Detecting and Describing Variability-Aware Design Patterns in Feature-Oriented Software Product Lines

Sven Schuster, Christoph Seidl and Ina Schaefer
TU Braunschweig, Braunschweig, Germany

Keywords: Software Product Line, Feature-Oriented Programming, Design Pattern, Family Role Model.

Abstract: Software Product Lines (SPLs) enable customization by reusing commonalities and variabilities within a family of similar software systems. Design patterns are best practices of established solutions in object-oriented source code for recurring design challenges. Although certain design patterns realize variability, they are only defined in the context of stand-alone systems and not for SPLs. Employing design patterns to realize variability allows using best practices in design for SPL development. However, the exact usage of design patterns within SPLs has not been explored, and a formal notation to capture their usage within different features does not exist. In this work, we provide a model-based analysis method to determine the variability-aware usage of design patterns in source code within the context of Feature-Oriented Programming (FOP). Moreover, we introduce Family Role Models (FRMs) as an extension to role modeling, which offer a language-independent, unified, formal notation for decomposed design patterns. We apply the analysis method in a case study on the variability-aware usage of design patterns in feature-oriented SPLs and derive FRMs from the results.

1 INTRODUCTION

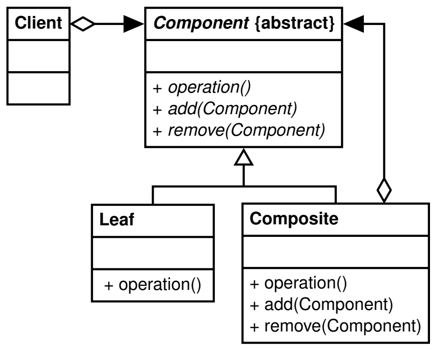
In recent years, *Software Product Lines (SPLs)* gained momentum due to an increasing demand for customizing software (Clements and Northrop, 2001; Pohl et al., 2005). SPLs reuse commonalities and variabilities to realize customization while decreasing cost and effort. *Feature models* (Kang et al., 1990; Czarnecki and Eisenecker, 2000) are variability representations for SPLs describing commonalities and variabilities on a conceptual level in terms of *features* (e.g., see Figure 10). Features may be either *optional* or *mandatory* and are usually arranged in a tree-structure (Kang et al., 1990). *Feature-Oriented Programming (FOP)* (Prehofer, 1997; Batory et al., 2004) is a compositional approach for SPL development, which allows modularizing realization artifacts, such as source code in various languages, along increments to functionality of individual features (Apel and Kästner, 2009). According to a *configuration* for a stakeholder (i.e., a specific feature selection), a core module is composed with *feature modules* of the selected features to form a particular *variant* of the SPL.

Software design is a crucial task during software development. Various design concepts emerged, including *design patterns* as best practices for recurring design problems in *Object-Oriented Programming (OOP)* (Gamma et al., 1994) in various languages. To guide the design process, catalogs of design patterns have been assembled that list patterns by the main concern they address. Some of these patterns address encapsulating variation to achieve modularity and reusability (Apel et al., 2013).

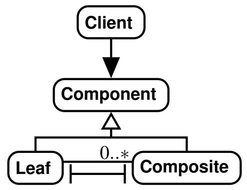
Due to its modular nature, FOP offers a new layer of design that allows refining realization artifacts and, thus, extending them with new functionality. However, only little is known about realizing high-quality design in the context of FOP (Apel and Beyer, 2011; Kästner et al., 2011). Regarding design patterns, it is known *that* they are applied in the context of FOP (Schuster et al., 2013), but it is unclear exactly *how* they are used to implement variability. Moreover, no dedicated formalism for the description of design patterns in the context of SPLs exists.

In this work, we analyze the usage of design patterns in the context of source code used within FOP, i.e., how their realization is decomposed along features. As design patterns are established solutions for common design problems, we cannot *reason* on their variability-aware implementation, but have to *collect evidence* on their existence and their exact application. This information is intended to serve as basis for documenting best practices in designing SPLs, which may be used for the implementation of vari-

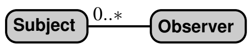
Schuster, S., Seidl, C. and Schaefer, I.
Detecting and Describing Variability-Aware Design Patterns in Feature-Oriented Software Product Lines.
DOI: 10.5220/0006749307310742
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 731-742
ISBN: 978-989-758-283-7



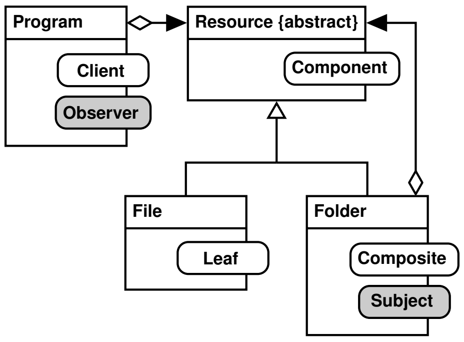
(a) Class diagram of the Composite pattern



(b) Role diagram of the Composite pattern adapted from (Riehle and Gross, 1998).



(c) Role diagram of the Observer pattern.



(d) Class ability diagram mapping Composite and Observer patterns to a tree-structured file system.

Figure 1: Specifying and applying the Composite and Observer patterns using role modeling.

ability within various object-oriented (OO) languages. To obtain the relevant information, we present a model-based analysis method and a corresponding implementation based on the *Eclipse Modeling Framework* (EMF) and the *Java Model Parser and Printer* (JaMoPP) to allow an automated analysis of the design pattern usage in feature-oriented SPLs. In this work, we focus on design patterns identified as beneficial to implementing variability due to their structural properties and intended use–namely the *Composite*, *Observer*, *Strategy* and *Template Method* patterns (Apel et al., 2013).

The contribution of this paper is twofold:

1. We provide a model-based analysis method to allow the automated detection of design pattern usage in FOP.
2. We introduce *Family Role Models (FRMs)* as a domain-specific language (DSL) based on role modeling (Reenskaug et al., 1996) to describe the variability-aware usage of design patterns in SPLs independent of the concrete implementation in source code.

We evaluate our approach by analyzing seven existing SPLs for variability-aware usage of selected design patterns and we document our findings to serve as basis for future implementations.

