Structural Feature Interaction Patterns – Case Studies and Guidelines

Sven Schuster, Sandro Schulze, Ina Schaefer
Technische Universität Braunschweig
{s.schuster,sanschul,i.schaefer}@tu-braunschweig.de

ABSTRACT

Feature interactions have been an intensive field of research for more than a decade. Recently, it gained much attention in the context of software product lines (SPLs), a methodology that takes features explicitly into account to distinguish between similar yet different programs. In its very essence, feature interactions occur whenever the presence of two or more features leads to a considerable (and maybe unwanted) change of the behavior of the underlying program. Amongst others, feature interactions may occur on source code level, for example, in form of additional code that is needed due to interacting features. In this paper, we focus on such structural interactions. Particularly, we investigate structural feature interaction patterns, that is, patterns that are related to structural feature interactions, and vice versa. To this end, we propose an approach to detect design patterns in feature-oriented SPLs. We show that such design patterns exist on feature level and these patterns involve feature interactions. Based on our results, we reason about feature interactions in the presence of design patterns. Particularly, we show when feature interactions occur and that they are beneficial, for example, to achieve a higher degree of abstraction. Furthermore, we propose an initial guideline when certain design patterns are applicable.

Categories and Subject Descriptors
H.2.2 [Software and its engineering]: Patterns

General Terms
Design

Keywords
design pattern, feature interactions, feature-oriented programming

1. INTRODUCTION

Feature interactions describe the common observation that two or more features (functionally) interact so that the behavior of the underlying program may be changed. Such feature interactions are especially annoying, if the change of behavior is undesired, because they are often hard to detect. However, not all feature interactions occur suddenly and without purpose. Although feature interactions have been observed years ago and later on in different systems of different domains, they gain superior importance in the presence of variable software systems. Such systems, also known as software product lines (SPL), enable developers to generate different variants from a common code base. To this end, we distinguish common and variable parts of particular variants of an SPL in terms of features. Here, features represent a user-visible increment of functionality. Due to this specific, feature-centric view on a set of programs, feature interactions are a prevalent issue in SPLs.

Recently, Apel et al. proposed a categorization of feature interactions to structure recent and ongoing research in the field. Amongst others, they define the visibility of feature interactions as the context in which such an interaction occurs. For instance, feature interactions are visible by external behavior but also appear internally, i.e., on code level.

In the remainder, we focus on internal feature interactions, which can be further divided into two categories: structural and operational feature interactions. While this categorization is useful for detecting internal feature interactions, it provides only limited support for reasoning about feature interactions such as why they occur and whether they are beneficial or not.

In this paper, we address feature interactions for feature-oriented programming (FOP), an implementation approach for SPLs that aims at modularizing features. Particularly, we present the idea of feature interaction patterns as a mean to reason about feature interactions. To this end, we analyze feature-oriented SPLs for design patterns, a concept introduced to suggest general solutions to recurring problems, independent of the underlying programming language. In the following, we use the term design patterns to relate to the patterns introduced by Gamma et al. We argue that the presence of such patterns from a feature-centric viewpoint indicates how features interact to realize higher level concepts or even design decisions. To evaluate this assumption, we propose a detection algorithm for design patterns, based on structural properties of source code, and present a case study, where we apply this algorithm to show the presence and relation of design pattern with respect to feature interactions. Particularly, we make the following contributions:

- We reason about the possible presence of design patterns in feature-oriented SPLs, why such patterns may
occur across features, and what implications of such patterns are.

- We provide a design pattern detection algorithm that we adopted to FOP. We detect design patterns that take features into account as first-class entities to investigate, whether features interact to realize higher level concepts.

- We present a case study on feature interaction patterns for feature-oriented SPLs. With this study, we show that different design patterns occur (with different frequency).

- We point out how these patterns are related to feature interactions. Based on these results, we provide a reasoning about internal feature interactions, how this interactions takes place and how our approach may support feature interaction detection.

