Software product lines (SPL) gain momentum as a mean for developing and managing a set of related software systems under one umbrella. While intensive research on design and implementation of SPLs exist, the consequences of continuous evolution over time such as a decay of design or implementation have been neglected so far. In this context, refactoring has been shown to be an appropriate mean for improving the structure of source code. In this paper, we provide support for fine-grained program refactoring of feature-oriented SPLs. Particularly, we extend existing, object-oriented refactorings by taking the additional dimension of variability into account. To this end, we present the tool VAMPIRE as a basic framework for such refactorings and explain our considerations during implementation, which has been mainly guided by the idea of decomposing refactorings for ease and understandability. Additionally, we provide a detailed discussion about problems and limitations we faced during the implementation and come up with future challenges that have to be tackled for reliable and automated refactoring of software product lines.

Categories and Subject Descriptors
[Software and its engineering]: Software product lines; [Software and its engineering]: Maintaining software

General Terms
Design

Keywords
feature-oriented programming, refactoring, software product lines, maintenance, software evolution

Implementing Refactorings for FOP – Lessons Learned and Challenges Ahead

1. INTRODUCTION

Software product lines (SPL) is an approach for developing a family of programs of a certain domain. The core idea of SPL engineering is to manage the common and variable parts of the particular programs (called variants) in terms of features. In this context, a feature is an increment in functionality that is visible to a stakeholder. As a result, product lines enable developers to systematically reuse artifacts between variants and foster program generation [7]. For developing SPLs, different implementation approaches exist. In this paper, we focus on feature-oriented programming (FOP), a compositional approach that aims at modularizing software systems in terms of features [18, 4]. Particularly, all artifacts that are related to a feature are encapsulated by a feature module.

Although product lines exhibit differences in the development process, compared to single software systems, their lifecycle is also subject to software evolution, for instance, due to new or changing requirements [12]. While this evolution is a usual and partially wanted process, it may lead to the problem of software aging [17]. As a consequence, the underlying software system may suffer from architectural decays, design flaws, or code pollution. Hence, it is necessary to keep the system up to date, for example, by adjusting the architecture or maintaining the code basis. For the latter, refactoring has been proposed as a mean for improving the internal structure of a system (i.e., the source code) while preserving the external behavior. However, while (source code) refactoring is well-explored and mature for single software systems, variability-enriched programming languages introduce new dependencies among program constructs in terms of features that have to be considered for refactoring.

In previous work, we proposed the notion of variant-preserving refactoring, which means that refactoring for SPLs has to take variability into account and must ensure that all variants preserve their behavior [23]. Additionally, we proposed possible adoptions of existing refactorings for feature-oriented SPLs such as Pull Up Method (PUM) [8].

In this paper, we extend this work by providing a tool for semi-automated, variant-preserving refactoring of feature-oriented software product lines. Particularly, we provide insights into design and implementation of our tool. While this may be particularly valuable as a starting point for a set of SPL refactorings (for FOP), the main contribution of this paper is a comprehensive discussion of lessons learned during the implementation of variant preserving refactorings. We share our experiences on possible limitations of our refactoring tool, discuss the reasons and derive challenges that have to be tackled for the future of variant-preserving refactorings for FOP. By this, we hope to pave the way for finding new directions in both, SPL refactoring in general and refactoring for FOP in particular.
2. FROM OBJECT-ORIENTED TO VARIANT-PRESERVING REFACTORING

In this section, we provide an overview of object-oriented refactorings and details about the notion of variant-preserving refactoring. Particularly, we point out why refactoring product lines is different to traditional, object-oriented refactoring. Moreover, we provide an example of how the object-oriented refactorings can be adopted to feature-oriented SPLs.