The rest of the paper is organized as follows. In Section 2, we provide background information on design patterns and role modeling. In Section 3, we present the model-based analysis method. In Section 4, we introduce FRMs as a DSL for describing variability-aware design patterns. In Section 5, we describe our case study on the decomposition of variability-aware design patterns. In Section 6, we discuss related work. In Section 7, we close with a conclusion and present an outlook to future work.

2 BACKGROUND

In the following, we describe *design patterns* as best practices for design in object-oriented languages and *role modeling* as a notation to capture dynamic object collaborations instead of static class design.

2.1 Design Patterns

Design patterns are time-proven standard solutions for common, recurring design problems in object-oriented software. (Gamma et al., 1994) documented design patterns by extracting best practices from real-world examples and describing them consistently.

Despite design patterns being general descriptions of best practices, they are documented using a notation similar to class diagrams, which would suggest a definite design that can be copied to be used. However, design patterns do not constitute final design decisions, but rather general descriptions of how to solve the problem, i.e., design patterns are not static solutions. To apply a design pattern in an OO-language, it has to be tailored to the specific application, e.g., class and operation names may have to be adapted.

Figure 1a illustrates the *Composite* pattern (Gamma et al., 1994) as an example of a design pattern. The intent of the Composite pattern is to realize a hierarchical tree structure to represent part-whole hierarchies of components. The Composite pattern consists of a `Client`, referencing a `Component`. Instances of the `Component` can either be of type `Leaf` or `Composite`. `Composites` hold children by referencing the common superclass `Component`, i.e., children can be treated uniformly, regardless whether they are instances of `Leaf` or `Composite`.

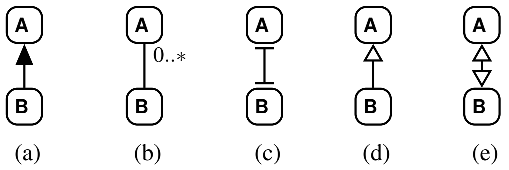


Figure 2: Role diagram notation (Riehle and Gross, 1998). (a) Use (b) Association (c) Prohibition (d) Implication (e) Equivalence.

2.2 Role Modeling

In OOP, developers are concerned with static class design, i.e., object composition via references as well as class inheritance (as captured in UML class diagrams). However, depending on the view on the object and the considered interactions, an object can play multiple different *roles* in a specific *context*. To capture dynamic object collaborations, *role modeling* was introduced (Reenskaug et al., 1996).

Role modeling is a modeling language addressing the collaborations of different "players". In a role model, *roles* as well as constraints between these roles are used to describe such collaborations (Riehle and Gross, 1998). Roles may be mapped to rigid objects (e.g. objects of OOP languages, model elements, etc.), which is perceived as the object *playing* that role. However, role modeling is a general modeling notation not limited to OOP, which means that roles can be mapped to any kind of entity of other modeling concepts, such as, e.g., features of a feature model.

As the intent of role modeling is to model collaborations instead of a definite design, it is a suitable basis for describing design patterns (Riehle, 1996; Riehle, 1997). In this work, we adopt the role diagram language introduced by (Riehle and Gross, 1998), consisting of five role constraints (cf. Figure 2).

Use. An object playing role B *uses* an object playing role A.

Association. An object playing role B *"knows"* of a given number of objects playing role A.

Prohibition. An object playing role B may never play role A in the same context.

Implication. An object playing role B also plays role A.

Equivalence. An object playing role B also plays role A and vice versa.

Figure 1b depicts a role model for the Composite pattern (Riehle and Gross, 1998). To realize the unified treatment of `Leaf` and `Composite`, objects playing the *Leaf* or *Composite* role are also regarded *Components*. An object playing the *Composite* role references a number of objects playing the *Leaf* role, which creates the parent-child relation. Objects playing the *Composite* role may not play the *Leaf* role in the same context, i.e., an object may not reference itself as a child. This way, a part-whole hierarchy may be constructed.

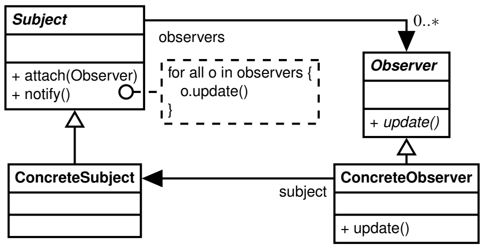


Figure 3: Class diagram of Observer pattern (adapted from Gamma et al. (Gamma et al., 1994, p. 294)).

Furthermore, Figure 1c depicts a role model for the Observer pattern (Riehle and Gross, 1998). The Observer pattern is used to achieve an event notification of objects (*Observers*) depending on another object's (*Subject*) state. Hence, a *Subject* is observed by a number of *Observers*.

These language and realization-independent representations of design patterns may be mapped to a static class design such as the tree-structured file system illustrated in Figure 1d. This way, a `Folder` may contain an arbitrary number of `Resources`, which can either be instances of `File` or `Folder`. Moreover, the `Program` may access a `Resource` and observe a `Folder`, e.g., for file changes.

Although this is a suitable implementation for the Composite and Observer patterns, it is not the only possible implementation. Different incarnations of the patterns can be implemented by creating alternative mappings to classes that fulfill the role models of the patterns. Hence, role modeling is a suitable technique to describe the general collaborations of a design pattern independently of its actual implementation and the respective concrete language.

3 DETECTING VARIABILITY-AWARE DESIGN PATTERNS

In previous work (Schuster et al., 2013), we established that design patterns exist in feature-oriented SPLs and that they are implemented across several features. However, there was no inspection of the actual usage of these patterns, i.e., how exactly they are decomposed over features. In this section, we present an analysis method to automatically identify and locate design pattern instances and determine how a detected design pattern instance is distributed across features.