- Finally, we propose the idea of guidelines for applying design patterns, that is, under which circumstances it is beneficial (and applicable) to introduce design patterns in feature-oriented SPLs.

The remainder of this paper is structured as follows. In the next section, we provide background information on design patterns and product lines. In Section 3 we reason about the relation between feature interactions and design patterns in FOP. Afterwards, we present our approach for automatically detecting design patterns in feature-oriented SPLs (Section 4). We then present a case study where we applied our approach to detect design patterns in feature-oriented SPLs (Section 5). Based on the results and a discussion, we attempt to propose guidelines for applying design patterns in FOP in Section 6. Finally, we state related work in Section 7 and finish with our conclusion and future work in Section 8.

2. BACKGROUND

In the following, we introduce the concept of design patterns, followed by the basic characteristics of feature-oriented programming.

2.1 Design Patterns

When developing software, it is common to face recurring problems concerning design and implementation. Design patterns address such problems on an abstract level by providing a general description of the solution [11]. For instance, implementing a set of different algorithms that should be interchangeable at compile-time or run-time is a common task for developers. While design patterns are not a language- or even paradigm-specific concept, we will focus on the object-oriented implementations of design patterns for explaining the general idea [11]. Hence, we will use the term design patterns to refer to these object-oriented design patterns in this section.

With an appropriate use of design patterns, object-oriented design principles such as abstraction, information hiding or a loose coupling of objects can be achieved. Moreover, the reusability and maintainability of the software system can be increased. Finally, modularity such as of algorithms, states, or object creation, is another benefit that can be obtained by applying design patterns.

To reflect these properties, design patterns can be classified in three categories: creational, structural, and behavioral patterns. Creational patterns describe when and how objects are instantiated, whereas the main concern of structural patterns is the composition of classes and objects. Finally, behavioral patterns provide dynamic behavior at runtime by dealing with the interaction between objects.

As an example for a design pattern, we illustrate the observer pattern in Figure 1. The idea of this pattern is that observing classes (i.e., observers) can register at a subject class and this class notifies all registered observers on each state change by calling their update method. As a result, we achieve a loose coupling of classes.

![Figure 1: Class diagram of an observer pattern implementation](11, p. 294)

2.2 Feature-Oriented Programming

As a result of increasing demand for reusability and configuration of complex software systems, the idea of software product lines (SPL) gained momentum in recent years. Basically, an SPL encompasses a set of software systems, which share a common code base consisting of a set of features [18]. Different compositional and annotational approaches for the implementation of SPLs have been developed over time, such as feature-oriented programming (FOP) [19], delta-oriented programming (DOP) [22], or preprocessors [14].

In this paper, we concentrate on FOP, which aims at modularizing the code along features [19]. A feature is an increment of the program’s functionality, visible to any stakeholder. For each feature, a modular implementation unit called feature module contains all corresponding artifacts. Based on the available features, the user can specify a feature configuration that encompasses all features that should be included in a certain variant. To ensure that the selection of features produces a valid variant, a stakeholder (developer or domain expert) specifies commonalities, differences, dependencies amongst features by means of a feature model [10]. Finally, the feature modules that correspond to the selected features are composed based on the composition mechanism used.

In Figure 2, we illustrate an exemplary implementation of a stack product line using FeatureHouse [3]. In this example, the feature BaseStack provides a basic implementation of the class Stack with methods push and pop. Moreover, Feature Peek adds a method peek to class Stack. Finally, Feature Undo adds a field backupPush and a method undo, but also refines the method push by adding a statement and afterwards calls the original implementation of the method using the keyword original. The composed class consists of all defined fields and methods.

3. REASONING ABOUT FEATURE INTER-ACTION PATTERNS

Feature interactions describe the fact that two or more features interact and thus, the behavior of the underlying software system changes. Especially in software product
lines, such feature interactions are challenging, because they may break with feature modularity or impede compositional reasoning. Additionally, feature interactions are often considered to be undesired and thus, their detection is of superior importance. Hence, different approaches have been proposed to detect feature interactions, e.g., [13, 6, 7].

