence, instead of moving code within the class hierarchy, as OOP PULL UP does, we move the code within the feature hierarchy.

We illustrate the application of PULL UP TO COMMON FEATURE by means of the example in Figure 5. The example is a product line consisting of the features *Bar* and *Baz*, whose parent feature is *Common*. *Common* introduces class `Foo`, which is refined by both, *Bar* and *Baz*. More importantly, *Bar* and *Baz* contain a Type-1 inter-feature code clone, the method `answer` (highlighted in gray in (c) and (e)). By applying PULL UP TO COMMON FEATURE, the definitions of `answer` in *Bar* and *Baz* are replaced with a single definition in feature *Common* (see (d), (f), and the green highlight in (b)) and thus, the code clone is removed. As a result, common functionality

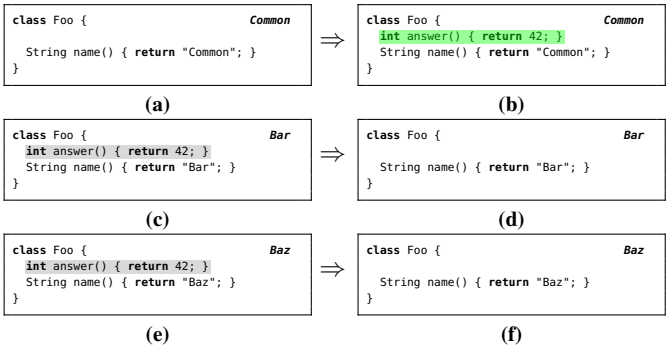


Figure 5. Application of our PULL UP TO COMMON FEATURE refactoring to move the common definition of method `answer` in class `Foo` from features *Bar* and *Baz* (gray highlight in (c) and (e)) to the common feature *Common* (green highlight in (b)).

of product variants containing features *Bar* or *Baz* has been consolidated, while preserving the behavior of these variants.

Generally, PULL UP is applicable to different elements, such as fields, methods, and constructors. Since all of them are very similar, we focus on PULL UP METHOD TO COMMON FEATURE as a representative, and explain this refactoring in terms of arguments, preconditions and mechanics. The arguments are supplied by the developer or some analysis tool and specify, for instance, which method definitions to pull up. Preconditions are properties that must hold in order for the program transformation to be behavior-preserving. Finally, the mechanics specify the program transformation itself.

Arguments:

1) n source features $F_s = \{f_{s_1}, f_{s_2}, \ldots, f_{s_n}\}$ with $n > 1$,
2) A target feature f_t,
3) A class name c,
4) A method signature m,
5) A selection $i, 1 \leq i \leq n$, denoting the definition m_i of method m in class c in feature f_{s_i}, where $f_{s_i} \in F_s$.

Preconditions:

1) Each $f_s \in F_s$ must contain a class c that defines a method with signature m.
   We suggest pulling up definitions only if they constitute Type-1 clones as these are currently the only ones for which we can ensure fully automatically that the refactoring is behavior-preserving. However, developers may override this suggestion, e.g., to pull up method definitions that constitute higher-level clones.
2) All fields and methods referenced in m_i must be defined in f_t or one of the features implied by f_t.
   This precondition ensures that the selected method definition, m_i, will not reference any fields or methods that are undefined in the target feature. If such references exist, PULL UP TO COMMON FEATURE *must be applied to the respective fields and methods first.*
3) Target feature f_t must be a concrete feature.
   Only concrete features can contain code.
4) Any valid configuration that contains an $f_s \in F_s$ must also contain f_t and vice versa.

This precondition enforces that all products containing one of the source features will have access to the new location of the code. Since this additionally requires that at least one source feature is present whenever the target feature is present, we prevent the pulled up method definition from shadowing other definitions, i.e., from features composed before the target feature.

5) If a class c already exists in the target feature f_t, it must not define a method with signature m.
6) With F being the set of all features, no valid configuration containing an $f_s \in F_s$ may contain an $f_d \in F \setminus F_s$ that fulfills the following criteria:
   a) f_d contains a class c that defines a method with signature m,
   b) f_d is composed after f_t but before f_s.

   This precondition prevents definition m_i from being overwritten or refined by a definition from another feature before the source feature is composed.

Mechanics:

1) Create class c in f_t (unless it already exists) and create a new method with signature m in that class.
2) Copy the selected method definition m_i from s_i into the newly created method.
3) Delete the old definitions of m from all classes c in all source features $f_s \in F_s$.

Determining the Target Feature: A critical point of PULL UP TO COMMON FEATURE is how to determine a suitable target feature. In general, the target feature must be part of any valid configuration that contains one of the source features, and it must be composed before the source features. In case all source features have a common parent feature, the parent might serve as the target feature. Otherwise, a new target feature can usually be created. In Algorithm 1, we show two functions, FINDTARGETFEATURES and CREATETARGETFEATURE, which implement these tasks. Specifically, the first function will identify suitable target features, and if none exist, the second one tries to create a new target feature. Both functions take as inputs the set of source features F_s, the name of the defining class c, the method signature m, and the feature model fm, given as a propositional formula. Additionally, CREATETARGETFEATURE receives the set of existing configurations as $\mathcal{C}$. In a migration context, these are the configurations of the products being migrated.