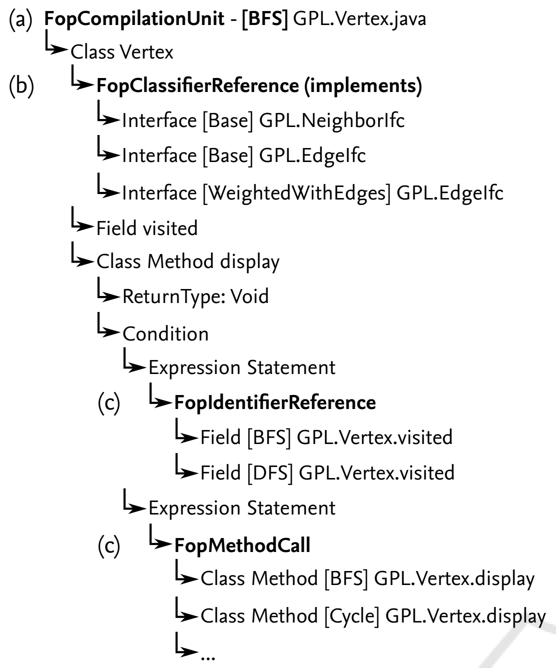

(a) FopCompilationUnit - [BFS] GPL.Vertex.java
Class Vertex
(b) FopClassifierReference (implements)
Interface [Base] GPL.NeighborIfc
Interface [Base] GPL.EdgeIfc
Interface [WeightedWithEdges] GPL.EdgeIfc
Field visited
Class Method display
ReturnType: Void
Condition
Expression Statement
(c) FopIdentifierReference
Field [BFS] GPL.Vertex.visited
Field [DFS] GPL.Vertex.visited
Expression Statement
(c) FopMethodCall
Class Method [BFS] GPL.Vertex.display
Class Method [Cycle] GPL.Vertex.display
...

Figure 4: Extract of a 150% ASG for feature-oriented Java code from GPL.

3.1 Model-Based Representation of Feature-Oriented Code

The analysis method should capture the entire variability and, thus, each occurrence of a design pattern with each possible decomposition. To this end, the analysis has to be performed family-based, i.e., considering all features. Hence, the system representation must capture the entire product line, in contrast to capturing a single product. We decided to develop a new system representation for feature-oriented code, which can express the features as enclosed units in the *Abstract Syntax Tree (AST)* to express variability. In reference to the notion of a 150% model with annotative variability realization mechanisms, we call this a *150% AST* (Schaefer et al., 2012).

We retrieve the 150% AST by parsing the source code of *all* available feature-modules of FOP and forming an AST that contains the respective variabilities for the software family instead of the information for just one system. We realized the 150% AST for feature-oriented SPLs developed in Java by extending JAMOPP[1], the *Java Model Parser and Printer*. JAMOPP is a parser for Java source code that represents parsed code as an EMF[2]-based model. To obtain the 150% AST, extensions to JAMOPP are necessary because feature modules encompass illegal Java code.

[1]http://www.jamopp.org

[2]http://www.eclipse.org/modeling

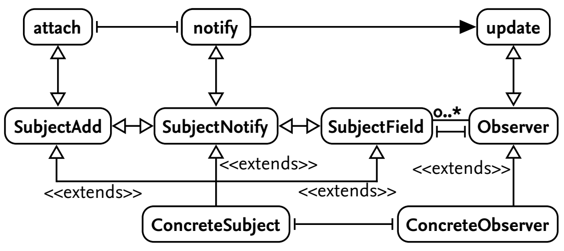


Figure 5: Decomposed role diagram of Observer pattern.

They contain the `original`-call, which is used to realize method refinements in the FOP tool FEATUREHOUSE (Apel et al., 2009). Moreover, reference resolving has to be expanded as feature modules contain *inter-feature references* (i.e., references to classifiers, methods and identifiers located within other features). To facilitate family-based analysis, the 150% AST should represent references to elements as *multi-target references* (i.e., references to classifiers, methods and fields can target multiple declarations). Executing reference resolving creates an *Abstract Syntax Graph (ASG)* from the 150% AST, which we accordingly call 150% ASG. The created 150% ASG captures multiple declarations and refinements of elements when resolving a reference. Furthermore, each class in a feature module should be annotated with its containing feature to capture the variability information. We extended JAMOPP's metamodel, parser and reference resolver accordingly.

Figure 4 illustrates an extract of a 150% ASG from GPL. The differences to a Java ASG are annotated:

(a) A `FopCompilationUnit` encompasses a class within a feature module annotated with the containing feature.
(b) References to classifiers are, e.g., contained in the implements relation of classes. While this relation may capture multiple interfaces, it may now target multiple declarations of one interface.
(c) Elements (i.e., identifier and methods) may be declared multiple times or refined in different features. Hence, references to such elements target multiple declarations in the 150% ASG.

In a nutshell, the devised 150% ASG contains variability information on each class in each feature module, a resolved original-call as well as resolved multi-target references, whose targets may be located in different features.

3.2 Representation of Design Patterns

To automatically detect pattern instances, a representation for design patterns has to be devised that is fine-grained enough to identify technical realizations of a

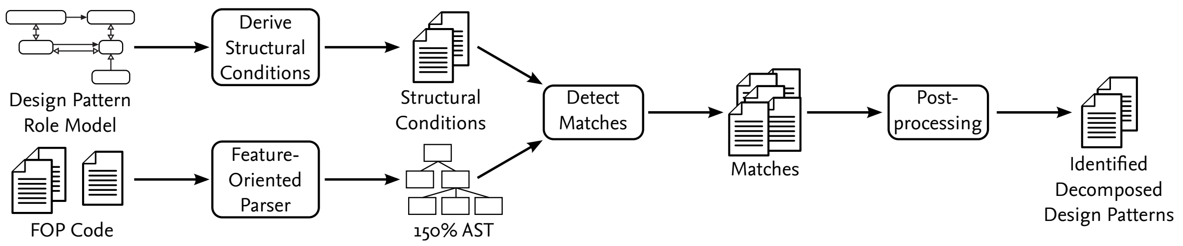


Figure 6: Workflow of the family-based design pattern detection technique.

pattern. (Seidl et al., 2017) introduced a notation based on role modeling to describe the characteristics of design patterns. As it is used to detect decomposed design patterns across features, the maximum possible decomposition of the necessary elements that form a design pattern has to be described. The detection technique uses this notation as input to analyze which elements are applied in a decomposed manner.

