Object-Oriented Design in Feature-Oriented Programming

Sven Schuster
TU Braunschweig
Braunschweig, Germany
s.schuster@tu-bs.de

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sanschul@tu-braunschweig.de

ABSTRACT

Object-oriented programming is the state-of-the-art programming paradigm for developing large and complex software systems. To support the development of maintainable and evolvable code, a developer can rely on different mechanisms and concepts such as *inheritance* and *design patterns*. Recently, feature-oriented programming (FOP) gained attention, specifically for developing software product lines (SPLs). Although FOP is an own paradigm with dedicated language mechanisms, it partly relies on object-oriented programming. However, only little is known about feature-oriented design and *how* object-oriented design mechanisms and design principles are used within FOP. In this paper, we want to raise awareness on design patterns in FOP and stimulate discussion on related topics. To this end, we present an exemplary review of using OO design patterns in FOP and limitations thereof from our perspective. Subsequently, we formulate questions that are open and that we think are worth to discuss in the context of feature-oriented design.

Categories and Subject Descriptors

H.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*inheritance,patterns*

General Terms

Languages

Keywords

design pattern, feature-oriented programming

1. INTRODUCTION

When developing software systems, an extensible and reusable design is crucial for the durability and maintainability of the system. To achieve such a clear and maintainable structure, different mechanisms and design principles exist, depending on the used programming paradigm. For *object-oriented programming (OOP)*, abstraction and information hiding play a pivotal role for the foundation of a clear design. On the technical side, *inheritance* but also *interfaces* are mechanisms that provide the developer with capabilities to realize different levels of abstractions. Additionally, object-oriented *design patterns* exist to provide general solutions for complex, recurring problems with [6].

While this is the state-of-the-art for complex, stand-alone software system, the concept of *software product lines (SPL)* gained momentum in recent years [4, 9]. Different approaches exist to implement software product lines, which can be divided in two categories: *annotative* and *compositional* [7]. In this paper, we focus on the emerging paradigm of *feature-oriented programming (FOP)*, a compositional approach that extends OOP by providing reuse facilities for building product lines at large-scale. Although FOP distinguishes from OOP by specific mechanisms such as *refinements* for implementing software product lines, a clear and evolvable design is crucial for both approaches, FOP and OOP.

For OOP, well-established design mechanisms (inheritance, interfaces) and concepts (design patterns) exist while for FOP only little is known about design issues. However, we argue that object-oriented design mechanisms and concepts, especially design patterns, can be applied to FOP as well, because of related concepts between FOP and OOP. This, in turn, inevitably leads to several questions: Do we apply OO design patterns within FOP already (but rather implicitly than on purpose)? Is there a way to make design decisions such as usage of design patterns explicitly in FOP? Are OO design patterns applicable to FOP? What are limitations? And are there dedicated feature-oriented design patterns?

With this position paper, we want to stimulate the discussion on these (and maybe forthcoming) questions, because we believe that they are important for future work on feature-oriented design *and* languages. To this end, we provide a review on using OO design patterns in FOP by means of different examples. Furthermore, we point out limitations that we observed during our review.

In a broader sense, this paper also contributes to an ongoing discussion on modularity and design in FOP [8]. In this context, we stimulate discussion on the question whether dedicated feature-oriented design patterns are needed to ensure an evolvable and maintainable feature-oriented design.

Limitations: With this paper, we do not present fully-fledged and finished research results. Rather, we want to raise awareness on the role of feature-oriented design and its relation to object-oriented design (patterns). Further-

FOSD'12, September 24–25, 2012, Dresden, Germany.

more, we focus on a specific feature-oriented approach called FeatureHouse. Finally, we rely on the exemplary design patterns presented by Gamma et al. [6], although other realizations of these patterns are possible.

2. BACKGROUND

In this section we will provide a short background on object-oriented design patterns and the paradigm of feature-oriented programming.