Recently, Apel et al. proposed two dimensions for characterizing feature interactions: order and visibility. Additionally, they propose a categorization for each dimension to answering questions about which feature interactions exist and how do they manifest. While this already supports reasoning about feature interactions, it may be still insufficient to answer the question why feature interactions occur and whether their presence is intentional or not.

Regarding feature-oriented programming, it is imperative that features interact, because FOP relies on the collaboration between classes and features. Moreover, modularizing features into feature modules plays a pivotal role in FOP. However, modularity is not restricted to product lines but also exists for object-oriented software systems. Amongst others, design patterns can be used as a mean to modularize different parts of a system [11, 10]. For instance, with the strategy pattern, different interchangeable algorithms can be encapsulated in different strategy classes. The visitor pattern, as another example, encapsulates an algorithm for an arbitrary number of elements in one visitor class. As a result of applying design patterns, we can achieve not only modularity but also variability, mainly at runtime.

Taking into account that design patterns are used for object collaboration, raises the question whether such patterns also occur between collaborating features and thus, may indicate feature interactions. In fact, some work exist that implies the existence of design patterns for FOP. For instance, in previous work we reasoned about design patterns in feature-oriented programs [24]. Particularly, we proposed to decompose design patterns over several features to realize one-to-one mappings between features and strategies or observers. In Figure 3, we illustrate this decomposition by means of a simple graph product line, where the strategy pattern is applied to features BFS and DFS. This way, we can combine the variability offered by inheritance and object composition with the variability of class composition and refinements. Another application of design patterns in FOP, specifically the decorator pattern, has been proposed by Rosenmüller et al. [21]. Particularly, they use delegation chains with decorator layers for dynamic composition of classes. In both cases, design patterns are applied across features and thus, implicitly foster feature interactions. In the remainder we refer to such design patterns as feature-oriented design patterns.

Given the observations above, we argue that feature-oriented design patterns may exist and that detecting such higher level patterns is beneficial for detection and reasoning about feature interactions. For instance, given a certain pattern that involves several features indicates feature interactions which, in turn, provides information about feature combinations that should be subject to testing or verification. Furthermore, if features interact in the context of design patterns, this may indicate that such an interaction occurs on purpose with the clear goal of providing a solution on a higher level of abstraction. Finally, detecting and analyzing design patterns in feature-oriented SPLs is useful to understand when to apply such patterns and how interactions between features take place. It may even support developers to identify and encapsulate interacting parts of features and, thus, provide better control for feature interactions.

To figure out whether feature-oriented design patterns exist and to verify our aforementioned assumptions, we propose a respective detection algorithm in the following section and a case study about such patterns in feature-oriented SPLs (Section 5).

4. AUTOMATED DETECTION OF FEATURE-ORIENTED DESIGN PATTERNS

In previous work, we manually analyzed feature-oriented SPLs with respect to design patterns, which is a very tedious task with incomplete results. Consequently, an automated yet feature-aware technique for the detection of design patterns is necessary to support this task. Fortunately, design pattern mining is a quite extensive topic and many different approaches and techniques exist to detect patterns. Hence, it is reasonable to reuse and adapt an existing approach to FOP. Rasool and Streitferdt provide a survey with an evaluation of recent mining techniques and tools for object-oriented design patterns, which we considered to be a good starting point. Within this survey, the authors point out that many of these mining techniques make extensive use of dynamic analysis, that is, analyzing the program at runtime. Unfortunately, this would require to analyze single products of an SPL instead of the whole SPL. Since analyzing every valid product (i.e., product-based detection) would be too exhaustive and selecting only a subset of products (i.e., sample-based detection) could lead to an incomplete result, we decided to develop a family-based approach. As a result, we are limited to static analysis only, since dynamic analysis would require an executable program, which, in FOP, would require a valid feature combination (i.e., a