In Section 2.2, we described the benefits of using role models to describe design patterns. Using multiple roles, we can further describe the decomposition of a pattern role along multiple features. The equivalence relation between roles denotes that these roles contribute to the same realization artifact. We illustrate the description of decomposed patterns using the Observer pattern as an example. Figure 3 shows the class diagram of the essential elements required to form the Observer pattern (Gamma et al., 1994, p. 294). The names are merely used as identifiers for the different elements.

Figure 5 depicts a *design pattern role model* (DPRM), which describes the maximum decomposition of the elements necessary to form the Observer pattern of Figure 1c. The maximum decomposition of a pattern is derived by reasoning which elements can be introduced in different features. For the `Subject`, e.g., an `attach` method, a `notify` method and an association to the `Observer` are necessary. Those elements manifest in adding methods and fields to the class. In the role diagram in Figure 5, this manifests in the *Subject* being split into the three roles *SubjectAdd* for the `attach` method, *SubjectNotify* for the `notify` method and *SubjectField* for the association to the `Observer`. As these three roles contribute to the same `Subject` class, they are connected by an *equivalence* constraint. In contrast, the associated *Observer* must not contribute to the same realization artifact as specified by the *prohibition*. Finally, concrete implementations of the *Subject* and *Observer* are required, which inherit from (or are the same as) the types playing the *Subject* or *Observer* roles.

In OOP, a role is mapped to an object playing that role. In FOP, roles of a decomposed role model are mapped to class contributions. While in FOP, the definition of a class contribution within a feature is referred to as a *role* (Smaragdakis and Batory, 2002), to avoid terminology conflicts with the role modeling paradigm, we call this a *feature-oriented role*.

3.3 Design Pattern Detection

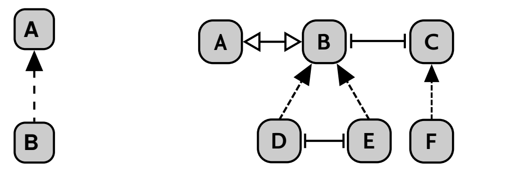
Detecting design patterns is an extensive research topic for OO-languages (Rasool and Streitferdt, 2011). However, none of the existing approaches is tailored to SPLs. We devised a family-based approach, which can be applied to a 150% ASG.

The detection technique is based on that of (Heuzeroth et al., 2003) who developed algorithms to detect the specific structural characteristics of design patterns. In addition, our approach is influenced by the detection technique of (Tsantalis et al., 2006) who use graph pattern matching to identify graph structures that are similar to design patterns.

Figure 6 illustrates the workflow of our detection technique. From the DPRM, we manually derive structural conditions that have to be met in order for a specific class and object collaboration to form parts of a design pattern. These conditions include, e.g., that a feature-oriented role has to contain a specific method or that two feature-oriented roles contribute to the same class. We detect structural matches for a design pattern through graph matching of the structural conditions on the 150% ASG.

We implemented the structural design pattern detection with EMF-IncQuery[3], a tool for high-performance graph searches on EMF models. EMF-IncQuery offers a declarative query language, which we used to encode the derived structural conditions. The query language supports defining imprecise queries, e.g, transitive relations. This is helpful for detecting variants of design patterns. Based on the query a matcher is generated that searches an EMF model for the specified graph structures. This matcher can be run on the 150% ASG of an SPL resulting in a set of candidates for design pattern instances. We further use automatic postprocessing steps (e.g., data and control flow analyses) to eliminate false positives and duplicates from structural matches.

[3]http://www.eclipse.org/incquery



(a) Requires relation. (b) Example of Family Role Model.

Figure 7: Language extension of role modeling and example for Family Role Models (FRMs).

4 DESCRIBING VARIABILITY-AWARE DESIGN PATTERNS

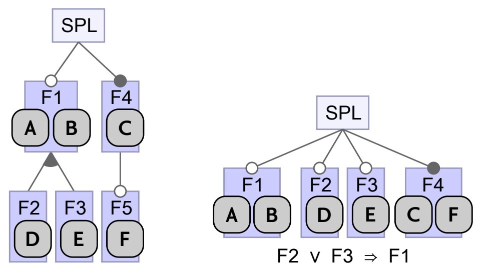
In previous work, a modeling notation to describe design patterns in SPLs, which is based on role models and which is called *Family Role Model (FRM)* was introduced (Schuster, 2014; Seidl et al., 2017).

Feature models describe a *definite design* of variability, while different feature models can express the same variability in multiple ways, i.e., different semantically equivalent feature models can exist (Alves et al., 2006). To describe the feature collaborations of design patterns while disregarding a definite design of a feature model, role modeling is a suitable means. As mentioned in Section 2.2, roles may be mapped to any element, so also mapping to features or partial structures of a feature model is possible.

Family Role Models (FRMs) describe the decomposition of a design pattern across features without depending on a concrete feature model (Schuster, 2014; Seidl et al., 2017). The main idea of FRMs is to consider the maximum possible decomposition of a design pattern in relation to the variability information without depending on a concrete design of a feature model. To this end, one *feature role* for each decomposable element of the DPRM is introduced, i.e., a role that can be played by a feature. The decomposition of the pattern to features is described by role constraints between these feature roles. This decomposition can then be mapped to features. As FRMs describe the decomposition of the pattern, different similar feature models may be represented by the same FRM.

Only specific language constructs are necessary for FRMs. We adopt the *prohibition* (cf. Figure 2c) and the *equivalence* (cf. Figure 2e) constraints. Moreover, we introduce *requires* constraints as illustrated in Figure 7a (Schuster, 2014; Seidl et al., 2017).

Figure 7b depicts an exemplary FRM illustrating the notation. In this example, the roles *A* and *B*, introducing different parts of a pattern, must be played by the same feature. However, this feature must not



(a) Subfeatures. (b) Cross-Tree Constraint.

Figure 8: Equivalent feature models with respect to FRM in Figure 7b.

also play the role *C*. Moreover, *D* and *E* must not be played by the same feature, while either one of the two implies the presence of *B*. Analogously, a feature playing *F* requires a feature playing *C*. Figure 8 shows two example feature models that are concrete manifestations of the FRM illustrated in Figure 7b. Figure 8a fulfills the constraints using subfeatures whereas Figure 8b employs cross-tree constraints.