2.1 Object-Oriented Design Patterns

During design and implementation, it is common that certain recurring problems emerge, which have to be solved without decreasing maintainability or reusability. A *design pattern* is a textual description for such a common problem and its possible solution [6]. Following principles for "good" object-oriented design, patterns aim at improving the structure of a program and increasing reusability and maintainability of the source code by making it more flexible and more adaptable to changes. Examples for such design principles that are reflected by patterns are:

- favor object composition over inheritance
- program to an interface, not to an implementation
- encapsulate what varies

While different possibilities exist to realize design patterns, we here focus on the implementation and representation (using UML class diagrams) originally proposed by Gamma et al. [6]. Examplary, we illustrate the *Strategy* pattern [6, p. 315 ff.] by means of a class diagram in Figure 1. This pattern takes a family of algorithms and makes them interchangeable by defining an abstract strategy interface. In this pattern, the class `Context` holds an object of type `Strategy`, which provides the interface to be used. This object can be replaced with other objects of the same type, resulting in an interchangeable algorithm for the defined interface.

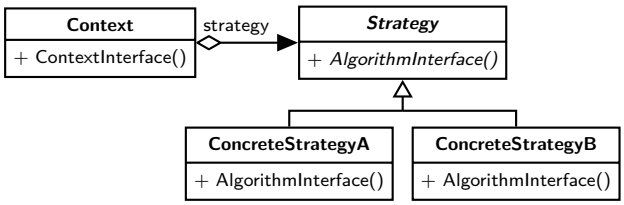

Figure 1: Class diagram of *Strategy* pattern

Design patterns are classified by their purposes into three categories of patterns: creational, structural and behavioral patterns. Creational patterns describe when and how objects are instantiated such as the *Factory Method* [6, p. 107 ff.], which encapsulates and simplifies the creation of similar objects. The main concern of structural patterns is the composition of classes or objects, like the *Facade* [6, p. 185 ff.], which hides the structure of a subsystem behind a new, simplified interface. Finally, behavioral patterns deal with the interaction between objects and provide dynamic behavior at runtime, like the aforementiond *Strategy*.

2.2 Feature-oriented Programming

Feature-Oriented Programming (FOP) is a paradigm to implement *software product lines (SPL)* in a compositional way [10]. Different approaches and languages exist to implement feature-oriented software product lines such as AHEAD [3], FeatureHouse [1], or FeatureC++ [2]. The core idea of FOP is to decompose a program into features. All artifacts (code and non-code) belonging to a certain feature are modularized within one cohesive unit, called *feature module*. A feature is an increment in functionality, visible to any stakeholder. A *feature model* describes commonalities and differences between the different programs of a product line and thus possible and *valid* combinations of features. Due to its modular fashion, FOP provides a one-to-one mapping between its implementation units (i.e., feature modules) and the features of a feature model.

Feature *BaseStack*

```
class Stack { ...
  void push(int v)
       {/*...*/}
  int pop() {/*...*/}
}
```

Feature *Peak*

```
class Stack {
  int peak() {/*...*/}
}
```

Feature *Undo*

```
class Stack { ...
  int backupPush;
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    original(v);
}}
```

Composed class

```
class Stack { ...
  int backupPush;
  int pop() {/*...*/}
  int peak() {/*...*/}
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    /* original */
}}
```

Figure 2: Feature-oriented implementation of *Stack* with features *Peak* and *Undo*

In Figure 2, we show an excerpt of a stack product line implementation with FeatureHouse [1], a language-independent approach for FOP, which uses *superimposition* as its composition mechanism. Feature *BaseStack* provides the base implementation of class `Stack`. The two other features, *Peak* and *Undo* extend the functionality of this class. In the context of FOP, this extension or increment of functionality is called *refinement*. Basically, refinements offer the possibility to add or extend classes, for instance, by adding new methods or fields or changing existing ones. Methods can be composed using a specified keyword (`original` in FeatureHouse) to access an already existing method body. As an example, feature *Undo* extends method `push` by adding an additional statement followed by the `original` keyword, which invokes method `push` of the original class `Stack`. Feature *Peak* simply adds the method `peak`. To generate a program, the selected features (i.e., the corresponding source code) is composed using superimposition. For instance, if a user selects features *BaseStack*, *Peak* and *Undo* results into class `Stack` with four methods (`push, pop, peak, undo`) and one field.

