A Taxonomy of Software Product Line Reengineering

Wolfram Fenske
University of Magdeburg
wfenske@ovgu.de

Thomas Thüm
University of Magdeburg
tthuem@ovgu.de

Gunter Saake
University of Magdeburg
saake@ovgu.de

ABSTRACT

In the context of single software systems, refactoring is commonly accepted to be the process of restructuring an existing body of code in order to improve its internal structure without changing its external behavior. This process is vital to the maintenance and evolution of software systems.

Software product line engineering is a paradigm for the construction and customization of large-scale software systems. As systems grow in complexity and size, maintaining a clean structure becomes arguably more important. However, product line literature uses the term "refactoring" for such a wide range of reengineering activities that it has become difficult to see how these activities pertain to maintenance and evolution and how they are related.

We improve this situation in the following way: i) We identify the dimensions along which product line reengineering occurs. ii) We derive a taxonomy that distinguishes and relates these reengineering activities. iii) We propose definitions for the three main branches of this taxonomy. iv) We classify a corpus of existing work.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Languages

Keywords

Software Product Lines, Taxonomy, Reengineering, Refactoring

1. INTRODUCTION

Software product line engineering (SPLE) has emerged as a paradigm for developing a diversity of similar software applications that share a managed set of features. These applications are developed from a common set of core assets in a prescribed way [13, 33]. Applications developed in this manner are called the *instances* of a *software product line (SPL)*. The focus on reuse of core assets means that changes to one asset can affect any number of instances of the SPL. This adds a layer of complexity that is not present in single-system engineering [4, 43, 46]. In order to handle this complexity, SPLE requires maintaining a clean structure – a task that refactoring promises to solve.

Refactoring is an important part of the software development process, making code easier to understand, easier to modify, and easier to extend [15, 18, 36]. We repeat the definition proposed by Fowler et al. [15]:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Literature on SPLE, however, uses the term "refactoring" in a more liberal fashion. For instance, *feature oriented refactoring* has been proposed as a means to bootstrap an SPL from a legacy application [26, 31, 49]. Furthermore, *a model of refactoring physically and virtually separated features* has been presented, which allows transforming an SPL using a certain implementation technique into an SPL using another technique [25]. More conservative extensions of refactoring to SPLs have also been discussed [45, 46]. For instance, specialized versions of several single-system refactorings (such as *Pull Up Method*) have been proposed as a means to reduce code duplication in feature-oriented SPLs.

The common characteristic of these SPL reengineering activities is behavior preservation, i. e., given the same input, the reengineered programs produce the same output as the original ones. However, the connection to maintenance and evolution is often more obscure. We argue that overloading the term "refactoring" in this manner makes it difficult to distinguish SPL reengineering activities and to see how they are related.

Our work alleviates this problem by identifying three orthogonal *dimensions* along which SPL reengineering occurs. From these dimensions, we construct a detailed *taxonomy* of SPL reengineering, thereby giving distinct names to distinct activities and showing their relationship. Furthermore, we propose *definitions* for the three major classes of reengineering that result from our taxonomy. To show the utility of the taxonomy, we re-classify a part of the corpus previously surveyed by Laguna and Crespo [30] and classify some additional work.

VaMoS '14, January 22 - 24 2014, Sophia Antipolis, France

http://dx.doi.org/10.1145/2556624.2556643

2. SOFTWARE PRODUCT LINES

Software product line engineering is a paradigm for developing similar software applications that share a managed set of features. A feature is an increment in functionality that satisfies a requirement, represents a design decision, and provides a *configuration* option [6].

2.1 Domain Modeling

Relationships and dependencies between the features of an SPL are often described using feature models (FMs) [20]. An FM describes the domain in a tree structure, establishing parent-child relationships between features. Children may be *required* or *optional*, and can form *or-groups* and *alternatives*. At least one child in an or-group must be selected if its parent is selected. In an alternative, exactly one must be selected. Additional dependencies can be specified through *cross-tree constraints.*

A *configuration* determines which combination of features will be present in a concrete product. Any configuration must conform to the constraints imposed by the FM.

2.2 Domain Implementation

Features need to be instantiated on the implementation level. Therefore, changes to the code must respect the constraints of the FM. Moreover, changes to the FM must obey dependencies on the implementation level [4, 43, 46].

A number of different techniques have been proposed for SPL domain implementation. They can be separated into *annotation-based* and *composition-based* approaches [4].