To describe the decomposition of a design pattern on features using FRMs, an FRM has to be combined with the DPRM. The DPRM describes the maximum decomposition of the design pattern. Using an FRM, we describe the actual, possible decomposition of the pattern. Hence, the maximum decomposition is constrained by conditions that have to be met for a realization within an SPL regarding the distribution of the design pattern's elements to different features. To this end, each role of the role model is annotated with a corresponding feature role.

To create an FRM, the results of an analysis, i.e., the features contributing to detected design pattern instances, are analyzed. The relations between these features are generalized to cover all detected pattern instances and then documented using an FRM.

Figure 9 illustrates the combination of the annotated role model and the FRM for the Observer pattern as well as a possible manifestation in a feature model. In Figure 9a, each role of the decomposed Observer pattern is annotated with a feature role *A–I*. In Figure 9b, feature roles are set into relation in using an FRM. In this case, feature roles *A–G* (i.e., the abstract subject and observer-related roles) are played by the same feature as denoted using the equivalence constraint. In contrast to *H* (the *ConcreteSubject*), *I* (the *ConcreteObserver*) must be played by another feature than *A–G* as denoted using the role prohibition. Moreover, *I* requires features playing *G* and *H*, whereas *H* requires a feature playing the subject-related roles *D–F*. Figure 9c depicts a possible manifestation of the FRM in an exemplary feature model of the tree-structured file system. With this mapping, all constraints defined in the FRM are fulfilled.

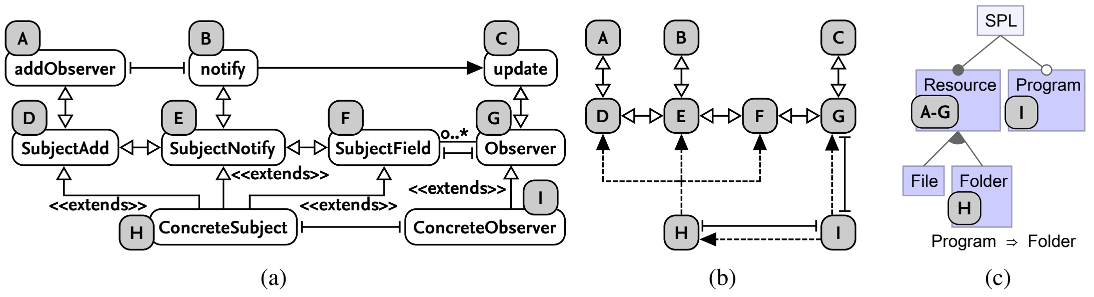


Figure 9: Describing the Observer pattern in variability: (a) DPRM annotated with feature roles, (b) FRM describing detected decomposition constraints and (c) Mapping of feature roles to a possible manifestation in a feature model.

5 CASE STUDY

In this section, we present a case study on variability-aware design patterns. In order to achieve our research goal of analyzing the decomposed application of design patterns, we pose the following research questions:

RQ1: *How are design pattern implementations decomposed along features?*
Design patterns consist of different collaborations that may be implemented in different features. Hence, the study needs to reveal which parts of a pattern are implemented in which features.

RQ2: *What variability-aware application is common for the analyzed design patterns?*
With multiple features being involved in the implementation of a design pattern, we aim at revealing the relationships between these features. Combining all results for a specific design pattern, an FRM can be derived, describing the feature collaborations necessary to implement the pattern.

5.1 Setup & Methodology

We conducted the case study on several feature-oriented SPLs as listed in Table 1, which is a representative set of different sizes and domains. We focused on design patterns that have been identified to be suitable for product line design (Apel et al., 2013). In particular, we analyzed the feature-oriented application of the design patterns *Composite*, *Observer*, *Strategy/Objectifier* and *Template Method*.

We conducted the case study as follows. We started with parsing the source code of a feature-oriented SPL to a 150% ASG. After that, we started the automated pattern detection for a specific design pattern. Although the majority of false positives is eliminated automatically, we reviewed the results manually in order to eliminate each remaining false positive by

Table 1: Overview of the analyzed SPLs.

SPL	#SLOC[1]	#FM[1]	Description
BerkeleyDB[3,4]	44 969	100	database engine
ChatSystem[3]	868	10	chat program
FeatureAMP[2]	2 497	29	audio player
GameOfLife[3,4]	1 466	21	cellular autom.
GPL[3]	1 930	27	graph library
Notepad[3]	1 751	13	text editor
Violet[3,4]	7 470	88	UML editor

[1] SLOC: Source Lines of Code, FM: feature modules
[2] Source: SPL2go – http://spl2go.cs.ovgu.de
[3] Source: Fuji – http://fosd.net/fuji
[4] Refactored from object-oriented legacy system

checking each match. We combined the analysis results of a specific design pattern and, based on these results, derived the FRM by inspecting the distribution of the involved features. This FRM describes the actual decomposition of all detected pattern instances in a generalized fashion.

5.2 Results

Table 2 lists the absolute numbers of detected design patterns in the inspected SPLs. For each design pattern, there are three categories:

- $\sum$ The mere results of the automated pattern detection containing duplicates and false positives.
- $\sum_C$ The number of *correctly identified* design pattern instances after eliminating duplicates and false positives.
- $\sum_D$ The number of *decomposed* design pattern instances that include at least two distinct features contributing to the design pattern implementation.

Table 2 shows that we were able to detect instances of each design pattern. However, in the two smallest SPLs *ChatSystem* and *Notepad*, no patterns were detected. Moreover, a number of false positives

Table 2: Absolute numbers of detected design patterns.