3. COMPARING OBJECT-ORIENTED AND FEATURE-ORIENTED DESIGN

Object-oriented design mechanisms and patterns are well-understood and commonly accepted as a mean to achieve a clear and maintainable design. FOP partly relies on object-oriented concepts and mechanisms. This raises the question, *how* and *where* both approaches consolidate, especially regarding the design of the underlying programs. In this section, we present some initial thoughts on that question. In particular, we compare and contrast inheritance and refinements and discuss whether (and how) object-oriented design patterns could be applied in feature-oriented programming.

3.1 Inheritance versus Refinements

While OOP offers class inheritance as the main language mechanism to gain variability and abstraction in software design, FOP additionally offers class refinements to achieve feature modularity. In the following, we will distinguish these mechanisms.

Both, inheritance and refinements, are mechanisms to achieve code reuse and to extend classes, but beyond that, they do not have much in common. In Table 1, we provide a short distinction of both mechanisms.

Inheritance ...	Refinements ...
... creates a new subclass to extend a class	... extend the original class itself
... achieves variability at runtime	... achieve variability at compile time
... is integrated within the language	... are not integrated within the language

Table 1: Inheritance versus Refinements

```
class Stack { ...
  void push(int v)
       {/*...*/}
  int pop() {/*...*/}
}
```

```
class PeakStack
  extends Stack{
  int peak() {/*...*/}
}
```

```
class UndoStack
  extends Stack { ...
  int backupPush;
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    super.push(v);
}}
```

```
class UndoPeakStack
  extends PeakStack { ...
  int backupPush;
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    super.push(v);
}}
```

Figure 3: Object-oriented implementation of *Stack* with features *Peak* and *Undo*

We illustrate the differences between inheritance and refinement with two code examples in Figure 2 and 3, respectively. The feature-oriented implementation of *Stack* consists of only one class that is refined in each feature module (cd. Figure 2). Hence, for a certain variant, only one *composed* class exists, which contains the whole functionality of the selected features. In contrast, in our object-oriented implementation of *Stack*, we have to introduce a new class for every feature and every combination of features, resulting in four different classes (cf. Figure 3). In a nutshell, extending a class with inheritance always leads to a new subclass, while refinements extend the original class itself.

Another difference between both mechanisms is their integration within the language and their scope. Inheritance is a language mechanism, which can be used to achieve varying behavior at runtime by creating subtypes and providing interchangeability between objects. In our example, all variants of *Stack* are interchangeable, since they are subtypes of the same superclass. In contrast, refinements disappear when composing the feature modules at compile time. Hence, they allow for selecting which features and thus which refinements should be included for a certain variant *before* this variant is generated. Overall, inheritance and refinements can be seen as two different, orthogonal dimensions, which are rather complementing than contradicting.

3.2 Design Patterns in FOP

Since FOP and OOP share some language mechanisms, object-oriented design patterns should be applicable in feature-oriented SPLs. Furthermore, refinements should not contradict with language mechanisms used for design patterns such as inheritance or interfaces, for the previously mentioned reasons. Hence, we argue that we can use refinements to modularize design patterns in terms of features. In the following, we present examples how design patterns could be extended or modified using refinements.

Feature *Foo*

```
class Factory {
  Product createProduct(int id) {
    if(id == FOO)
      return new Foo();
}}
```

Feature *Bar*

```
class Factory {
  Product createProduct(int id) {
    if(id == BAR)
      return new Bar();
    else return original(id);
}}
```

Figure 4: *Factory Method* extended by new Products using FeatureHouse

In Figure 4, we show an example for creational patterns in FOP. In particular, we apply a refinement to a variant of the *Factory Method* (cf. Section 2.1) by providing the method `createProduct(int id)` with feature module *Foo* and using refinements to add new products. Hence, we offer the possibility of creating products of type *Bar* only if the feature module *Bar* is included. Moreover, new factory methods or whole new factories with their respective products can be introduced with new feature modules. In the same way, other creational patterns can be refined as well. For instance, the *Prototype* pattern [6, p. 117 ff.] can be extended using a feature module that adds new prototypes to a list of prototypes.