The first main function, FINDTARGETFEATURES, starts by identifying the features that are implied by all source features and which, in turn, also imply that at least one of the source features is selected (Line 2). This implements precondition 4 of PULL UP TO COMMON FEATURE. In Lines 3 and 4, this set of features is reduced to features that are concrete (precondition 3), do not already define method m (precondition 5), and are composed before the source features. (Assume that $order$ returns the composition order of a feature.) Finally, REMOVESHADOWED is called on the set of remaining features (Line 5). This helper function implements precondition 6 by excluding potential target features that could be composed with another feature that contains a conflicting definition of method m (Line 27).

Algorithm 1 Find or Create a Target Feature for PULL UP TO COMMON FEATURE

```
1: function FINDTARGETFEATURES(F_s, c, m, fm)
2:     F_b ← {f ∈ features(fm) | implies(fm, f ⇔ ⋁_{f_s∈F_s} f_s)}
3:     o_min ← min({order(f_s) | f_s ∈ F_s})
4:     F_c ← {f ∈ F_b | order(f) < o_min ∧ concrete(f)
                      ∧ ¬defines(f, c, m)}
5:     return REMOVESHADOWED(F_c, F_s, c, m, fm)
6: end function

7: function CREATETARGETFEATURE(F_s, c, m, fm, C)
8:     F_i ← {f ∈ features(fm) | implies(fm, f ⇐ ⋁_{f_s∈F_s} f_s}
9:     o_min ← min({order(f_s) | f_s ∈ F_s})
10:    F_c ← {f ∈ F_i | order(f) < o_min}
11:    F_p ← REMOVESHADOWED(F_c, F_s, c, m, fm)
12:    if F_p = ∅ then
13:        ERROR("Too many conflicting definitions.")
14:    end if
15:    f_p ← CHOSECLOSEST(F_p, F_s, fm)
16:    f_t ← ADDCHILD(f_p, CONCRETE, OPTIONAL)
17:    ADDCTC(fm, f_t ⇔ ⋁_{f_s∈F_s} f_s)
18:    for C ∈ {C ∈ C | C ∩ F_s ≠ ∅} do
19:        C ← C ∪ {f_t}
20:    end for
21:    return f_t
22: end function

23: function REMOVESHADOWED(F_c, F_s, c, m, fm)
24:    F_d ← {f ∈ features(fm) \ F_s | defines(f, c, m)}
25:    F_r ← ∅
26:    for f_c ∈ F_c do
27:        if ∀f_s ∈ F_s, ∄f_d ∈ F_d : satisifable(fm ∧ f_c ∧ f_d ∧ f_s)
               ∧ order(f_c) < order(f_d) < order(f_s) then
28:            F_r ← F_r ∪ {f_c}
29:        end if
30:    end for
31:    return F_r
32: end function
```

The second main function, CREATETARGETFEATURE, first identifies parent features for the target feature f_t, which is about to be created (Lines 8–10). Note that these parents must also fulfill precondition 6 (Line 11) since conflicting method definitions would also affect f_t. If no suitable parent exists, the function aborts (Lines 12–14). Otherwise, a parent feature is chosen by CHOSECLOSEST (not shown). This helper function could be a metric or an interactive function that lets the developer decide which parent feature is most appropriate. Next, the target feature f_t is created as a concrete, optional child of the chosen parent feature (Line 16). To fulfill precondition 4, a cross-tree constraint is added stating that f_t must be selected if and only if one of the source features F_s is selected (Line 17). Finally, after updating all existing configurations so that they conform to this new constraint (Lines 18–20), the new target feature is returned.

```
class Circle {                                  s1
  int x, y, radius;
  int getX() { return x; }
  int getY() { return y; }
  int getRadius() { return radius; }
}
```

(a)

```
class Circle {                                  s2
  int centerX, centerY, diameter;
  int getCenterX() { return centerX; }
  int getCenterY() { return centerY; }
  int getDiameter() { return diameter; }
}
```

(b)

Figure 6. The code that models the circle's center in features s_1 and s_2 constitutes a Type-2 clone, but naming differences (highlighted in gray) prevent the consolidation using PULL UP TO COMMON FEATURE. By contrast, `getDiameter` and `getRadius` (highlighted in red), should not be consolidated, despite their similarity.

B. Rename

Code clone removal is sometimes impossible due to minor differences. For instance, if two methods are identical except for different names (as shown in Figure 6, gray highlights), PULL UP TO COMMON FEATURE is not applicable. However, such differences can be eliminated by means of *preparatory refactorings*. To this end, we propose a RENAME refactoring that takes variability into account. We use RENAME to convert Type-2 clones into Type-1 clones and, thus, make them amenable to subsequent consolidation via PULL UP TO COMMON FEATURE.

Similarly to PULL UP TO COMMON FEATURE, our RENAME refactoring can be applied to different elements, such as classes and interfaces, methods (static and instance methods), fields (static and instance fields), parameters and local variables, and for constructor parameters. We describe RENAME INSTANCE METHOD as a representative.