Annotation-based. SPL implementation techniques based on annotations do not physically separate concerns. Instead, there is a common code base that contains the implementation of every feature of the SPL (the "150 % model"). Annotations mark pieces in the code base that implement different features. Product line instances are created by modifying or removing annotated code with the help of a preprocessor [22]. Various preprocessors (e. g., the C preprocessor (`cpp`), `javapp`, `Frames/XVCL`) have been proposed; a number of commercial SPLE tools support this approach (e. g., `Gears`[1], `pure::variants`[2]) [14, 19, 22]. Preprocessors allow fine grained extension points (e. g., individual statements and method parameters), require little additional tool support, and have an easy to understand programming model [22, 24]. Moreover, most preprocessors are language-independent, allowing different artifact types to be handled in the same way. For instance, `cpp` can be used to annotate other kinds of text files, such as SQL scripts and XML documents, not just C source code.

One important variation of the preprocessor theme is *virtual separation of concerns (VSoC)* [22]. Traditional preprocessors are commonly criticized for their susceptibility to introduce subtle errors and their tendency to obfuscate the code base. Furthermore, their lack of separation of concerns and interference with code reuse have been criticized. The VSoC approach favors *disciplined annotations:* Annotations are limited to syntactically meaningful elements in order to prevent syntax errors. Moreover, VSoC proposes product-line-aware type systems and tool support in order to alleviate or solve other problems of traditional preprocessors.

[1] `http://www.biglever.com/`
[2] `http://www.pure-systems.com/`

Composition-based. Composition-based SPL implementation techniques aim at physical separation of concerns. Feature code, i. e., the implementation of a feature, is encapsulated in cohesive units. This encapsulation avoids some problems of annotation-based approaches, such as tangled and scattered feature code, and enables code reuse [6, 21]. Composition-based approaches employ advanced programming concepts (such as component frameworks, plug-in architectures, aspect-oriented programming (AOP)) [8, 17, 47, 48] or extend existing programming languages with feature-oriented constructs (such as feature-oriented programming (FOP), delta-oriented programming (DOP)) [9,37,44]. Compared to annotation-based approaches, composition-based approaches require more learning effort [22]. Furthermore, they are less suited for fine grained extensions, and implementing interacting features can be cumbersome [22, 24]. In contrast to preprocessors, most composition-based approaches are language-dependent [22]. For instance, AspectJ can only be used for Java, AspectC for C, and yet other mechanisms must be used in order to modularize XML or SQL files. However, somewhat language-independent exceptions exist, such as the AHEAD tool suite or FEATUREHOUSE [7,9].

3. TAXONOMY DIMENSIONS

Refactoring for single systems aims at improving the structure of existing code. This notion of refactoring also exists in the SPL context. However, we have identified additional reengineering activities, which pursue other goals. To make these goals more tangible, this section introduces three orthogonal *dimensions* along which SPL reengineering occurs.

3.1 Quality

In the single-system context, refactoring has primarily been associated with changes to improve the design of existing code [15]. Goals include making code easier to understand and modify, and enabling future extensions. Improved quality is easy to argue for many refactorings. For instance, *Extract Method* is used to split up long methods or eliminate code clones. However, the inverse refactoring, *Inline Method*, usually leads to longer methods and code duplication – a *decrease* in code quality. Nevertheless, *Inline Method* may *improve* other properties, such as readability or performance. If, under a given set of circumstances, these are more important than code duplication, the result is an overall improvement. Therefore, we introduce *Quality* as the first reengineering dimension.

3.2 SPL Implementation Technique

As shown in Section 2, annotation-based and composition-based approaches occupy different regions in the design space with regard to separation of concerns, granularity of variation points, or language independence. Our taxonomy reflects this by consistently distinguishing both classes.

Further differences exist between concrete techniques. For instance, although both VSoC and `cpp` are annotation-based, VSoC enforces disciplined annotations, while `cpp` alone does not [22]. Likewise, AspectJ and FEATUREHOUSE are used for composition-based SPL implementation. Nevertheless, AspectJ employs pointcuts and advice, while FEATUREHOUSE relies on superimposition.

Refactoring and other reengineering techniques must cope with these differences between implementation techniques. This leads to the second dimension along which our taxonomy distinguishes SPL reengineering approaches. We call it *SPL implementation technique.*

No.	Name	Description
(1)	*Quality*	Improve some property of the code, the feature model, or the feature-to-code mapping (e. g., readability, extensibility)
(2)	*SPL implementation technique*	Differentiates between SPL implementation techniques (e. g., AOP, FOP, VSoC); Annotation-based and composition-based techniques are distinguished
(3)	*Legacy→SPL*	One or several legacy software product(s) are migrated to an SPL in order to enable mass-customization and systematic code sharing