![Feature-oriented implementation of Stack with features Peek and Undo using FeatureHouse](image)
method.
c.
classes. First, the algorithm searches for occurrences of a
ods, extended by the features containing the corresponding
C
candidates
notations of the observer pattern that are described by the
pattern algorithm. This algorithm searches for implementa-
tions of the observer pattern is described as a tuple
by a tuple of program elements. For example, an imple-
mentation of a respective design pattern instance, which is defined
traverse the AST of a program for a quite general descrip-
tion mining technique developed by Heuzeroth et al. 
12. After we evaluated the existing approaches, which use static analysis, we decided to base our approach on the pattern mining technique developed by Heuzeroth et al. 
The reason is that (a) the approach allows to define design patterns in a descriptive way and (b) it works on an abstract syntax tree. Hence, it was easy to adopt for our purposes and considered implementation approach. Actually, this approach combines static and dynamic analysis to detect design patterns by their structural and behavioral characteristics, respectively. Since we were limited to structural analysis, we omit the dynamic analysis part of this approach. Nevertheless, even without the dynamic part of the analysis, the approach by Heuzeroth et al. is still appropriate to find at least candidates for design pattern imple-
mentsations. Additionally, we have to remove false positive patterns manually from the detected candidates.
For the static analysis, Heuzeroth et al. defined algorithms for different concrete design patterns. To this end, they traverse the AST of a program for a quite general description of a respective design pattern instance, which is defined
as a tuple of program elements. For example, an imple-
mentation of the observer pattern is described as a tuple
Subject.addListener
Subject.removeListener
Subject.notify
Listener.update
Algorithm 1: Static detection algorithm for observer pattern in FOP

\[
\begin{align*}
    C & := \emptyset \\
    \text{foreach } & \text{feature } f_c \text{ in } c \text{ do} \\
    Y & := \emptyset \\
    \text{foreach } & \text{class } c \text{ do} \\
    \text{foreach } & \text{method } m \text{ in } c \text{ do} \\
    \text{foreach } & \text{parameter type } p \text{ in } m \text{ do} \\
    \text{if } (p \subseteq c \land c \subseteq p \land p \neq c) \text{ then} \\
    \text{foreach } & \text{feature } f_p \text{ do} \\
    \text{foreach } & \text{call from } c.n \text{ to } p.u \text{ do} \\
    \text{if } isNotifyListener(c.n,p.u) \text{ then} \\
    \text{foreach } & (f_c, c.a1, c.n1, f_p, p1.u1) \in Y \text{ do} \\
    \text{foreach } & (f_c, c.a2, c.n2, f_p, p2.u2) \in Y \text{ do} \\
    \text{if } (c.n1 = c.n2 \land p1.u1 = p2.u2) \text{ then} \\
    C & := C \cup \{ f_c, c.a1, null, c.n1, f_p, p1.u1 \} \\
    \text{else} \quad C & := C \cup \{ f_c, c.a1, c.a2, c.n1, f_p, p1.u1 \} 
\end{align*}
\]

c.n. The second part of the algorithm verifies whether the detected addListener method \((c.a1)\) really resembles an addListener implementation by testing whether the parameter is either used on the right hand side of an argument (potential store) or is passed to another method (potential store method). Moreover, we merge tuples having the same notify and update entries.
As for the observer pattern, we also adapted the algo-
algorithm for the detection of the visitor pattern by Heuzeroth et al. We developed the algorithm for the detection of the strategy pattern from scratch, because no implementation of Heuzeroth et al. exists. The implementation of all algo-
rithms, together with the subject systems for our case study (cf. Section 5) can be downloaded from the web. An im-
portant requirement of our algorithms is that each detected pattern must at least involve two features for being consid-
ered a feature-oriented design pattern. The reason is that, otherwise, the pattern could occur only between classes of one feature and thus, represent the original, object-oriented design pattern. This, in turn would neither be specific to feature oriented SPLs nor exhibit any feature interactions.
Limitations:
Since we only use static analysis for our family-based de-
tection of design patterns, we face some limitations with our approach. First, we can only search for the described structures within the underlying AST and thus, may not detect all implementations of a certain pattern. For example, for the observer pattern, the concrete listener object may not directly implement the listener interface, but inherit this interface from another, possibly abstract, class that imple-
mports the listener interface. Such a different structure would not be detected by our algorithm. Second, the structural analysis has to be as general as possible to detect as many

different pattern implementations as possible. This, in turn, may lead to many false positives. Hence, the result of our detection has to be reviewed manually to identify and remove such incorrectly detected patterns.