Arguments:

1) Old method signature m_o,
2) The class, c, that defines m_o,
3) The feature, f, containing the definition of c,
4) New method name n.

Auxiliary Definitions: We introduce these definitions to describe preconditions and mechanics more concisely.

1) Let m_n be the new method signature. It is constructed from m_o by replacing the old method name with the new one, n.
2) Let D_{m_o} be the set of classes containing c, as well as all classes that define methods that override or are overridden by c's definition of m_o. If m_o is private in c, D_{m_o} contains only c as private methods cannot be overridden in Java. Otherwise (m_o is non-private), D_{m_o} contains c, as well as all sub- and superclasses of c that contain a non-private definition of m_o.

Preconditions:

1) The introductions and refinements of the classes in D_{m_o} must not define a method with signature m_n.
   This precondition prevents RENAME *from producing duplicate definitions in the classes that define a method with signature* m_o.
2) If m_o is non-private in c, then for all classes d in D_{m_o} it must hold that there is no introduction or refinement of a subclass of d that defines a method m_n with a lower visibility than that of m_o in d.
   Overriding methods in Java must not reduce the visibility of superclass methods. This precondition prevents RENAME *from breaking that rule.*
3) For all classes d in D_{m_o} it must hold that there is no introduction or refinement of a superclass of d that defines a non-private method with signature m_n with a greater visibility than that of m_o in d.
   This precondition is also related to Java's visibility rules. Specifically, it prevents renamed definitions of m_o *from restricting the visibility of preexisting definitions of* m_n.

Mechanics:

1) Find all references to method m_o.
2) In all introductions and refinements of classes in D_{m_o} that contain a definition of m_o, create a new method with signature m_n and copy the contents of m_o into this new method.
3) Update the collected references to point to m_n.
4) Remove the old definitions of m_o.

Note that our preconditions are rather liberal, thus potentially allowing renamings that are not variant-preserving. For instance, coming back to Figure 6, our preconditions would allow us to rename `getDiameter` to `getRadius` (or even to `getX`!), which is not a sensible change (see red highlights). We could prevent this with more restrictive preconditions, but only at the cost of precluding many useful applications, such as renaming `getCenterX` to `getX` and `getCenterY` to `getY` (gray highlights).

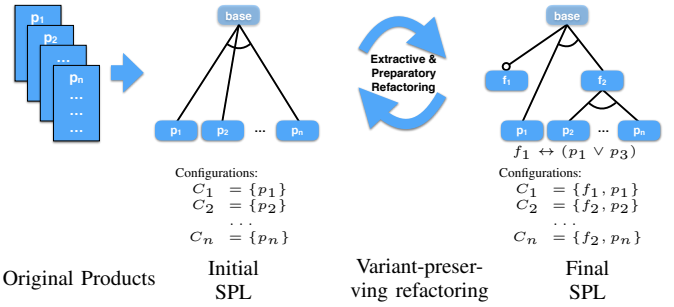


Figure 7. Feature-model perspective of our migration process, showing the migration of n cloned product variants to a feature-oriented SPL via extractive and preparatory refactoring.

V. FEATURE-ORIENTED MIGRATION

We now describe how we combined the refactorings explained in the previous section with code clone detection to form a process for the variant-preserving migration of cloned product variants to an FOP SPL.

Initializing the SPL: We depict our migration process in Figure 7. Starting with n product variants, $p_1, p_2, \ldots, p_n$, the first step is to create a trivial initial SPL whose feature model contains only one alternative. This alternative consists of n features, $p_1, p_2, \ldots, p_n$, one for each of the original products. Moreover, the source code of each original product variant is moved into the corresponding feature module. Together with the feature model, n configurations, $C_1, C_2, \ldots, C_n$, are created, each with exactly one of the features $p_1, p_2, \ldots, p_n$

selected, while all others are deselected. Hence, it is possible to recreate the original product p_1 by choosing configuration C_1, whereas p_2 is recreated by choosing C_2, and so on.

Code Clone Extraction: While the first step does not yield any improvement in terms of reuse, it forms the basis for the subsequent, iterative refinement process that constitutes the core of our migration approach. To this end, we propose two steps, *code clone extraction* and *preparatory refactoring*, to reduce the amount of code clones in the features $p_1, p_2, \ldots, p_n$. To identify code clones, we use clone detection, as we will outline shortly. If possible, a code clone is removed in an extraction step by applying PULL UP TO COMMON FEATURE. Recall that this refactoring removes code that is cloned across two or more features by moving this code to a single location, called a *common feature*. In the example in Figure 7, f_1 and f_2 are such common features.

Preparatory Refactoring: If an extraction step is not directly possible, it is preceded by a preparatory refactoring step, which aligns differing clones. As previously discussed for Figure 6, methods with identical bodies but different names cannot be pulled up. Similarly, extraction is impossible when otherwise identical classes have different names. These examples constitute Type-2 clones, which our tool support can detect. Thus, our tool helps the developer identify code that will benefit from preparatory refactoring. To align naming differences, we offer our RENAME refactoring.