2.1 Object-Oriented Refactoring

Originally, refactoring has been defined as changing the internal structure of a software system without changing its observable behavior [16, 8]. Since this definition implies that refactoring provides no benefits in terms of new functionality, developers are scared about applying refactoring: Why should they do so? The answer is that refactoring enables developers to improve quality properties of their source code in an efficient and controlled manner. As a result, the source code is easier to understand, to maintain and less risky to modify (e.g., lower probability to introduce bugs). Additionally, the reliability (and thus acceptance) of applying refactorings depends on two requirements: First, the support of the refactorings process by semi-automated tools. Second, a formal foundation that can be used to ensure the preservation of behavior.

Regarding the aforementioned characteristics and benefits, refactoring is a suitable countermeasure for software aging, since its application can eliminate decays in the design or clean up the code. To identify refactoring opportunities, code smells, such as duplicated code or long methods, have been proposed as "structures in the code that suggest (sometimes they scream for) the possibility of refactoring" [8].

Figure 1: Class diagram for exemplary Pull Up Method refactoring

Fowler summarizes a variety of refactorings in a catalogue-like manner [8], which serves as a foundation for our work. Most of these refactorings are based on the fact that certain preconditions have to be checked and satisfied before applying a certain refactoring. We explain this concept by means of the Pull Up Method refactoring. A common use case for this refactoring is an identical method in two or more classes that share a common superclass. For instance, in Figure 1 we show a class diagram before and after applying PUM. In this example, two classes (PCMenu, MobileMenu) share a common superclass (Menu) and contain a method with the same name(createMenu()) that should be moved to the common superclass to remove redundancies. As a starting point for the PUM refactoring, we have to check whether the methods are identical. This encompasses not only syntactical identity but also semantically equivalent methods (i.e., with same functionality). In the latter case, additional refactorings such as Form Template Method may be applied in advance [8]. Next, we have to check whether the methods have different signatures. If so, we have to decide, which one we want to use in the superclass. Finally, we create the method in the superclass, copy the body and delete the method from the subclasses.

2.2 Variant-Preserving Refactoring

With feature-oriented programming, all artifacts (code and non-code) that belong to a certain feature are modularized into one cohesive unit called feature module. Furthermore, a feature module can be mapped to its corresponding feature in a feature model. Hence, we can compose feature modules, based on a user-specific feature selection, to generate a certain variant of the SPL. Due to the aforementioned decomposition, FOP is characterized by a tight relation between features and classes. More precisely, a set of classes contributes to the realization of a certain feature. Beyond that, a particular class can contribute to different features. Regarding the aforementioned characteristics, FOP is similar to collaboration-based design, where feature modules represent collaborations [31, 25]. In the context of collaborations, a class plays different roles for different features, where each role encompasses all functionality that the class contributes to the respective feature. For instance, in Figure 2 class Menu plays a role in feature PCMenu and feature MobileMenu. In FOP, a role is typically an increment in functionality. This can be realized by adding new functionality or refining existing functionality (of previous roles) in terms of classes, methods, or variables.

A new dimension of refactoring

The presence of variability in general poses new challenges to the analysis of software systems [27]. In particular, the previously explained relation between features and classes is a root cause in FOP for the limited applicability of object-oriented refactoring. The reason is that features add a new dimension for refactoring. Hence, we have to take both, features and classes, into account when considering a certain refactoring, especially for checking the preconditions whether a refactoring can be applied or not. This results in four dimensions of refactoring, which we show in Figure 2.

![Figure 2: Collaboration diagram with different dimensions of refactoring for the TankWar SPL (see http://sp12go.cs.ovgu.de/projects/7 for details)](http://sp12go.cs.ovgu.de/projects/7)
Each dimension implies certain risks and challenges, which are highly related to the scope of refactoring. For instance, refactorings of dimension D0 are restricted to a certain role (one feature, one class) and thus the implications are rather local (i.e., do not affect other classes/feature). In contrast, dimension D3 encompasses different classes and features, which may be affected by the refactoring. Interestingly, even for dimension D0, we have to consider all features for checking the preconditions to avoid side effects during feature composition.