Table 1: Summary of SPL Reengineering Dimensions

First Author	Classification	Note
Alves [1–3]	*1→AOP*, *Many→AOP*	Eight transformations to extract variable parts of one legacy software product to AspectJ aspects [3]; Description of process to migrate one or several legacy software product(s) to an AOP SPL [2, 3]; Approach is implemented in FLiP tool suite [1]
Calheiros [12]	*1→AOP*, *Many→AOP*	Tool demo of FLiPEx migration tool (part of FLiP tool suite); Implements approach of Alves et al. [2, 3]
Couto [14]	*1→javapp*	Case study migrating the open source UML tool ArgoUML to ArgoUML-SPL
Kästner [23]	*1→AOP*	Case study migrating BerkeleyDB to an SPL using AspectJ; Argues that AOP is unsuitable for migration to an SPL
Kästner [26]	*1→AOP*	Presents tool *ColoredIDE* (now called *CIDE*) for migration of a legacy software product to an AOP SPL; Uses *derivative* model by Liu et al. [31] to migrate interacting features
Liu [31]	*1→FOP*	Algebraic model of migration of one legacy product to AHEAD; Introduces *derivative* model for interacting features
Lopez-Herrejon [32]	*1→FOP*	Migration of single legacy Java products to FEATUREHOUSE; Identifies eight migration patterns
Olszak [35]	*1→FOP*	Semi-automatic feature location and automatic restructuring of one legacy software product; features modules are represented as Java packages; Classes implementing more than one concern are not split, resulting in limited separation of concerns
Trujillo [49]	*1→FOP*	Case study migrating code, XML documentation, and tests of the AHEAD tool suite to a composition-based SPL (implemented in AHEAD)
Xue [51]	*Many→XVCL*	Migration of family of cloned legacy software products to an annotation-based SPL; Combines FM comparison and code clone detection to locate common and variable features
Zhang [52]	*Many→XVCL*	Experience report migrating cloned legacy products to an SPL in order to create product variants for mobile devices
Apel * [4]	Various	Refactoring as path toward a product line (pp. 203–210); Refactoring example catalog (pp. 201, 202) contains *Extract Feature* and *Extract Shared Code* suitable for migration of legacy software products
Valente * [50]	*1→VSoC*	Semi-automatic feature location and annotation in VSoC tool CIDE

Table 2: Classified Work on Variant-Preserving Migration

3.3 Legacy→SPL

A large number of software systems exist that have not been developed as an SPL. The code of these systems may very well implement disparate pieces of functionality which could be called "features". However, including or excluding features in order to create customized variants requires substantial effort. We refer to these systems as *legacy software products*. Various approaches to bootstrap an SPL from legacy software products have been proposed [2, 31, 32, 50, 51]. In order to distinguish them from other reengineering approaches, we introduce a third dimension, *Legacy→SPL*.

Some approaches consider only a single legacy product [31, 32, 50]. The result is an SPL from which new, tailored variants of the original product can be created. In the legacy context, however, customization is sometimes realized through *clone-and-own,* also referred to as *forking*: In order to create tailored variants of a successful software product, the product is copied and adapted as needed [40–42]. The result is a *family of related legacy software products*. Single-product approaches are ill-suited to transform those legacy product families into an SPL, as they lack the means to consolidate similar or identical functionality that is present in more than one product. Specialized approaches have been proposed for this task [2, 51]. For instance, Xue combines model comparison techniques and code clone detection in order to locate common and variable functionality in the legacy products being transformed [51]. Analyses of this kind are neither possible nor necessary when only a single legacy product is migrated. This indicates a marked difference between single-product and multi-product migration approaches. Consequently, we identify two important starting points on the *Legacy→SPL* dimension: *one* legacy software product and *many* legacy software products.

Table 1 summarizes the reengineering dimensions from which our taxonomy is built. For each dimension, we give its name and a short description of its meaning. The taxonomy is discussed in the following section.

4. TAXONOMY OF SPL REENGINEERING

From the dimensions derived in the previous section, we have constructed a taxonomy of SPL reengineering activi-

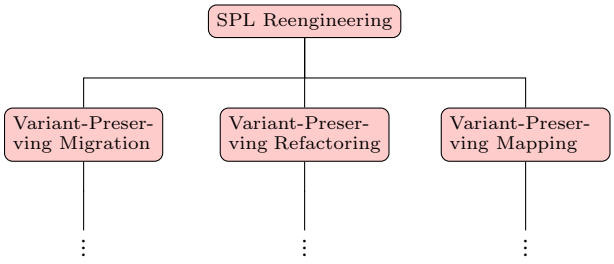

Figure 1: Main Branches of SPL Reengineering

ties. Figure 1 shows an overview.[3] Our taxonomy has three main branches, *variant-preserving migration, variant-preserving refactoring,* and *variant-preserving mapping.* Each of these branches corresponds to one of the three dimensions, which is its *primary* dimension. For instance, *variant-preserving migration* transforms legacy software along the dimension *Legacy→SPL*. To each main branch, we have applied the *secondary* dimension *SPL implementation technique*. With this secondary dimension, we can, for instance, discriminate between *variant-preserving migration to VSoC* and *variant-preserving migration to FOP*.