Step-Wise Refinement: Both steps, code clone extraction and preparatory refactoring, can be repeated as often as needed. Since each step only affects a small part of the code base, the correctness of each step can be easily verified. Moreover, as we propose to perform changes by means of variant-preserving refactoring, the external behavior of the affected variants is preserved. Hence, all variants remain in a working state, even if the migration is still ongoing. This is of special importance as it allows releasing new product versions during the migration period. We see this as an advantage over "big-bang" approaches (e. g., [1], [53]), which require migration to be completed before allowing other changes to the code.

Inter-System versus Inter-Feature Code Clones: Cloning frequently occurs within a single software product, e. g., if a method is cloned. This is called an *intra-system* clone [21]. However, we are interested in functionality that is common to two or more variants, thus, focusing on *inter-system* clones, that is, clones that result from copying code from one product variant to another.

As we initially convert each variant into a distinct feature, followed by many small refactoring steps, the origins of each individual clone are difficult to trace over time. Thus, tracking inter-system clones becomes imprecise. In order to avoid these difficulties, we relax the concept of inter-system clones and approximate them with *inter-feature* clones, i. e., clones between different features. Hence, our approach is to use a code clone detector and filter its results so that only inter-feature clones are reported but intra-feature clones are not. As a result, developers can focus on the code clones that constitute the common functionality they need to extract, without being sidetracked by other, irrelevant clones.

VI. TOOL SUPPORT

We integrated our migration approach into the ECLIPSE IDE. To this end, we reused several existing tools, namely FEATUREIDE [29], [46], FUJI [6], the Eclipse refactoring framework, and our variability-aware extension of the *Copy/Paste Detector (CPD)*[2] [48]. In this section, we discuss how we reuse each tool in our implementation.

As the basis for our approach, we use FEATUREIDE, an Eclipse framework for feature-oriented software development. It supports several phases for the development of SPLs, such as domain engineering (i. e., feature modeling), product configuration, and product generation. Notably, FEATUREIDE already integrates the composer FEATUREHOUSE, as well as the variability-aware compiler FUJI [6].

To guarantee variant-preservation of our refactorings, we need a variability-aware AST over all product variants. We reuse FUJI's type checker to generate this AST.

To identify code clones, we reuse CPD, the token-based clone detector that is part of the PMD suite of static source code analysers. We adapted CPD to identify inter-feature code clones [48]. A screenshot of the tool is shown in Figure 8. As the figure illustrates, the code clones are highlighted with a warning, and the corresponding tool tip shows how much code is cloned in other features.

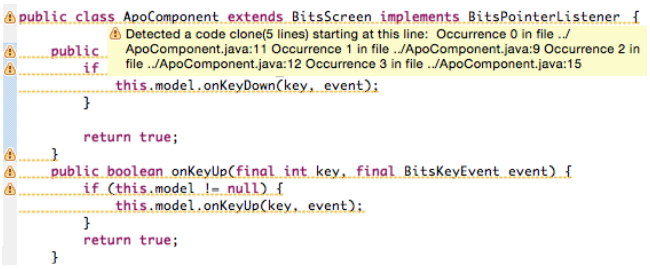


Figure 8. Inter-feature code clone detection based on CPD.

Finally, we implemented our RENAME and PULL UP TO COMMON FEATURE refactorings based on the Eclipse refactoring framework. Thus, we reuse existing machinery, such as the refactoring wizard. In Figure 9, we exemplarily show the application of PULL UP TO COMMON FEATURE on method `getVecY`. In the left dialog at (1.), the destination feature is selected. At (2.) all occurrences of the same method in other features are listed. We mark methods that are Type-1 clones at (3.). In the right dialog, we display a preview of the refactoring. At (4.), we list all the source files that will be modified. A detailed preview of the changes to a specific file is shown at (5.).

VII. EVALUATION

In this section, we evaluate the effectiveness of our approach to migrate existing cloned product variants into an SPL.

[2]http://pmd.sourceforge.net/cpd.html

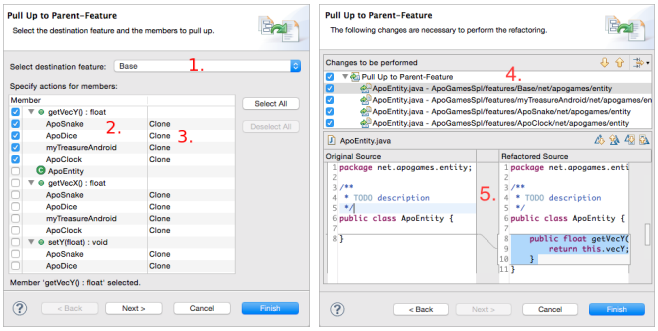


Figure 9. Wizard for the PULL UP TO COMMON FEATURE refactoring.

Specifically, we evaluate how effective our approach is in finding and consolidating inter-feature code clones. To this end, we answer the following research questions:

RQ 1: How much cloned code can be safely and automatically migrated using the* PULL UP TO COMMON FEATURE *refactoring? An important aspect of our approach is to what extent we can automate our migration process. Hence, it is crucial to identify and migrate respective code fragments automatically.