To address the new dimension for refactoring explicitly, we proposed a definition for variant-preserving refactoring (VPR) of software product lines in previous work [23]. In a nutshell, two conditions must be fulfilled for a refactoring to be variant-preserving: First, all valid feature combinations of features remain valid after the refactoring. This condition is mainly concerned with feature model refactoring [1, 28, 5] and thus out of scope of this paper. The second condition requires that each valid combination of features must be (a) compilable and (b) exhibit the same external behavior as before the refactoring. Hence, this condition addresses refactoring of product-line implementation and is subject to the remainder of this paper. We illustrate this notion by an adopted version of the Pull Up Method refactoring.

**Pull Up Method to Parent Feature**

As an example for variant-preserving refactoring, we briefly explain the Pull Up Method refactoring, adopted to FOP. For a more elaborate description, we refer to [23]. As a starting point, we take the collaboration diagram of the TankWar SPL in Figure 2. Additionally, we show an excerpt of the feature model (Figure 3b) and the source code that is subject to refactoring (Figure 3a). As we can see, the class Menu has a method createMenu() in features PCMenu and MobileMenu, respectively. This method is subject to the PUM refactoring and our goal is to move this method to the common parent feature Platform. First, we have to check several preconditions such as whether a common parent feature exists and whether the two methods are identical. If these preconditions are satisfied, we check, whether the class Menu has a role in the target feature. Since this is not the case in our example, we have to create a new role of class Menu in feature Platform. Afterwards, we create an empty method createMenu() in the newly created role and copy the body of one of the original methods. Finally, we delete the original methods in feature PCMenu and MobileMenu, respectively.

**3. SEMI-AUTOMATED REFACTORING FOR FEATURE-ORIENTED SPLS**

In this section we present VAMPIRE, an Eclipse plug-in that supports variant-preserving refactoring for feature-oriented software product lines. Currently, the Pull Up Method refactoring is implemented and an adaptation of Extract Method is in progress. The plug-in is available at https://www.tu-braunschweig.de/isf/research/vampire.

First, we provide details about the general concepts of VAMPIRE and existing tools on which our tool is built. Afterwards, we present details about the implementation with a special focus on decomposition and reuse of refactorings.

**3.1 Concept & Design**

Regarding the example of the Pull Up Method refactoring in Section 2.2, we have to consider both, the feature model as well as the implementation of an SPL, to validate whether a refactoring is applicable or not. To gather the required information for this validation, we rely on existing tools that we introduce in the following. Furthermore, we highlight which information the particular tools provide and how we use this information for the refactoring process.

**FeatureHouse, Fuji, and FeatureIDE**

FeatureHouse\(^1\) is a general approach for language-independent, automated software composition. To this end, FeatureHouse represents software artifacts such as source files as feature structure trees (FST). An FST can be considered as a stripped-down abstract syntax tree (AST) that contains only information about the most modular elements such as packages, classes, or methods. To compose different FSTs (and thus, different software artifacts), FeatureHouse provides two approaches, superimposition and three-way merge, but we focus on the former approach only. In the context of product-line development, FeatureHouse provides information about the different roles for a given set of source

\(^1\)http://www.fosd.de/fh
files and a list of features. For more details about technical insights into FeatureHouse, we refer to [2].

Fuji is a native and extensible Java compiler for feature-oriented programming [3]. While other approaches, such as FeatureHouse, rely on source-to-source transformation, Fuji is a "real" compiler that produces Java byte code. To this end, Fuji makes use of the JastAdd meta-compilation system, in particular the corresponding JastAddJ compiler [9]. For our purposes, the most useful feature of Fuji is the possibility to generate an AST of the software product line, containing information about variability (i.e., features). Furthermore, Fuji is compatible with FeatureHouse, meaning that Fuji makes use of its composing mechanism.