Figure 2 depicts the relationship of the three SPL reengineering activities to legacy software and software product lines. Via variant-preserving migration, legacy software products (one or many) are transformed into an SPL. The internal structure of an SPL is improved via variant-preserving refactoring. Variant-preserving mapping transforms a product line SPL into an equivalent one, SPL', that uses a different variability implementation technique. In Section 4.2–4.4, we further elaborate on these activities.

We structure existing work according to our taxonomy. We describe the literature selection process in the next subsection. In the following subsections, we give definitions for the main taxonomy branches and describe them in detail.

4.1 Literature Selection

We re-classify a part of the corpus created by Laguna and Crespo [30]. The authors have surveyed 74 publications related to legacy system reengineering and product line refactoring. The scope of their survey is broader than ours, including literature that concentrates on guidelines, processes, or organizational issues. For instance, Bosch and Bosch-Sijtsema discuss the introduction of agile development methods in a project to reengineer an existing SPL [11]; techniques to change code or FM are not discussed. We, in contrast, focus on those techniques. Hence, some of the material selected by Laguna and Crespo has been excluded. As we are interested in changes to a software system, analyses and metrics, such as those presented by Berger et al. [10], have not been re-classified. Moreover, we have excluded work on *refactoring feature modules (RFMs)*, proposed by Kuhlemann et al. [28]. RFMs are essentially single-system refactorings that are applied automatically at product generation time. Therefore, they avoid the SPLE-specific complexity of synchronizing code changes with FM constraints.

This results in a total of 19 re-classified publications of the original 74. We extend this selection with two older and four newer publications (including one book) relevant

[3]A figure of the complete taxonomy can be accessed through our website, `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-reengineering-taxonomy/`.

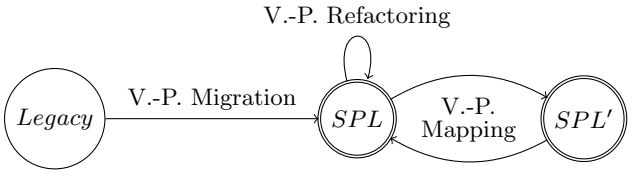

Figure 2: Relationship of Variant-Preserving SPL Reengineering Activities
Note: *V.-P.* stands for *Variant-Preserving.*

to our topic. We believe that this classification gives a representative overview of existing work and shows the utility of our taxonomy. However, we do not claim completeness. The classification result is summarized in Tables 2–4. Approaches with a star (*) next to the author name were not part of the survey by Laguna and Crespo. They are positioned in the lower part of the tables.

4.2 Variant-Preserving Migration

Definition. The first main branch of our taxonomy is concerned with the transformation of one legacy software product or a family of related legacy software products into an SPL. Figure 3 shows a sketch of the corresponding section of our taxonomy. Before we can define variant-preserving migration, we first have to explain the meaning of *legacy software product.*

Definition 1. *A* legacy software product *is a piece of software that has been designed without planning for variability or strategic reuse of the artifacts from which the legacy software product is constructed.*

In short, in the context of this work, we regard any piece of software that has not been developed according to SPLE guidelines a legacy software product. Continuing our discussion, we propose to define *variant-preserving migration* in the following way:

Definition 2. Variant-preserving migration *is the process of transforming one legacy software product or a family of related legacy software products into a software product line such that for each migrated legacy software product there is a product line instance with the same external behavior.*

Dimensions. Variant-preserving migration is reengineering that occurs along the dimension *Legacy→SPL*. The difference between approaches to migrate one or many products is reflected by the nodes *1→SPL* and *Many→SPL*, respectively. In order to further differentiate approaches, we have applied the secondary dimension *SPL implementation technique*. The leaf nodes are labeled according to the targeted SPL implementation technique.

The vertical dots in Figure 3 represent parts of the taxonomy that have been omitted for brevity. For example, the dots to the right of *1→VSoC* stand for reengineering activities that migrate one legacy software product to an SPL implemented in some other annotation-based approach (e. g., `XVCL` or `javapp`). The subtree rooted in *Many→SPL* (also represented by dots) is isomorphic in structure to the one below *1→SPL*.