Program Name	Composite			Observer			Strategy			Template Method		
	Σ	Σ_C	Σ_D	Σ	Σ_C	Σ_D	Σ	Σ_C	Σ_D	Σ	Σ_C	Σ_D
BerkeleyDB	12	1	0	2516	–	–	55	–	–	94	11	2
ChatSystem	–	–	–	–	–	–	–	–	–	–	–	–
FeatureAMP	–	–	–	74	7	7	1	–	–	–	–	–
GameOfLife	–	–	–	2	1	1	3	1	1	–	–	–
GPL	–	–	–	12	–	–	–	–	–	1	–	–
Notepad	–	–	–	–	–	–	–	–	–	–	–	–
Violet	5	1	0	133	–	–	7	–	–	12	9	9

The results for the Strategy pattern also contain the results for the structurally equivalent Objectifier pattern.

occurred, especially for the Observer pattern. Nevertheless, the majority of the design patterns detected by our analysis is implemented across multiple features.

As an example, we present the results of the Observer pattern in the SPL *FeatureAMP*, a feature-oriented audio player. In this case, seven decomposed instances of the Observer pattern occurred. All of them are used to notify objects about events concerning the playback of a song, such as the `PlayListener`, which is notified when a song starts.

While all existing `XYZListeners` implement the same `Observer` interface and are registered at the same `Subject`, we regard them as different instances of the Observer pattern because each of them has their own `add/remove` and `notify` methods. The rest of the overall number of detections are duplicate matches of these seven instances. Each of the involved methods only references the `Observer` interface instead of the concrete `XYZListener`, which is why the automated detection cannot distinguish between the methods for the `XYZListeners`.

Figure 10 illustrates an excerpt of the feature model of *FeatureAMP* consisting of all features relevant for the decomposed implementation of the Observer pattern. The features are annotated with the respective roles of the Observer pattern as defined in Figure 5. Many features are participating in implementing the Observer pattern. Furthermore, the roles of the Observer pattern are distributed across the feature model. Listing 1 shows an exemplary implementation of the Observer pattern in FeatureAMP. The roles of the pattern are annotated using comments.

Combining both Figure 10 and Listing 1, the way of decomposing the Observer pattern in FeatureAMP becomes evident. In the base feature *BASE_FEATUREAMP*, an abstract class playing all subject-related roles *SubjectAdd* (*SA*), *SubjectNotify* (*SN*) and *SubjectField* (*SF*) are introduced as well as an interface playing the *Observer* (*O*) role. Hence, the abstractions implementing the basic functionality of the event notification are all introduced within the (mandatory) base feature. In the subfeatures of *FILE_SUPPORT*, *MP3* and *OGG*, two classes playing the *ConcreteSubject* (*CS*) role are introduced, which both inherit from the abstract class playing the subject-related roles. In the feature model in Figure 10, the seven different occurrences of *ConcreteObservers* are denoted using numbers from 1–7. Each number represents an implementation of the common *Observer* interface, while multiple instances of each implementation in different features may exist.

All other decomposed findings for all patterns look similar and only vary subtly. In all cases, the abstractions introducing the general concepts of the pattern are encapsulated within one single feature whereas concrete implementations of these abstractions are decomposed along multiple features. While concrete implementations of the Strategy pattern are encapsulated within a feature group (similar to *ConcreteSubjects*), concrete implementations of the Template Method pattern are distributed across the entire feature model (similar to *ConcreteObservers*).

5.3 Discussion

The results of the case study suggest that design patterns are frequently and, if so, similarly decomposed. In this section, we discuss the results and answer the research questions.

The results shown in Table 2 contain a high number of false positives, especially for the Observer pattern. In cases of the BerkeleyDB and Violet, all detected matches have been identified as false positives by manual elimination. We argue that a high number of false positives results from the structural descriptions of the respective design pattern being too unspecific for static analysis. As a design pattern is mainly described by dynamic object collaborations, the derived structural conditions cannot be too specific because that could eliminate genuine matches. Especially for the Observer pattern, mostly dynamic conditions exist as the pattern is used for object de-

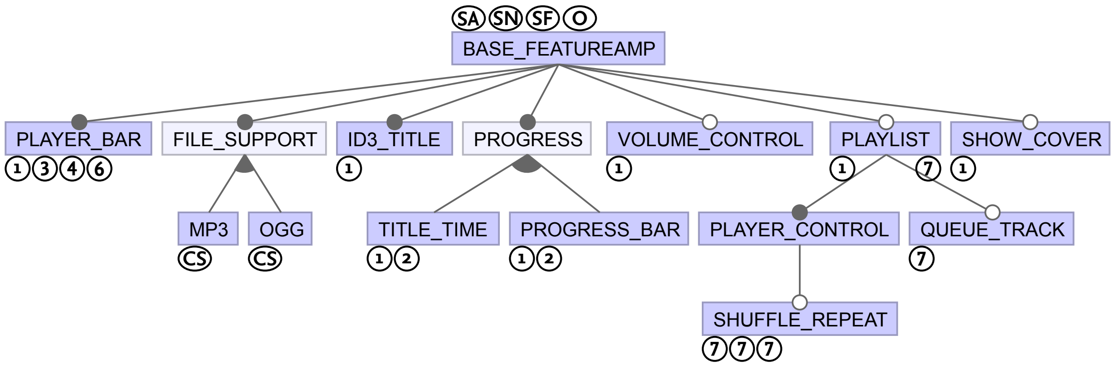


Figure 10: Extract of the feature model of *FeatureAMP* showing all features relevant to the observer patterns. Annotations show the roles of the decomposed observer pattern (cf. Figure 5) they introduce. SA: SubjectAdd, SN: SubjectNotify, SF: SubjectField, O: Observer, CS: ConcreteSubject, Numbers denote the different occurrences of ConcreteObservers.

coupling. Using static analysis, such unspecific structural descriptions result in the majority of matches constituting the same structure as an instance of the respective pattern. However, the semantics of the respective design pattern may not be fulfilled.

We argue that an encapsulation of the abstractions and a decomposition of concrete implementations is reasonable. The different subject roles (i.e., *SubjectField*, *SubjectAdd* and *SubjectNotify*) can be seen as one unit of functionality. Introducing a list of observer objects would not be sensible if it is never used. Furthermore, an observer interface is not sensible without a subject to observe. Hence, in the findings, there is always a feature that encapsulates the entire functionality required to realize event notification for the respective application scenario (e.g., updating the view on model changes). This argumentation holds for other design patterns as well. Decomposing abstract implementations that realize the basic functionality of the pattern is infeasible as they introduce the functionality *together*.