RQ 2: How do preparatory* RENAME *refactorings increase the amount of migratable code clones? As cloned products evolve, code clones diverge. With a preparatory RENAME refactoring we can remove some of these divergences. We quantify its effectiveness by measuring the amount of migrated code again after renaming.

RQ 3: How much cloned code remains and why? Our approach cannot migrate all inter-feature code clones into common features. With this question we investigate which portion of the products still remains as cloned code, and why we could not consolidate these clones.

A. Subject Systems

For answering our research questions, our subject systems must have the following properties. First, they must be Java programs that are cloned from each other. Second, the variants must implement custom functionality, which will remain as variant-specific code. Third, the variants must have evolved over time. Evolution may lead to shared code that is modified without synchronization to other variants. Such modifications allow us to study the nature of divergences and to gauge the effectiveness of our preparatory RENAME refactoring.

In this evaluation, we use five programs of the ApoGames[3] shown in Table I. These programs are diverse games for Android written in Java, created via cloning and evolved independently. In Table I, we show the size of the programs in lines of code (LOC, non-blank, non-comment lines of code, as measured by CLOC[4]) and how much of each program is identified as an inter-feature clone (LOCC). For the latter metric, we used the clone detection tool CPD with a minimum of 10 cloned tokens. As the code clone rates (CCR) show, all programs have a high portion of inter-feature code clones between 57.2 % and 80.0 %.

[3]http://apo-games.de/index_android.php
[4]https://github.com/AlDanial/cloc

Table I
STATISTICS ON THE SUBJECT SYSTEMS

Product	LOC	LOCC	CCR
ApoClock	3,584	2,643	73.7 %
ApoDice	2,504	2,003	80.0 %
ApoMono	6,483	4,382	67.6 %
ApoSnake	2,946	2,350	79.8 %
myTreasure	5,322	3,042	57.2 %
total	20,839	14,420	69.2 %

LOC = lines of code; LOCC = lines of code clones; CCR = code clone rate

B. Methodology

We use the following methodology. First, we transfer the cloned products into a trivial product line as described in Section V. Second, we automatically apply PULL UP TO COMMON FEATURE to all program elements (methods, fields, etc.) that constitute Type-1 clones. After this step no more methods or fields can be refactored using PULL UP TO COMMON FEATURE. By measuring the remaining LOC and LOCC of the individual modules and of the extracted common modules we answer ***RQ 1.*** Third, we apply preparatory RENAME refactorings to all methods, fields and classes that only differ in name but not in content. Afterwards, we apply PULL UP TO COMMON FEATURE again and measure LOC and LOCC. Additionally, we manually inspect the remaining source code of the feature modules representing the five original variants. Thus, we can answer ***RQ 2*** and ***RQ 3.***

C. Results

We applied our methodology to the five ApoGames. In Figure 10, we show the results regarding LOC and LOCC of each feature module of the five variants and the modules for common code. For each module we show from left to right: the initial LOC before migration, the LOC after applying the PULL UP TO COMMON FEATURE refactorings, and the final LOC after preparatory RENAME and PULL UP TO COMMON FEATURE. Additionally, we report the lines that are identified as inter-feature code clones in the upper part (illustrated with a brighter color).

In the first step, we were able to extract 110 fields into 32 common fields and 291 methods into 74 common methods out of three distinct classes. Overall, we reduced the LOC by 4.2 % (879 LOC) and the LOCC by 7.6 % (1,095 LOCC). In doing so, we created nine additional features for common code. The common code size is 419 LOC, of which 187 LOC are shared among all variants.

In the second step, we identified and renamed 84 classes and 8 methods using the RENAME refactoring. After preparation we could pull up 473 additional fields into 150 common fields and 862 methods into 245 common methods out of 26 distinct classes. Compared to the initial variants, the overall LOC is reduced by 15.6 % (3,259 LOC) and the LOCC by 25.4 % (3,664 LOCC). The final size of the common code is 1,779

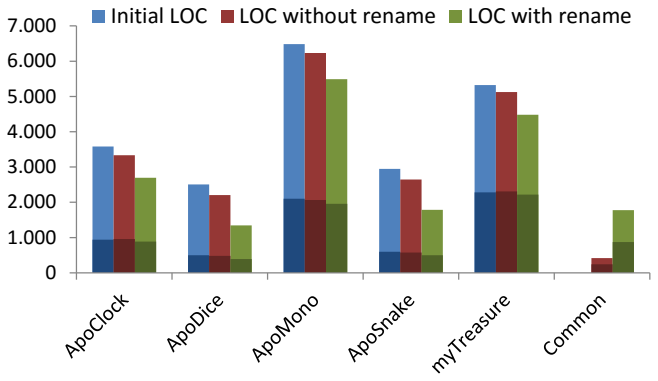


Figure 10. Lines of code in each migration step of each program variant and amount of common code. Detected code clones are illustrated in brighter color.

LOC. The final overall size of the product line is 17,580 LOC with 10,756 LOCC (CCR is 61.2 %). The code size of the five variants (i. e., without the common code) is reduced by 23.2 % (5,038 LOC/LOCC) to 15,801 LOC with 9,850 LOCC. The final feature model has 15 common features. Of these common features, the largest one contains code that is shared among all variants (619 LOC), and the second-largest one contains code shared between ApoDice and ApoSnake (466 LOC).