FeatureIDE is an Eclipse plug-in for feature-oriented software development [29]. Beside the fact that FeatureIDE supports different phases of the SPL development process, it reconciles different implementation approaches for SPLs. In particular, the aforementioned tools FeatureHouse and Fuji are available in FeatureIDE for product line implementation. Additionally, FeatureIDE provides support for feature modeling and thus allows for reasoning about feature dependencies such as verifying feature configuration.

Finally, we used Eclipse, a popular and frequently used integrated development environment (IDE). Eclipse is realized as a framework that makes heavily use of plug-ins, which can be built on top of the Eclipse IDE. Since we wanted to provide refactoring support for developers in their natural environment, we decided to realize VAmPIRE as an Eclipse plug-in. Another reason for choosing Eclipse is the tight integration of FeatureIDE with Eclipse. Finally, Eclipse provides object-oriented refactorings together with a corresponding user interface, which we considered to be useful for our purposes.

**Using the existing machinery**

We use the previously introduced tools mainly for two phases of our refactoring process: analysis and change. Next, we illustrate the interaction of VAmPIRE with the existing tools in Figure 4 and explain how this supports both phases. Within the analysis we have to check preconditions to decide whether a refactoring is applicable or not. For this purpose, code changes andMove Method are checked before applying the refactoring while the latter encompass certain operations (e.g., code changes) that have to be applied for the actual refactoring.

**3.2 Implementation**

In the previous subsection, we explained how we use existing tools for analysis and change, required by refactorings. In the following, we will explain how we implemented the actual variant-preserving refactorings based on this information. Overall, our implementation is guided by two principles: decomposition and reuse. The first means that we decompose refactorings into micro-refactorings [19]. This is beneficial for several reasons: easier to understand, less complex and thus less error-prone. Furthermore, we divide micro-refactorings even further by separating them into rules and operations. The first contain preconditions that have to be checked before applying the refactoring while the latter encompass certain operations (e.g., code changes) that have to be applied for the actual refactoring.

The reuse of refactorings is a kind of consequence of their decomposition. Generally, by decomposing complex refactorings, we can reuse the resulting micro-refactorings in different, concrete refactorings. For instance, changing a feature from abstract to concrete may be a micro-refactoring that is useful in different situations such as Pull Up Method or Move Method. Next, we explain both, rules and operations, more detailed and how we put them together for concrete refactorings. We depict an overview of this relation by means of a class diagram in Figure 5.

**Rules**

This part of our tool encompasses the preconditions that have to be checked to decide whether a refactoring can be
applied or not. A developer can implement concrete rules, each checking a certain precondition. To this end, we provide the abstract class ARules, that serves as a common superclass for all implemented rules. While this superclass provides a common foundation, each rule has to implement the actual check of the corresponding precondition on its own. Moreover, a rule may contain further rules that have to be fulfilled in advance. For instance, a rule that checks whether a feature is abstract or concrete may rely on a previous precondition that this feature even exists. Beyond the particular rules, we also implemented logical operators that allow to specify more complex, chained rules by reusing existing ones. Overall, we kept the implementation of rules as modular as possible without dependencies between different rules. As a result, each rule can be reused for different refactorings without any need for adaptations.

Operations

While rules represent the preconditions for refactorings, operations encompass code modifications such as moving or renaming methods that have to be executed when a refactoring is applied. Similarly to the aforementioned rules, we provide an abstract class AOperation (cf. Figure 5) as a common superclass for all concrete operations. Furthermore, AOperation already implements the check of precondition so that it can be reused by all subclasses (i.e., operations) and must not be implemented for each operation separately. For each concrete operation, we can set the respective preconditions that have to be satisfied. More precisely, this is the point where we connect our rules with the corresponding operations. For instance, in the current implementation, we have an operation for creating a new role that contains two rules: One to check whether the respective feature exists and one rule that checks whether the class already exists in this feature.