The decomposition takes place for the concrete roles of the pattern – in this case of both subject and observer. While the *ConcreteSubject* in *GameOfLife* is also introduced in one feature together with the *Subject* and *Observer* roles, in *FeatureAMP*, two *ConcreteSubject* roles are introduced in two different features of a mandatory or-group. Hence, if the Observer pattern is introduced, the existence of at least one *ConcreteSubject* role is necessary.

A *ConcreteObserver* does not add to the functionality of the *Subject* but rather to another functionality that depends on the state of a *Subject*. Hence, there might be variants in which no dependent objects exist. Due to this reason, *ConcreteObservers* are introduced in other features than the general event notification functionality. Moreover, many *ConcreteObservers* are enclosed by optional features, especially in *FeatureAMP* (cf. Figure 10). As it depends on the state of a *Subject*, the general event notification has to exist for the *ConcreteObserver* to be introduced.

This argumentation holds for other patterns as well. Decomposing concrete roles of the pattern abstractions is feasible because they either introduce similar, interchangeable behavior (such as *ConcreteSubject*) or they do not contribute to the main functionality but rather extend it by new features that depend on the main functionality (such as *ConcreteObservers*). With these results and the above argumentation, it is possible to answer the research questions.

RQ1: The observer pattern implementation in *FeatureAMP* is distributed across the entire feature model. Although the abstractions (i.e., the *Subject* and *Observer*) are all encapsulated within one single feature, both *ConcreteSubjects* are introduced within different features. Several features introduce *ConcreteObservers* while sometimes introducing more than one. Other design patterns are decomposed in a similar way. Abstractions are always encapsulated by a single feature. Concrete strategies of the Strategy pattern are encapsulated in separate features that are enclosed by a feature group. Concrete implementations of the Template Method pattern are distributed across the entire feature model.

RQ2: The abstractions (i.e., *Subject* and *Observer*) are introduced in the base feature of *FeatureAMP* and mandatory for each possible variant. Using a mandatory *or*-group, the two very similar *ConcreteSubjects* are introduced. This means that at least one of these implementations exists in each possible variant.

While *ConcreteSubjects* are introduced in a feature group, no regularity seems to exist in which *ConcreteObservers* are introduced. Different *ConcreteObservers* are implemented across the entire feature model. The same holds for concrete implementations of the Template Method pattern. We argue, that such

Feature *BASE_FEATUREAMP*

```
public interface Listener<T> { // Observer
  public void update(T object);
}
public abstract class AbstractAudioController
    implements AudioController {
  // SubjectField
  protected LinkedList
   <Listener<AudioController>> playListeners;
  // SubjectAdd
  public void addPlayListener(
      Listener<AudioController> l) {
    this.playListeners.add(l);
  }
  // SubjectNotify
  protected void notifyPlayListeners() {
    for (Listener<AudioController> l :
        this.playListeners) {
      l.update(this);
    }
  }
}
```

Feature *MP3*

```
public class Mp3Controller // ConcreteSubject
    extends AbstractAudioController {
  public void play() {
    /* ... */
    this.notifyPlayListeners();
  }
}
```

Feature *PLAYER_BAR*

```
public class PlayerBar {
  class PlayListener // ConcreteObserver
      implements Listener<Controller> {
    public void update(AudioController a) {
      playButton.setEnabled(false);
      pauseButton.setEnabled(true);
      stopButton.setEnabled(true);
    }
  }
}
```

Listing 1: Observer pattern in FEATUREAMP.

a decomposition results from the functionality added by the features. *ConcreteObservers* may add any kind of functionality and only depend on a subject. They do not extend the subject's functionality. On the other hand, as *ConcreteSubjects*, concrete strategies of the Strategy pattern are introduced in single features encapsulated by one feature group. The reason is that concrete strategies only add to the functionality described by the abstract strategy.

From these results, an FRM can be derived. Figure 9 already depicted the FRM for the Observer pattern, derived from the results of this case study. This FRM encapsulates the abstractions of the Observer pattern in a single feature, whereas the concrete roles are introduced in different features that *depend* on the abstractions being present.

5.4 Threats to Validity

As the results depend on the selection and setup of the case study, in the following, we list threats to validity.

5.4.1 Construct Validity

No formal standard description of the characteristics of a design pattern exists (Dong et al., 2009). Moreover, each pattern can be implemented in a variety of ways. However, we formalized the design patterns according to a general understanding of the patterns adopted from literature.

5.4.2 Internal Validity

We might reject actual instances of design patterns because of too much deviation from our characteristics. The static detection technique depends on the description of design pattern characteristics. Nevertheless, we detected multiple genuine design pattern instances.

5.4.3 External Validity

We only analyzed a small set of SPLs and design patterns, which might impair generalizability and reproducibility of our results. However, we argue that we cover a variety of sizes and different domains. This way, the probability of detecting design patterns and also the generalizability and reproducibility of the results is increased. Moreover, most of the other design patterns do not implement variability. Hence, we argue that with this set of design patterns, we cover common and important variability patterns. Furthermore, our approach is extensible with more patterns.

6 RELATED WORK

With this work, we continue the effort on learning how design patterns affect design in FOP and how the use of FOP affects the implementation of design patterns (Schuster and Schulze, 2012; Schuster et al., 2013; Schuster, 2014; Seidl et al., 2017). We extended this work by introducing a model-based analysis method for detecting variability-aware patterns and employing FRMs for their description.

After (Gamma et al., 1994) introduced design patterns, there has been ongoing research, documenting new patterns, e.g., the *Objectifier* (Zimmer, 1995) or

Extension Objects (Gamma, 1996) patterns. Moreover, the application of patterns (Beck et al., 1996) and their relationships (Zimmer, 1995) have been analyzed. Furthermore, design patterns have been described as role models (Riehle, 1996), however, not for decompositions in SPLs. We contribute to the overall research on design patterns by documenting feature-oriented variants of patterns that are decomposed across multiple features.