D. Discussion

Regarding ***RQ 1***, applying PULL UP TO COMMON FEATURE only reduced the code size by 879 LOC (4.2 %). We also observed that it was possible to migrate code from only three out of 114 distinct classes. The reason for the rather low reduction was a peculiar naming convention which required class names to have variant-specific prefix. For instance, the menu class was called `ApoClockMenu` in ApoClock but `ApoDiceMenu` in ApoDice. It was necessary to revert these changes first to improve the effectiveness of migration, which leads to ***RQ 2.***

We renamed 8 methods and 84 classes, reducing the number of distinct classes to 56. This enabled us to apply PULL UP TO COMMON FEATURE to a considerably larger amount of cloned code. After renaming, we were able to migrate code from 26 distinct classes and reduced the variants' code by an additional 3,300 LOC (15.8 %). We argue that by applying both extractive and preparatory refactorings, we substantially reduced code clones and fostered systematic reuse. Also, the number of additional features (15 in our case study), is still manageable, which indicates that our refactoring algorithm avoids creating unnecessarily complex feature models.

Regarding ***RQ 3***, there remain 9,850 LOCC. We reviewed the code clones and found that most of them come from large, similar methods with minimal customizations. (Much of this is graphics-related code.) As it is necessary that methods are completely identical to safely apply PULL UP TO COMMON FEATURE, we were unable to migrate such methods. To consolidate further code clones, we propose to extract local customizations using other preparatory refactorings, such as EXTRACT METHOD and EXTRACT CONSTANT. Corresponding variant-preserving refactorings are part of our future work. We also observed that different variants use different releases of third-party libraries, which have conflicting APIs. This is another source of divergences that prevent the consolidation of clones. Possible solutions would be to either change all variants to use the same library releases or to introduce a façade pattern to abstract from the API differences. However, neither solution lends itself well to automation.

Threats to Validity: Our tool support integrates several external tools and a new implementation of variant-preserving refactoring. To ensure the validity of our implementation, we developed unit-tests for both refactorings. Furthermore, we manually inspected the refactoring results. Finally, we generated all five variants after the migration and observed no compile errors.

For code clone detection we used CPD. We configured CPD that at least ten tokens need to be identical to be identified as code clone. On the one hand, we miss code clones that might be extractable (e. g., very short methods). However, a low number of tokens also leads to many meaningless clones being reported. On the other hand, with a higher number of tokens we would also miss methods that we were able to extract.

In our evaluation, the preparatory RENAME refactoring highly increased our ability to migrate methods and fields. However, this effect is caused by the previously mentioned naming convention in our case study. For other systems, we do not expect RENAME to have such a high impact, but we expect the total reduction of code clones to be similar.

The goal of clone-and-own is to reuse existing code and to integrate customizations. In other systems we expect the amount of custom code to be smaller than the amount of reused code. As the effect of our migration depends on how much code is reused among variants, we expect that we can extract more reused code into shared components.

Our implementation and approach currently only applies to Java programs. Nevertheless, the underlying concept of our approach can be applied to other programming languages, as they all come with elements that can be refactored (i. e., methods or fields).

VIII. RELATED WORK

Apart from the approaches to migrate cloned variants into an SPL that were already discussed in the introduction [1], [38], [51], [52], there is also work on *migrating a single software product* to an SPL (e. g., [10], [26], [49]). A major challenge for migrating multiple variants, however, is to identify and consolidate commonalities (and differences) between those variants. This challenge is not addressed by single-system approaches.

There are multiple approaches to *reverse-engineer commonalities and variations* from the source code of cloned variants [20], [22], [25], [27], [30], [54]. Specifically, Mende et al. used clone detection to identify (but not consolidate) core functionality in SPLs that were evolved by unmanaged, large-

scale copying of code [30]. Koschke et al. recover individual product architectures and combine them into a common product line architecture [22]. Others propose program-dependency graphs [20] or architecture reengineering techniques [27] for identifying product-specific variations. Similarly to us, some of these approaches incorporate code clone detection [22], [27], [30]. Moreover, there is work on identifying features in multiple variants [54] and establishing feature-to-code mappings [25]. While the aforementioned work focuses on analyses, we also focus on source-code transformations to execute the actual migration. Due to the higher level of abstraction, their analyses may complement our approach.

Others have proposed *frameworks for migrating cloned product variants* [28], [40]. These frameworks abstract migration activities such as similarity analysis or merging of commonalities. We, by contrast, provide a specific application of clone detection and concrete refactorings. Both could be used to instantiate these frameworks.

Refactorings for different variability mechanisms were explored, among them aspect-, delta- and feature-oriented programming, as well as C code with preprocessor directives [2], [24], [41], [42]. Our PULL UP TO COMMON FEATURE refactoring is a generalization of the PULL UP METHOD TO PARENT FEATURE refactoring we proposed earlier [41]. Contrary to our current work, these refactorings either lack the migration context [24], [41], [42] or are geared toward single-system migration, thus neglecting the challenges of identifying and consolidating commonalities in multiple variants [2].