Putting the Pieces Together

Finally, we combine the implemented rules and operations for specifying concrete refactorings. To this end, we provide an abstract class ASplRefactoring that defines the scope of all concrete refactorings. Moreover, we provide an implementation of how to perform a concrete refactoring, which holds for each refactoring that extends ASplRefactoring. For instance, for realizing the PUM refactoring (cf. section 2), we make use of four operations and the corresponding rules. Two of these operations (for making a feature concrete and for creating a class in a certain feature) are optional, meaning that their application depends on the precondition check. For example, if a feature is already concrete, we must not apply the respective operation. The remaining two operations realize the pull up of all members that are affected by the refactoring and the removal of the original method(s). Note that the rules are not added explicitly to the refactoring but to the operations. Hence, by adding operations to a refactoring, we implicitly add the respective rules as well.

4. DISCUSSION

During the implementation of VAMPIRE, we faced some problems and limitations. In this section, we share our experiences and discuss potential implications for our refactorings. Moreover, based on our experiences, we derive and discuss challenges for future implementations of refactorings for both, our tool and for SPLs in general.

Problems and Limitations

The first experience we made is that managing all dimensions of refactoring (cf. Figure 2) manually is nearly impossible. We identified several reasons for this problem. First, for a certain object-oriented refactoring, it is not clear which dimensions are affected and thus have to be considered for adopting this refactoring. For instance, we implemented the PUM refactoring along dimension D1, which means that we move a method in a different feature but do not move the method to another class. But what about dimension D2 and D3? Does it make sense to define a PUM refactoring for these dimensions? How can we deal with probably hidden feature interactions? One reason that makes it difficult to answer this questions is caused by the fact that we do not know much about use cases of refactoring in SPLs. While for single software systems code smells and anti patterns have been defined to identify refactoring opportunities, such guidelines do not exist for SPLs. Second, the additional dimensions increase the complexity of the refactorings, especially regarding the preconditions that have to be checked. Even for dimension D1 (regarding the PUM refactoring), we had to check a variety of preconditions that involve both, domain and implementation dependencies. We argue that for dimension D2 and even more D3, it is very difficult to ensure soundness and completeness of preconditions. Hence, doing this manually (as we did in our current implementation) is error-prone, because of the risk to miss a certain precondition and thus apply a refactoring that is not variant-preserving. Additionally, too strong preconditions may hinder the application of a refactoring.

A second problem that we experienced is related to the decomposition of refactorings. Generally, decomposing refactorings into micro-refactorings has been shown to be beneficial regarding understandability, complexity, and testing [19, 22]. However, we made the experience that finding the appropriate granularity for decomposition is not as trivial as expected. For instance, we could decompose refactorings into very fine-grained operations such as adding classes (or members thereof), changing the type of a variable, changing access modifiers and so on. This would result into dozens of operations that can be combined in an arbitrary way.

We argue that this fine-grained decomposition may not always be beneficial. First, it may be difficult to manage dozens or even hundreds of micro-refactorings, especially for developers with minor knowledge about refactoring. Hence, the process combining micro-refactorings may be erroneous and lead to refactorings that are not variant-preserving. Second, some of the micro-refactorings we implemented depend on each other, that is, they occur always together. For example, checking whether a feature is abstract always implies a previous check whether this feature exists. This raises the question whether it is necessary to further decompose such micro-refactorings in a fine-grained fashion. Regarding our implementation of operations, we decided not to decompose refactorings at the finest possible grain, because it fits best to our needs. However, further investigations such as empirical studies are required to evaluate an appropriate degree of granularity for decomposing refactorings.