First Author	Classification	Note
Alves [2]	AOP	Focus on FM transformations that maintain or increase configurability; Synchronization with code changes remains vague
Ghanam [16]	*design patterns*	Introduce variation points in unit tests using the *factory* design pattern (composition-based); Reactive, agile process of SPL refinement
Şavga [43]	FOP	Challenges of refactoring in both *problem* and *solution spaces* (roughly: FM and code)
Apel * [4]	Various	Ch. 8 (pp. 193–212) defines refactoring in *feature-oriented software development*; Catalog of *variability smells* (roughly: code smells in an SPL)
Neves * [34]	AOP	Presents templates for the *safe evolution of SPLs* (considers code, FM, feature-to-code mapping); Validation of templates by analyzing evolution of two SPLs
Schulze * [45]	DOP	Code smell catalog and selected refactorings for DOP; Tool DeltaJ
Schulze * [46]	FOP	Definition of consistent refactoring of code and FM; Catalog of four FOP refactorings that cross feature boundaries (e. g., *Move Field Between Features*)

Table 3: Classified Work on Variant-Preserving Refactoring

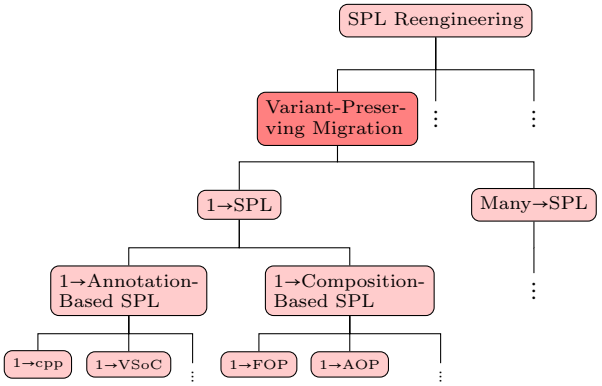

Figure 3: Variant-Preserving Migration

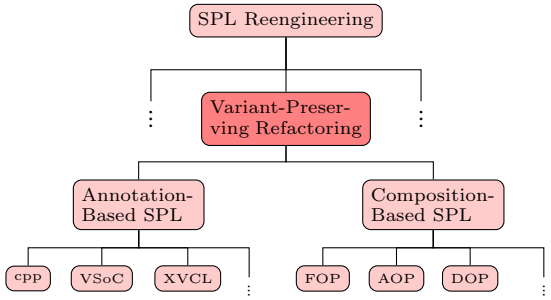

Figure 4: Variant-Preserving Refactoring

Classification. Liu et al. propose *feature-oriented refactoring (FOR)* as a process to decompose a single legacy program into its features [31]. A theory is developed that relates feature refactoring to algebraic factoring [31]. Furthermore, a five-step process is presented. The process relies on a domain expert to name features and label members of a feature (e. g., fields, methods). The code extraction step itself is automated. However, FOR also proposes *derivatives,* special feature modules for interacting features. If two or more features interact, a derivative is created that contains the implementation of this interaction. Reengineering these derivatives again requires manual intervention. FOR only considers a single legacy product. The result of FOR is an SPL implemented using AHEAD, an FOP technique. Consequently, we classify it *1→FOP*.

Alves et al. present eight patterns to extract variable functionality from legacy Java code to AspectJ aspects [1–3]. These patterns are automated by the tool FLiPEx, part of the FLiP tool suite [1, 12]. Furthermore, Alves et al. discuss FM refactoring patterns [2]. However, the combination of code extraction patterns and FM refactoring is not explored in detail. Besides single product migration, a process to migrate a family of legacy products to an SPL is outlined. Each member of the legacy product family is to be migrated separately to a temporary SPL. In order to form the final SPL, the temporary SPLs are superimposed.

Of the material we have selected from the survey by Laguna and Crespo, the largest part (13 of 19 publications) deals with variant-preserving migration. Table 2 shows an overview. Most approaches migrate just a single legacy product, with the work of Alves et al., Xue et al., and Zhang et al. being notable exceptions. Moreover, there appears to be an emphasis on approaches that target composition-based approaches.