Similarly to our work, Higo et al. combine clone detection and refactoring to provide holistic tool support to address code clones [15]. They consider more refactorings than we do, e. g., FORM TEMPLATE METHOD. Due to their focus on an OOP context, however, they do not address the challenges of variability, which arise during migration.

IX. CONCLUSION AND FUTURE WORK

Clone-and-own is frequently used to create variants of a software product. While initially cheap and simple, the effort for synchronizing changes across variants leads to high maintenance costs. Software product lines, by contrast, enable the development of large sets of product variants while keeping maintenance costs low. Existing approaches to migrate cloned products to a product line focus on models and analyses, but techniques to migrate the source code are rare. We propose an approach based on feature-oriented programming that tackles the source-code aspects of migration.

Our approach uses code clone detection to identify commonalities of cloned variants and variant-preserving refactorings to extract these commonalities into shared artifacts. We implemented our approach in FEATUREIDE, thus providing tool support for the tedious and error-prone migration tasks of identifying and extracting common functionality. Beyond their use in migration, we argue that our refactorings, whose object-oriented counterparts are frequently used in single-system development, will in general contribute to the maintenance and evolution of product lines using feature-oriented programming. Moreover, we propose an incremental migration process, which, in contrast to previous "big-bang" approaches, enables enhancements while the migration is ongoing. We evaluated our approach on five cloned product variants, reducing the amount of cloned code by 25 %. Based on qualitative insights, we discuss possible limitations and enhancements that will reduce code clones further, thereby decreasing long-term costs for maintenance and evolution compared to clone-and-own.

In future work, we will design and implement additional refactorings to enable extracting more code clones than currently possible. Moreover, we plan to extend our approach to other variability mechanisms. We are particularly interested in annotation-based mechanisms, such as the combination of C with preprocessor directives, which is popular in industrial product line development. The purely source-code-centric view of our approach may not lead to optimal results in terms of the target architecture or the feature model. Hence, we will incorporate analyses on more abstract levels, such as architectural views and feature location techniques (e. g., [20], [22], [25], [51]). We envision that this will guide refactoring decisions toward an improved overall product line architecture.

ACKNOWLEDGMENTS

This work was partly funded by project EXPLANT of the German Research Foundation (DFG, grant SA 465/49-1) and project NaVaS of the Federal Ministry of Education and Research (BMBF, grant 01IS14017B).

Special thanks to Thomas Thüm for helping develop the idea of variant-preserving migration and for fruitful discussions on the story line of this paper.

REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE '06)*. ACM, 2006, pp. 201–210.

[2] V. Alves, P. Matos Jr, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, "Extracting and evolving code in product lines with aspect-oriented programming," in *Trans. Aspect-Oriented Softw. Development IV*. Springer, 2007, pp. 117–142.

[3] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, Ş. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *Companion to the Proc. Int'l Conf. on Software Engineering (ICSE '14)*. ACM, 2014, pp. 532–535.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines – Concepts and Implementation*. Berlin Heidelberg, Germany: Springer, 2013.

[5] S. Apel, C. Kästner, and C. Lengauer, "Language-independent and automated software composition: The FeatureHouse experience," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 63–79, 2013.

[6] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich, "Access control in feature-oriented programming," *Sci. Comput. Prog.*, vol. 77, no. 3, pp. 174–187, 2012.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[8] P. Clements and C. Krueger, "Point / counterpoint: Being proactive pays off / eliminating the adoption barrier," *IEEE Softw.*, vol. 19, no. 4, pp. 28–31, Jul. 2002.

[9] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232–282, Jun. 1998.

[10] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '11)*. IEEE, 2011, pp. 191–200.

[11] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '13)*. IEEE, 2013, pp. 25–34.

[12] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan, "Forking and coordination in multi-platform development: A case study," in *Proc. 2014 ACM-IEEE Int'l Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, 2014, pp. 59:1–59:10.

[13] W. Fenske, T. Thüm, and G. Saake, "A taxonomy of software product line reengineering," in *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '14)*. ACM, 2014, pp. 4:1–4:8.

[14] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[15] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *Softw. Maint. Evol.: Res. Pract.*, vol. 20, no. 6, pp. 435–461, Nov. 2008.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[17] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: Patterns of cloning in software," *Empir. Softw. Eng.*, vol. 13, no. 6, pp. 645–692, 2008.

[18] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *J. Object Technol.*, vol. 8, no. 6, pp. 59–78, 2009.

[19] J. Kerievsky, *Refactoring to Patterns*. Boston, MA, USA: Addison-Wesley, 2004.

[20] B. Klatt, K. Krogmann, and C. Seidl, "Program dependency analysis for consolidating customized product copies," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '14)*. IEEE, 2014, pp. 496–500.

[21] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *Softw.: Evol. Proc.*, vol. 26, no. 8, pp. 747–769, 2014.

[22] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," *Software Qual. J.*, vol. 17, no. 4, pp. 331–366, 2009.