Finally, a limitation we observed with respect to our implementation is the specification of the refactorings. We
Taxonomy of product-line refactorings: Refactoring product lines means to cope with all dimensions of refactoring. Since this is a difficult and complex task, it would be beneficial to know in advance whether a certain dimension has to be considered for the adaptation of an object-oriented refactoring such as Pull Up Method. To this end, it is necessary to identify refactoring opportunities, including the dimensions that have to be considered. We suggest to develop a taxonomy of refactorings for SPLs that support developers in adopting refactorings to product lines. As a first step, this taxonomy requires at least a vague idea, when a refactoring is inevitable due to a structural decay of the source code. We argue that it is necessary to define code smells, similar to those in [8], that are specific to SPLs by taking variability into account. In particular, for FOP we should rely on the popular precondition-based approach used by common IDEs such as Eclipse. This approach allows for a straightforward implementation of refactorings, because we only have to identify the preconditions and relate them to certain refactorings. However, this approach has one major drawback: the implemented refactorings rely on rather loose descriptions without any (formal) specification to be built upon [20]. In fact, the implementation is the only specification that is available. This has several implications for correctness and reliability of the implemented refactorings. First, because of the missing specification, we could not verify the correctness of our refactorings. Especially in complex scenarios such as refactoring along dimension D3, one or more preconditions may be overlooked. Hence, this can lead to situations (especially corner cases) where refactorings may not produce a variant-preserving result. Second, without a formal specification, we not only may change the behavior but also may introduce semantic errors when applying refactorings. While such errors may be detected by rigorous testing, this poses another problem in the context of SPLs: Even a small number of features may lead to thousands of variants, which cannot be tested efficiently. Consequently, we cannot guarantee correctness of our VAMPIRE tool, although we implemented the refactorings with care.

Challenges and Future Directions

Based on our experiences, we identified three major challenges that are of superior importance for the future of product line refactoring:

- **Formal specification of product-line refactorings:** Designing and implementing correct refactorings using the traditional imperative and precondition-based approach has serious limitations and is tedious and error-prone. As illustrated in this paper, this gets even worse in case of variability-aware refactorings. Hence, as a long-term objective, FOP refactorings should be designed on the basis of a formal specification that allows for (1) the verification of correctness properties and (2) the automated generation of implementation templates.

In particular, a conceptual framework for the sound specification, analysis, composition and automated application of FOP refactorings should comprise:

- the formal verification of soundness properties of refactorings including the preservation of the syntactical correctness, the behavior and the variability of the refactored program,
- modularization and composition capabilities for building complex refactorings from micro refactorings [22], and
- conflict detection and resolution among (micro-) refactorings.
Therefore, the refactoring formalism should support the definition of generic refactoring rules and operations in a concise and language-independent way.

We believe that the theory of graph transformations constitutes a promising approach to achieve these goals. Graph transformations constitute a declarative and mathematically-founded, yet applicable framework with lots of existing tool support available [10]. Various standard OO refactorings have been specified on the basis of graph transformations [15]. Those approaches use an abstract representation of the source program by means of a typed graph. Graph nodes refer to common object-oriented entities and edges represent dependencies between those entities being relevant for the applicability of a particular refactoring. Thereupon, application conditions for refactorings are specified by means of combinations of pre-/postconditions comprising graph patterns, (negative) attribute constraints, and forbidden subgraphs. The refactoring application then performs a type- and dependency-preserving graph transformation that is backpropagated into the source program, e.g., using bidirectional graph transformations [24].

Making those techniques applicable to more complex and variability-preserving FOP refactorings requires various extensions to existing approaches. The typed graph representation of programs is to be extended, accordingly, to also comprise (1) modular FOP constructs with respective node types and (2) variability-aware edges capturing the different types of dependencies arising for the different dimensions of FOP refactorings. Thereupon, variability-aware extensions of preconditions and transformations can be specified. For the composition of compound refactorings from micro refactorings, recent approaches rely on critical pair analysis for conflict detection among concurrent graph transformations [15]. However, for more complex refactorings, programmed graph rewriting capabilities are required to compose declarative graph transformations via explicit control flow specifications [10].