(Hannemann and Kiczales, 2002) already modularized design patterns in the context of ASPECTJ (Kiczales et al., 2001), an extension to Java allowing Aspect-Oriented Programming (Kiczales et al., 1997). They reasoned on the modularization of design patterns, pointing out potential benefits compared to standard pattern implementations. However, they concentrated on cross-cutting characteristics of design patterns and did not investigate the actual usage of decomposed pattern instances.

(Kolesnikov, 2011) developed FUJI, a fully-fledged compiler and AST for feature-oriented JAVA code. FUJI annotates each element with information on which feature *introduces* the element, but not which features *refine* it. Our 150% ASG does not compose the SPL but rather parses the separate feature modules and resolves all references to elements defined in other features. Hence, the 150% ASG facilitates family-based analysis.

(Kästner et al., 2011) also perform variability-aware parsing of realization artifacts. However, they focus on parsing realization artifacts containing annotative variability, whereas we focus on compositional approaches. These approaches face elementary different challenges: (Kästner et al., 2011) parse 150% representations that contain elements and references for all possible variants. In contrast, we parse different feature modules whose references have to be resolved while respecting variability.

Extensive research exists on the topic of design pattern detection (Dong et al., 2009; Rasool and Streitferdt, 2011). For example, (Heuzeroth et al., 2003) capture the specific characteristics of design patterns that describe the minimal structural requirements. (Tsantalis et al., 2006) use graph pattern matching combined with a similarity scoring algorithm to identify graph structures that are exact representations or similar structures of a specific design pattern. As no design pattern detection for SPLs exists, we developed a static design pattern detection technique based on both approaches tailored to SPLs.

7 CONCLUSION

In this work, we presented and applied a model-based analysis method to determine the variability-aware usage of design patterns in the context of the source code of FOP-based SPLs. Moreover, we introduced *Family Role Models (FRM)* as a modeling notation based on role modeling, which can be used to constrain the decomposed usage of design patterns. The main observation of the conducted case study is that abstract parts of design patterns, which introduce the general concept of the pattern, are always introduced within one single feature. In contrast, concrete parts of design patterns, such as concrete implementations of Observers or Strategies, are often decomposed along features. For the *Observer* and *Template Method* patterns, features introducing concrete parts are distributed across the entire feature model, whereas, for the *Strategy* (and *Objectifier*) pattern, concrete parts are introduced in one feature group.

A decomposition of specific design patterns appears to increase modularity and reuse by decoupling specific implementations from abstraction. Fine-grained customization is allowed by exploiting variability offered by design patterns.

In the future, our efforts in variability-aware pattern mining may produce more insights into common practice of realizing SPLs using design patterns. Based on this, guidelines for best practices in implementing SPLs using modularized patterns may be derived. Moreover, entirely new design patterns may be revealed, dedicated to realizing variability.

REFERENCES

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring Product Lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 201–210, New York, NY, USA. ACM.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer.

Apel, S. and Beyer, D. (2011). Feature Cohesion in Software Product Lines: An Exploratory Study. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 421–430. IEEE.

Apel, S., Kastner, C., and Lengauer, C. (2009). FEATUREHOUSE: Language-Independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231, Washington, DC, USA. IEEE Computer Society.

Apel, S. and Kästner, C. (2009). An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371.

Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J. O., Dominick, L., and Paulisch, F. (1996). Industrial Experience with Design Patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society.

Clements, P. and Northrop, L. (2001). *Software Product Lines – Practices and Patterns*. Addison-Wesley.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Dong, J., Zhao, Y., and Peng, T. (2009). A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855.

Gamma, E. (1996). The Extension Objects Pattern. In *Proceedings of the 1996 Conference on Pattern Languages of Programs*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hannemann, J. and Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. *ACM SIGPLAN Notices*, 37(11):161–173.

Heuzeroth, D., Holl, T., Hogstrom, G., and Lowe, W. (2003). Automatic Design Pattern Detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. *SIGPLAN Not.*, 46(10):805–824.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An Overview of AspectJ. In Knudsen, J., editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP'97 — Object-Ori-ented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg.

Kolesnikov, S. (2011). An Extensible Compiler for Feature-Oriented Programming in Java.

Kästner, C., Apel, S., and Ostermann, K. (2011). The Road to Feature Modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 5:1–5:8, New York, NY, USA. ACM.

Pohl, K., Böckle, G., and Van Der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look At Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer.

Rasool, G. and Streitferdt, D. (2011). A Survey on Design Pattern Recovery Techniques. *IJCSI International Journal of Computer Science Issues*, 8(2):251–260.

Reenskaug, T., Wold, P., and Lehne, O. A. (1996). *Working with Objects: The OOram Software Engineering Method*. Manning Greenwich.

Riehle, D. (1996). Describing and Composing Patterns using Role Diagrams. In *White Object-oriented Nights: Proceedings of the 1st International Conference on Object-Oriented Technology*, volume 96.

Riehle, D. (1997). A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. Technical report, Ubilab Technical Report 97.1. 1. Zürich, Switzerland: Union Bank of Switzerland.

Riehle, D. and Gross, T. (1998). Role Model Based Framework Design and Integration. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 117–133, New York, NY, USA. ACM.

Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., and Villela, K. (2012). Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495.

Schuster, S. (2014). *Pattern-Based Software Product Line Design using Role Modeling*. Diploma thesis, Technische Universität Braunschweig.

Schuster, S. and Schulze, S. (2012). Object-Oriented Design in Feature-Oriented Programming. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 25–28, New York, NY, USA. ACM.

Schuster, S., Schulze, S., and Schaefer, I. (2013). Structural Feature Interaction Patterns: Case Studies and Guidelines. In *Proceedings of the 8th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 14:1–14:8, New York, NY, USA. ACM.

Seidl, C., Schuster, S., and Schaefer, I. (2017). Generative Software Product Line Development using Variability-Aware Design Patterns. *Computer Languages, Systems & Structures*, 48(Supplement C):89 – 111. Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15).

Smaragdakis, Y. and Batory, D. (2002). Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. (2006). Design Pattern Detection Using Similarity Scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909.

Zimmer, W. (1995). Relationships between Design Patterns. *Pattern Languages of Program Design*, 1:345–364.