[23] J. Krüger, W. Fenske, J. Meinicke, T. Leich, and G. Saake, "Extracting software product lines: A cost estimation perspective," in *Proc. Int'l Software Product Line Conf. (SPLC '16)*. ACM, 2016, pp. 354–361.

[24] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer, "Morpheus: Variability-aware refactoring in the wild," in *Proc. Int'l Conf. on Software Engineering (ICSE '15)*. ACM, 2015, pp. 380–391.

[25] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability extraction and modeling for product variants," *Softw. & Syst. Modeling*, pp. 1–21, 2016.

[26] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Proc. Int'l Conf. on Software Engineering (ICSE '06)*. ACM, 2006, pp. 112–121.

[27] J. Martinez and A. K. Thurimella, "Collaboration and source code driven bottom-up product line engineering," in *Proc. Int'l Software Product Line Conf. (SPLC '12) (Volume 2)*. ACM, 2012, pp. 196–200.

[28] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proc. Int'l Software Product Line Conf. (SPLC '15)*. ACM, 2015, pp. 101–110.

[29] J. Meinicke, T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich, "FeatureIDE: Taming the preprocessor wilderness," in *Proc. Int'l Conf. on Software Engineering (ICSE '16)*. ACM, 2016, p. 4.

[30] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '08)*. IEEE, 2008, pp. 163–172.

[31] M. P. Monteiro and J. a. M. Fernandes, "Towards a catalogue of refactorings and code smells for AspectJ," in *Trans. Aspect-Oriented Softw. Development I*. Springer, 2006, pp. 212–258.

[32] W. F. Opdyke, "Refactoring object-oriented frameworks," PhD Thesis, University of Illinois, Champaign, IL, USA, 1992.

[33] T. Pfofe, "Automating the synchronization of software variants," Master's Thesis, University of Magdeburg, Germany, Jan. 2016.

[34] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with VariantSync," in *Proc. Int'l Software Product Line Conf. (SPLC '16)*. ACM, 2016, pp. 329–332.

[35] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proc. European Conf. on Object-Oriented Programming (ECOOP '97)*. Springer, 1997, pp. 419–443.

[36] D. B. Roberts, "Practical analysis for refactoring," PhD Thesis, University of Illinois, Champaign, IL, USA, 1999.

[37] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Prog.*, vol. 74, no. 7, pp. 470–495, 2009.

[38] J. Rubin and M. Chechik, "From products to product lines using model matching and refactoring," in *Workshop Proc. Int'l Software Product Line Conf. (SPLC '10) (Volume 2)*. Lancaster University, 2010, pp. 155–162.

[39] ——, "A framework for managing cloned product variants," in *Proc. Int'l Conf. on Software Engineering (ICSE '13)*. IEEE, 2013, pp. 1233–1236.

[40] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: A framework and experience," in *Proc. Int'l Software Product Line Conf. (SPLC '13)*. ACM, 2013, pp. 101–110.

[41] S. Schulze, M. Lochau, and S. Brunswig, "Implementing refactorings for FOP: Lessons learned and challenges ahead," in *Proc. Int'l Work. on Feature-Oriented Software Development (FOSD '13)*. ACM, 2013, pp. 33–40.

[42] S. Schulze, O. Richers, and I. Schaefer, "Refactoring delta-oriented software product lines," in *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD '13)*. ACM, 2013, pp. 73–84.

[43] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake, "Variant-preserving refactoring in feature-oriented software product lines," in *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, 2012, pp. 73–81.

[44] M. Staples and D. Hill, "Experiences adopting software product line development without a product line architecture," in *Proc. Asia-Pacific Software Engineering Conf. (APSEC '04)*. IEEE, 2004, pp. 176–183.

[45] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and integrated variants in an open-source firmware project," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '15)*. IEEE, 2015, pp. 151–160.

[46] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Sci. Comput. Prog.*, vol. 79, pp. 70–85, 2014.

[47] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund, "Abstract features in feature modeling," in *Proc. Int'l Software Product Line Conf. (SPLC '11)*. IEEE, 2011, pp. 191–200.

[48] K. Tonscheidt, "Leveraging code clone detection for the incremental migration of cloned product variants to a software product line: An explorative study," Bachelor's Thesis, University of Magdeburg, Germany, Jun. 2015.

[49] M. T. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 737–754, Jul. 2012.

[50] J. H. Weber, A. Katahoire, and M. Price, "Uncovering variability models for software ecosystems from multi-repository structures," in *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '15)*. ACM, 2015, pp. 103–108.

[51] Y. Xue, "Reengineering legacy software products into software product line based on automatic variability analysis," in *Proc. Int'l Conf. on Software Engineering (ICSE '11)*. ACM, 2011, pp. 1114–1117.

[52] ——, "Reengineering legacy software products into software product line," PhD Thesis, National University of Singapore, Singapore, 2012.

[53] Y. Xue, Z. Xing, and S. Jarzabek, "Understanding feature evolution in a family of product variants," in *Proc. Working Conf. on on Reverse Engineering (WCRE '10)*. IEEE, 2010, pp. 109–118.

[54] T. Ziadi, L. Frias, M. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '12)*. IEEE, 2012, pp. 417–422.