Testing of product-line refactoring: Although graph-based abstractions allow for the formal verification of correctness properties of refactorings, testing newly developed refactorings on concrete sample programs is indispensable. Testing a program refactoring requires (1) to provide source programs to which the respective refactoring is applicable and (2) to observe that the program behavior is preserved after the refactoring is applied. This difficult task gets even worse in the presence of variability as (1) appropriate FOP input programs are hard to find and (2) the behavior preservation has to be investigated for all possible program variants. For (1), again, graph transformations techniques provide various approaches to systematically derive input models satisfying predefined constraints. For (2), sample-based as well as family-based testing approaches may be adopted for efficiently testing FOP refactorings [6, 14].

5. RELATED WORK

In this section, we acknowledge work that has been done on (feature-oriented) product-line refactoring and formalizing refactorings.

Liu et al. propose an approach for feature-oriented refactoring based on algebraic factoring [13]. Particularly, they support the process of decomposing legacy applications in features and specifically tackle the problem of different feature implementations in different variants. However, their approach works on a single software system with the goal of decomposing it into an SPL. In contrast, the goal of our approach is to improve the structure of an existing feature-oriented SPL. Kuhlemann et al. propose refactoring feature modules (RFM) to apply refactoring in feature-oriented SPLs [11]. With this approach, a certain refactoring (i.e., the respective implementation) is encapsulated in a separate (feature) module and a corresponding feature is added to the feature model. To apply the refactoring, the respective feature has to be selected for the variant that is subject to refactoring. Hence, this approach is limited to refactoring of certain variants of the SPL. In contrast, our work aims at applying refactoring to the whole SPL and thus to each possible variant. Finally, Borba et al. proposed a language-independent theory of product-line refinement to support the evolution of SPLs [5]. In particular, their theory covers the refactoring of feature models and product-line implementation and thus could be used for a more formal specification of product-line refactorings. In contrast, we propose a concrete, precondition-based implementation without an underlying, formal specification.

Furthermore, different work exists that aim at a more formal, declarative description of refactorings and thus verifying their correctness. Amongst others, Steinmann et al. transform the problem of refactoring to a constraint solvability problem (CSP) [26]. To this end, they propose a formal framework to model the access control rules of Java and thus overcome problems of precondition-based refactoring engines (with respect to access control). Similarly, Schaefer et al. express the preconditions for refactorings as dependency preservation problem [21, 20]. Particularly, they specify certain dependencies that may be affected by refactoring (e.g., name binding in case of the rename refactoring). As a result, they can verify the correctness of a refactoring by checking whether all dependencies are still present after applying the respective refactoring. Another approach of Tip et al. uses type checking to verify the correctness of refactoring [30]. Particularly, they derive a set of type constraints from a type-correct program, which are fulfilled by this (original) program. Afterwards, they show that alternative solutions, such as refactored versions of the original program, that fulfill the type constraints as well.

While all of the mentioned formal approaches are promising even for variant-preserving refactoring they have been developed for plain Java and thus do not take variability into account (as we do with our approach).

6. CONCLUSIONS

Refactoring is crucial for maintaining the design and structure of a software systems. Hence, intensive research has been conducted with the result that refactoring is well-understood and well-explored for object-oriented software systems. Unfortunately, for software product lines, which gain momentum in software development, only little work exists on refactoring and applying them in a semi-automated fashion is far from reality. In this paper, we presented a precondition-based approach for implementing variant-preserving refactoring for feature-oriented SPLs. To this end, we conveyed our conceptual thoughts on the overall design and how to integrate existing tools for our purposes. Furthermore, we provided details about the implementation of our refactoring tool VAMPIRE and how we realized decomposition and reuse of refactorings. While at the end, we only
implemented one concrete refactoring (Pull Up Method), we provide a starting point for the implementation of further refactorings with our tool. We hope that this may motivate other researchers to contribute to practical product-line in future.

Beyond the mere implementation, we also discussed several limitations and problems that we faced when implementing our tool. Based on these observations we came up with challenges that are essential and have to be tackled in future to pioneer practical and automated product-line refactoring.

7. REFERENCES