4.3 Variant-Preserving Refactoring

Definition. Figure 4 details the part of our taxonomy that covers activities to change the FM or code of an SPL in a behavior-preserving way. The concept of *variant-preserving refactoring* was first introduced by Schulze et al. [46]. We repeat the definition proposed in that paper:

DEFINITION 3. *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.*

Dimensions. The primary dimension along which variant-preserving refactoring changes code, FM, or the feature-to-code mapping is *Quality.* Approaches to variant-preserving refactoring can be distinguished by the implementation technique of the SPL being refactored: Refactoring of `cpp` code

First Author	Classification	Note
Kästner [25]	*VSoC→FOP*, *FOP→VSoC*	Inter-approach mapping between VSoC (annotation-based) and the FOP dialects AHEAD and FEATUREHOUSE (composition-based); Implementation as im-/exports in CIDE; Correctness proof for Featherweight Java
Ribeiro [38, 39]	*`if-else`→AOP*, *`if-else`→inheritance*, *`if-else`→mixins*, *`if-else`→patterns*	Tool to recommend inter-approach mappings from `if-else`-statements (annotation-based) to various composition-based mechanisms (e. g., AOP, mixins, design patterns) and to "configuration files" (configuration files are not explained)
Apel * [4]	Various	Variability smells that indicate need for mapping, such as *`#ifdef` hell or many extension points* (pp. 197–199); Mapping refactorings (e. g., *Change binding-time* maps runtime `if-else` to static `#ifdef` statements) (pp. 201, 202)
Kuhlemann * [29]	*AOP→FOP*	Discusses the feasibility of using FOP to implement 23 aspect-oriented design patterns; Rules to transform AOP into FOP

Table 4: Classified Work on Variant-Preserving Mapping

must handle annotations, whereas refactoring of AOP programs deals with pointcut expressions and advice. Hence, our taxonomy takes the dimension *SPL implementation technique* into account. Note that this results in a hierarchy that is structurally similar to the subtree rooted in *1→SPL* in Figure 3.

Classification. Variant-preserving refactoring pursues the same goal as refactoring defined by Fowler et al. [15], i. e., to improve the design of existing code. For instance, Schulze et al. define variant-preserving refactoring and propose the variant-preserving refactorings *Pull Up Field to Parent Feature, Pull Up Method to Parent Feature, Move Method Between Features, Move Field Between Features* for FOP SPLs [46]. They demonstrate the use of these refactorings for removing code clones, which is a typical maintenance activity.

Later work of Schulze et al. discusses refactoring delta-oriented SPLs [45]. A catalog of code smells and 23 DOP refactorings is presented. The refactorings are automated by the tool DELTAJ.

Alves et al. discuss FM refactoring and patterns for variant-preserving migration to an AOP SPL [2]. However, FM refactoring and migration are discussed separately.

The *factory* design pattern is employed by Ghanam et al. as a means to introduce variation points into an existing SPL [16].

Examples of *variability smells* (essentially code smells for SPLs) and corresponding refactorings are given by Apel et al. [4] (cf. ch. 8). Furthermore, different notions of behavior-preservation for SPL refactorings are discussed.

Table 3 summarizes work on variant-preserving refactoring. This overview indicates that much work is left for variant-preserving refactoring to reach the maturity of refactoring for single systems. The challenges of variant-preserving refactoring have been pointed out [43, 46]. However, the number of actual refactorings is still small, and (at least for FOP) automation is one of the open challenges [46]. Thus far, variant-preserving refactoring seems to concentrate on a small number of composition-based techniques. This is surprising, given the widespread use of annotation-based techniques in practice [4].

4.4 Variant-Preserving Mapping

Definition. Both annotation-based and composition-based SPL implementation techniques have their own benefits and drawbacks. It has been argued that one should be free to switch from one representation to the other, and even mix and match techniques, in order to profit from the respective benefits [5, 25]. For techniques that enable this switch, we propose the term *variant-preserving mapping*.

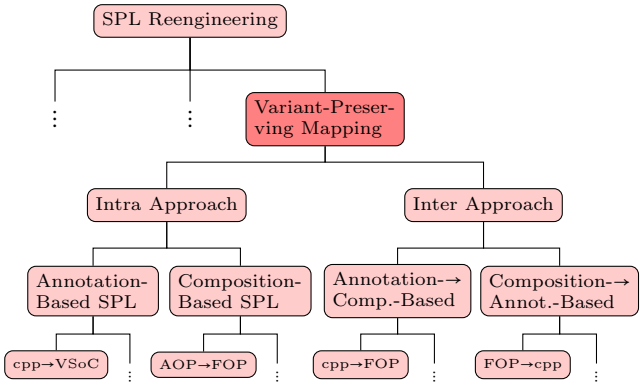

Figure 5: Variant-Preserving Mapping

DEFINITION 4. *A substitution of the implementation technique of a software product line is called* variant-preserving mapping *if for each instance of the original product line there is an instance of the new product line that has the same external behavior.*

Dimensions. Variant-preserving mapping changes the implementation technique of an SPL. Thus, the primary dimension is *SPL implementation technique*. As there is a source implementation and a target implementation, we apply *SPL implementation technique* a second time. Figure 5 shows a sketch of the resulting section of our taxonomy.