**Feature Template Method Pattern:**

In previous work, we not only focused on traditional design patterns in FOP, but also searched for FOP-specific patterns. As a result, we detected a pattern that is similar to the form template method pattern [11], which we named feature template method (FTM) [23, 24]. Basically, this pattern consists of an initially empty method in a certain feature that is extended during refinement by corresponding methods of other features. Usually, this pattern (also known as hook method) is applied to realize fine-grained extensions of methods, especially in the middle of a method body [15]. Since such extension may be required regularly and the respective pattern implies feature interactions as well, we also developed an algorithm to detect the feature template method pattern and included it into our evaluation.

5. CASE STUDY

In this section, we present our case study that aims at analyzing the occurrence of feature-oriented design patterns. Particularly, we point out how such patterns foster feature interactions to realize higher level concepts and thus, truly represent feature interaction patterns.

5.1 Research Questions

For a more structured process when conducting our case study, we formulate two research questions based on the reasoning about design patterns in Section 3.

**RQ.1:** To what extent do design pattern exist in feature-oriented SPLs?

Although we have shown in previous work that it is reasonable to introduce feature-oriented design patterns, it is unclear whether such patterns exist (and to what extent). With this question we aim at quantifying the existence of design patterns in FOP.

**RQ.2:** Do feature interactions occur in the context of design patterns and if so, how do the features interact?

In case that feature-oriented design patterns exist, we want to relate them to feature interactions. Particularly, we want to know whether feature interactions exist in the context of design pattern and, if so, which kind of interactions (e.g., refinement of classes/methods, caller callee relations) they resemble.

5.2 Setup & Methodology

To answer our research questions, we analyzed seven different SPLs of different size and domain. In Table 1 we show a summary of the analyzed product lines, including their respective number features and source lines of code (SLOC). All SPLs have been implemented in Java using Fuji, a full-fledged Java compiler for feature-oriented product lines [2].

For each subject system, we applied the respective detection algorithm of the four design patterns, introduced in Section 2, observer, strategy, visitor, and feature template method. We focus on these patterns for two reasons. First, these patterns are well-known and commonly applied in (object-oriented) software systems. For instance, the observer pattern is a solution to implement the model-view controller (MVC) principle. Second, all of these patterns can be analyzed statically, which is required for our family-based approach.

As a result of the detection step, we obtain a list of candidates for each pattern and subject system. Then, we analyzed each list manually to decide, whether a candidate is really a design pattern instance or not. Since we are interested in design patterns that address feature composition (and not only object composition), we focused on pattern implementations that are scattered across several features.

5.3 Results

In Table 2, we show the quantitative results of our study. Overall, we detected decomposed implementations of every design pattern. However, the strategy and feature template method pattern occur way more often than the two other patterns. In the following, we present the results for each pattern more detailed.

**Visitor Pattern:**

We detected this pattern in two product lines. In the Graph Product Line, the pattern is applied to encapsulate the graph algorithms such as finding circles or determining the number of each vertex. These algorithms are encapsulated in six different workspaces, which are implemented as visitors in (separate) sibling features. Then, in the two alternative features DFS and BFS, the visitable element is instantiated. In Figure 4 we illustrate one implementation of the visitor pattern in the GPL. Within this example, feature DFS contains the visitor interface WorkSpace, with the visit method preVisitAction, and the element class Vertex, with the accept method nodeSearch. Furthermore, in feature DirectedOnlyVertices, a Graph implementation is cre-
3.2 Pattern Recommendations

Regarding the results of our pattern detection, the answer of this question is twofold. Generally, we detected all of our analyzed (feature-oriented) design patterns. However, the number of patterns we detected differs depending on the concrete pattern. In particular, we observed frequent occurrences of the strategy and the FTM pattern, while visitor and observer pattern occur rarely. Regarding the strategy pattern, the frequent occurrences may be influenced by the selected subject systems. Most of these systems contain algorithms, encapsulated in features, that can be interchangeably used. In contrast, for the FTM pattern, we argue that this pattern is commonly used, independent of the concrete SPL, because we detected it in all of our product lines.

4 Discussion

In this Subsection, we interpret our results and explicitly relate them to our research questions.

RQ.1: To what extent do design pattern exist in feature-oriented SPLs?

Regarding the results of our pattern detection, the answer of this question is twofold. Generally, we detected all of our analyzed (feature-oriented) design patterns. However, the number of patterns we detected differs depending on the concrete pattern. In particular, we observed frequent occurrences of the strategy and the FTM pattern, while visitor and observer pattern occur rarely. Regarding the strategy pattern, the frequent occurrences may be influenced by the selected subject systems. Most of these systems contain algorithms, encapsulated in features, that can be interchangeably used. In contrast, for the FTM pattern, we argue that this pattern is commonly used, independent of the concrete SPL, because we detected it in all of our product lines.

RQ.2: Do feature interactions occur in the context of design patterns and if so, how do the features interact?

Due to the fact that we searched for decomposed patterns, implemented across features, we would assume that they interact, without knowing how this interaction takes place. To answer this question, we analyzed each pattern manually to qualitatively reason about the kind of feature interactions. First, for the GoF-Patterns (visitor, strategy, observer) we observed no direct interaction (e.g., no class refinements, no derivatives), because only new concrete classes are added (e.g., for strategy or observer), but they are never refined afterwards. However, this way of implementing the patterns directly affects the product-line structure. For instance, for the visitor pattern, all elements that can be visited have to be mandatory, because introducing a new element would require to implement a new visit method in every concrete visitor class and thus require class refinements of the visitors.
While this would be possible to implement, a more comprehensive implementation is necessary to evaluate pros and cons of different pattern implementations.

Generally, the implementation of patterns is similar to the visitor pattern we illustrated in Figure 4. Initially, all of the visitable elements exist in the SPL from the beginning. Then, for each feature, using these elements, new concrete visitor classes are introduced and code is added to call the visitor. Hence, we observed mostly caller-callee interaction between classes of different features. We made similar observations for the strategy and observer patterns, where concrete strategies or, in one case, a concrete observer are added in separate features.

Moreover, when adding new classes, additional code is required to actually use or at least register these classes. Consequently, class refinements are used within every feature that adds a concrete class to a pattern implementation.

Finally, the feature template method, as FOP-specific pattern, does occur quite often in every product line. In this pattern, the template method is always refined by sibling and mostly optional features. In most cases, the feature template method is applied to make use of new features, mainly to register them to the program or call them. Particularly, in the GPL it is used to call the graph algorithm visitors. To this end, each feature that introduces a new visitor class refines the run method by a call to the accept method of the element passing the corresponding visitor.

Overall, we argue that design patterns impose feature interactions in two ways: by establishing a caller-callee relation across features and by establishing a refinement chain that involves different features. Furthermore, we observed that the FTM pattern is used in combination with the detected GoF patterns.

5.5 Threats to Validity

Although we conducted our case study carefully, it may exhibit limitations. First, we use only static analysis to detect design patterns, which may lead to incomplete or false results. However, we reviewed detected patterns manually to exclude false positive and thus, to improve reliability of our results.

Second, we conducted our study on SPLs implemented with Fuji. Hence, our results may not be generalizable, especially regarding the relation of patterns and feature interactions. Nevertheless, we observed that certain patterns occur across all subject systems and that for all patterns, feature interactions take place in a similar way. Additionally, the selected subject systems may be a threat, because they may contain a superior amount of design patterns. For both threats, mentioned before, a more comprehensive study including other implementation approaches could provide valuable information on the generalizability of our findings.