Approaches that stay in the same class of implementation technique (either annotation-based or composition-based) are covered by the subtree rooted in *Intra Approach*. *Inter Approach* covers techniques that map the implementation from an annotation-based one to a composition-based one or the other way around.

Classification. Kästner et al. present twelve "refactorings", named R.1–R.12 [25]. R.1–R.5 map code from feature modules to annotated code. They are complemented by R.6–R.12, which map annotated code back to feature modules.

These transformations have been used to implement an import/export facility for the tool CIDE and proven correct for the Java subset Featherweight Java. CIDE itself uses the annotation-based technique VSoC. However, utilizing R.1–R.5 and R.6–R.12, it can import and export projects in the composition-based FOP languages AHEAD, FEATUREHOUSE and the AOP approach AspectJ. Hence, R.1–R.12 are *inter*-approach mappings.

Kuhlemann et al. previously explored the equivalence of AOP and FOP [29]. They provide a set of rules to map aspect-oriented programs to equivalent feature-oriented programs. These rules are validated by mapping 23 aspect-oriented design patterns to FOP.

Ribeiro et al. present a tool to recommend inter-approach-mappings from runtime annotation-based variability (using `if-else` statements) to static composition-based techniques [38,39]. The tool does not perform these mappings, though.

Apel et al. list mapping-related variability smells and informally discuss several refactorings (mappings) to rectify those smells [4] (cf. ch. 8).

Table 4 summarizes work on variant-preserving mapping. Imports and exports à la CIDE are an interesting application of variant-preserving mapping. Mapping may also prove useful as a tool for variant-preserving migration of legacy preprocessor code to more structured SPL implementation techniques.

5. RELATED WORK

Laguna and Crespo have performed a systematic mapping study on software product line evolution [30]. The scope of their survey is broader than ours. Whereas we concentrate on techniques that change a software system, they also discuss work on processes, organizational issues, and metrics. Laguna and Crespo recognize the diversity of reengineering terms (e. g., refactoring, migration, restructuring), yet they do not provide a taxonomy. We extend their work by deriving three dimensions of SPL reengineering and constructing a detailed taxonomy from these dimensions.

Krueger has discussed three approaches to SPLE adoption [27]. We see his work complementary to ours as he dicusses the overall processes, whereas we focus on concrete techniques. In Krueger's *extractive approach,* an SPL is built from one or several legacy application(s). Variant-preserving migration techniques can be employed for this task. The *reactive approach* means extending an existing SPL in order to satisfy new requirements. Any restructuring this might require can be achieved through variant-preserving refactoring. However, if features have to be separated from the common code, migration techniques could also be employed. Finally, the *proactive* approach is presented, which equates to constructing an SPL from scratch. This is not reengineering and hence unrelated to our work.

6. CONCLUSION

"Refactoring" for single software systems has a well-established meaning. In contrast, SPLE literature uses the term for rather diverse (behavior-preserving) reengineering activities. We have proposed a taxonomy that separates these activities into three categories: variant-preserving migration, variant-preserving refactoring, and variant-preserving mapping. We have proposed definitions for these categories and shown how they relate to SPLs and legacy software. Moreover, we have classified a corpus of existing work. Our classification indicates that little work on *variant-preserving* refactoring exists and so far is limited to a few composition-based implementation techniques.

We would like to discuss our taxonomy and definitions with the community in order to improve their utility. A systematic review to provide an exhaustive summary of existing approaches is left as future work. Furthermore, we would like to review approaches published outside of the SPL community, such as aspect-oriented refactorings or refactorings for preprocessor code, in order to assess their applicability to SPL reengineering.

7. REFERENCES

[1] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. FLiP: Managing software product line extraction and reaction with aspects. In *SPLC*, page 354. IEEE, 2008.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE*, pages 201–210. ACM, 2006.

[3] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. Extracting and evolving mobile games product lines. In *SPLC*, pages 70–81. Springer, 2005.

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

[5] S. Apel, C. Kaestner, M. Kuhlemann, and T. Leich. Pointcuts, advice, refinements, and collaborations: Similarities, differences, and synergies. *Innov. Syst. Softw. Eng.*, 3(3-4), 2007.

[6] S. Apel and C. Kästner. An overview of feature-oriented software development. *J. Object Technol.*, 8(5):49–84, 2009.

[7] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Trans. Softw. Eng.*, 39(1):63–79, 2013.

[8] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: The KobrA approach. In *SPLC*, pages 289–309. Kluwer Academic Publishers, 2000.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[10] C. Berger, H. Rendel, and B. Rumpe. Measuring the ability to form a product line from existing products. In *VaMoS*, pages 151–154. University of Duisburg-Essen, Germany, 2010.