6. TOWARDS GUIDELINES

In the previous section, we presented our results that indicate, that design pattern are common in feature-oriented SPLs. Next, we formulate an initial guideline that should support developers in applying design patterns by thinking about their usage in advance.

First of all, we argue that the application scenarios of feature-oriented design patterns are quite similar compared to the original design patterns. In fact, our case study indicates that design patterns support the modular decomposition of features. For instance, the visitor pattern in GUIDSL is used to traverse through a grammar or the AST of a program. In GAMEOFLIFE, visitors are applied to traverse through a graph and the observer pattern is applied to notify the GUI on model changes. Moreover, strategy patterns are used to implement alternative yet similar algorithms. All of these scenarios are very common applications of the respective design patterns and they are solved the same way in FOP as in OOP, with the difference that features are taken into account as first-class entities. Interestingly, applying the patterns in a decomposed way makes no use of class refinements to modularly extend concrete pattern classes such as strategies, observers or visitors. Although this would be possible, for instance, by implementing fine-grained visitors that are tailored to optionally added elements, we advise to refrain from doing so. The reason is that optional elements and optional visitors would lead to the optional feature problem, which is critical especially in the light of feature interactions.

Second, we observed decomposed design patterns mostly occur within sibling features. In particular, for example, visitors are used to implement different algorithms for the same group of elements. Thus, they are often quite similar to each other, serving the same purpose and thus are often encompassed by the same feature group. For instance, the graph algorithms in the GPL are all sub features of feature Alg. Hence, we advise to reflect design patterns also in the domain model by putting features implementing a design pattern together. We argue that this provides a more intuitive way of understanding how these features are related and reflects the actual use of design patterns.

For more profound and detailed guidelines, which we highly recommend, more data on the occurrence of design pattern and their relation to feature interactions in SPLs is needed.

7. RELATED WORK

Both, feature interactions and design patterns are subject to substantial research.

In previous work, we analyzed object-oriented design and especially design patterns in feature-oriented programming by reasoning on possible decompositions of design patterns and manually searching for such decomposed pattern instances [23, 24]. We extend this work by a) automating the design pattern detection and b) proposing the idea of feature interaction patterns.

Recently, Apel et al. proposed a categorization of feature interactions to structure recent and ongoing research in the field [5]. They concentrate on declaring, which kinds of feature interactions exist and how they manifest in the wild. However, their classification may still be insufficient to answer why feature interactions occur and whether their presence is desired. In contrast, we analyzed why and how feature interactions occur, concentrating on structural feature interactions related to higher level concepts such as design patterns.

Next, reasoning on feature interactions [9, 16] and detecting feature interactions [13, 6, 7] gained much attention in research. We complement both by providing information about the relation of feature interactions to higher level concepts, especially design patterns.

Moreover, by proposing guidelines for applying design patterns to FOP, we contribute to the research on design and modularity in SPLs [1, 16].
Finally, a variety of approaches exist to detect design patterns in object-oriented software [20]. We extend this work by proposing a feature-aware and family-based detection technique for feature-oriented SPLs.

8. CONCLUSION & FUTURE WORK

In this paper, we presented an approach for detecting design patterns in feature-oriented SPLs. Based on this approach, we conducted a case study to investigate the occurrence of such patterns and how they relate to feature interactions. As a main result, we have shown that (a) different (feature-oriented) design patterns exist and (b) that they impose feature interactions by class refinements and caller-callee relations.

Nevertheless, we consider our work as a first step for understanding SPL design, which requires more comprehensive analysis of such patterns. Besides detecting different design patterns, we aim at improving the detection algorithms by taking additional information such as naming conventions into account. Furthermore, we plan a more detailed characterization of the related feature interactions. Particularly, a case study on pattern-related interactions and possible type or syntax errors would give insights on the harmfulness of such interactions.

9. REFERENCES