[11] J. Bosch and P. Bosch-Sijtsema. Introducing agile customer-centered development in a legacy software product line. *Softw., Pract. Exp.*, 41(8):871–882, 2011.

[12] F. Calheiros, V. Nepomuceno, P. Borba, S. Soares, and V. Alves. Product line variability refactoring tool. In *WRT*, pages 32–33. Technical University of Berlin, Germany, 2007.

[13] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[14] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, pages 191–200. IEEE, 2011.

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

[16] Y. Ghanam and F. Maurer. Extreme product line engineering - refactoring for variability: A test-driven approach. In A. Sillitti, A. Martin, X. Wang, and E. Whitworth, editors, *XP*, volume 48 of *Lecture Notes in Business Information Processing*, pages 43–57. Springer, 2010.

[17] M. L. Griss. Implementing product-line features by composing aspects. In *SPLC*, pages 271–288. Kluwer Academic Publishers, 2000.

[18] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[19] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based variant configuration language. In *ICSE*, pages 810–811. IEEE, 2003.

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

[21] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *McGPLE*, pages 35–40. University of Passau, Germany, 2008.

[22] C. Kästner and S. Apel. Virtual separation of concerns – a second chance for preprocessors. *J. Object Technol.*, 8(6):59–78, 2009.

[23] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320. ACM, 2008.

[25] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *GPCE*, pages 157–166. ACM, 2009.

[26] C. Kästner, M. Kuhlemann, and D. Batory. Automating feature-oriented refactoring of legacy applications. In *WRT*, pages 62–63. Technical University of Berlin, Germany, 2007.

[27] C. W. Krueger. Easing the transition to software mass customization. In *PFE, Revised Papers*, pages 282–293. Springer, 2002.

[28] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *ICSR*, pages 106–115. Springer, 2009.

[29] M. Kuhlemann, M. Rosenmüller, S. Apel, and T. Leich. On the duality of aspect-oriented and feature-oriented design patterns. In *ACP4IS*. ACM, 2007.

[30] M. A. Laguna and Y. Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Prog.*, 78(8):1010 – 1034, 2013.

[31] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, pages 112–121. ACM, 2006.

[32] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed. From requirements to features: An exploratory study of feature-oriented refactoring. In *SPLC*, pages 181–190. IEEE, 2011.

[33] A. Metzger and K. Pohl. Variability management in software product line engineering. In *Companion to the ICSE*, pages 186–187. IEEE, 2007.

[34] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba. Investigating the safe evolution of software product lines. In *GPCE*, pages 33–42. ACM, 2011.

[35] A. Olszak and B. N. Jørgensen. Remodularizing Java programs for comprehension of features. In *FOSD*, pages 19–26. ACM, 2009.

[36] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Phd thesis, University of Illinois, 1992.

[37] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443. Springer, 1997.

[38] M. Ribeiro and P. Borba. Recommending refactorings when restructuring variabilities in software product lines. In *WRT*, pages 8:1–8:4. ACM, 2008.

[39] M. Ribeiro and P. Borba. Improving guidance when restructuring variabilities in software product lines. In *CSMR*, pages 79–88. IEEE, 2009.

[40] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *ICSE*, pages 1233–1236. IEEE, 2013.

[41] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *SPLC*, pages 101–110. ACM, 2013.

[42] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *SPLC*, pages 156–160. ACM, 2012.

[43] I. Şavga and F. Heidenreich. Refactoring in feature-oriented programming: Open issues. In *McGPLE*, pages 41–46. University of Passau, Germany, 2008.

[44] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91. Springer, 2010.

[45] S. Schulze, O. Richers, and I. Schaefer. Refactoring delta-oriented software product lines. In *AOSD*, pages 73–84. ACM, 2013.

[46] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-preserving refactoring in feature-oriented software product lines. In *VaMoS*, pages 73–81. ACM, 2012.

[47] D. C. Sharp. Reducing avionics software cost through component based product line development. In *DASC*, volume 2, pages G32/1–G32/8, 1998.

[48] P. Sochos, I. Philippow, and M. Riebisch. Feature-oriented development of software product lines: Mapping feature models to the architecture. In *Object-Oriented and Internet-Based Technologies*, pages 138–152. Springer, 2004.

[49] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE*, pages 191–200. ACM, 2006.

[50] M. T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *IEEE Trans. Softw. Eng.*, 38(4):737–754, 2012.

[51] Y. Xue. Reengineering legacy software products into software product line based on automatic variability analysis. In *ICSE*, pages 1114–1117. ACM, 2011.

[52] W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. Reengineering a PC-based system into the mobile device product line. In *IWPSE*, pages 149–160. IEEE, 2003.