Structural design patterns, e.g., *Facade* (cf. Section 2.1), are great examples for the benefits of combining patterns of OOP with FOP. The Facade pattern hides a whole subsystem behind a simplified interface. As a result, we may use refinements to modify or extend everything within the subsystem, without interfering any other class, as long as the interface is not modified.

Behavioral design patterns such as the *Strategy* pattern (cf. Section 2.1, Figure 1), can be extended by new strategies via features. In Figure 5, we show the Strategy in *Violet*[1], which is combined with the Prototype. While the abstract strategy class `Graph` offers the interface to grant access to the different prototypes for nodes (and edges), the concrete strategies like `ClassDiagramGraph` provide the corresponding prototypes. Since *Violet* is refactored in a very fine-grained manner, every prototype is included in its own feature module. Hence, the feature module *InterfaceNode* introduces the prototype for an interface node (cf. Figure 6). This leads to a one-to-one mapping of features and strategies as well as features and prototypes. We can even modularize more complex behavioral patterns using refinements. For example, in the *Observer* pattern [6, p. 293 ff.], the registration

[1] `source code on www.fosd.de/fh`

of different observers could be performed in different feature modules.

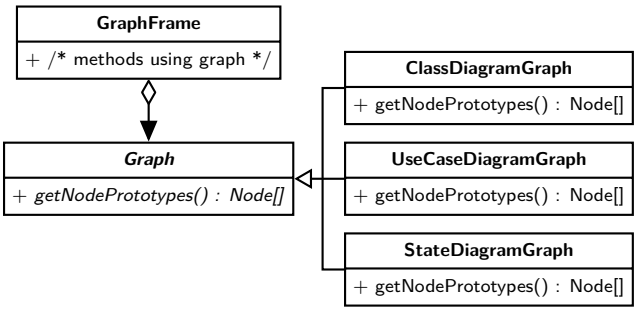

Figure 5: Strategy pattern in Violet

Feature *InterfaceNode*

```
public class ClassDiagramGraph {
  static {
    NODE_PROTOTYPES[1] = new InterfaceNode();
}}
```

Figure 6: Introducing an interface node in *Violet*

Since refinements are a structural mechanism, we cannot expect to change any dynamic behavior of the OO patterns. Hence, we argue, even though we are able to change the behavior of design patterns in a certain way by using refinements, we only gain advantages on a structural level.

3.3 Design Pattern in FOP – Use or Refuse?

Based on the review of OO design patterns and some initial insights on feature-oriented programs, we briefly address the questions that we posed at the beginning of this paper. For a more comprehensive overview, we refer to [12].

Do we already apply OO design patterns in FOP? Recently, we conducted a preliminary analysis on design patterns in feature-oriented programs [12]. As a result, we detected design patterns throughout all programs, regardless whether they have been refactored or developed from scratch. Hence, we argue that design patterns are already in use with FOP. Nevertheless, a more comprehensive and quantitative analysis is necessary to make claims regarding how and which patterns are used.

Are OO design patterns applicable in FOP? Based on our review and preliminary analysis of feature-oriented programs, the answer is yes. However, it is open which pattern fit very well with FOP and which do not. Furthermore, how concrete implementations look like for different feature-oriented languages has to be investigated. Another point, even discussed for OO languages, is the question whether design patterns are always beneficial or might even introduce drawbacks [5]. For instance, Smaragdakis et al. compare *mixin layers*, another approach for realizing compositional SPLs, with the *Visitor* pattern and point out certain characteristics where mixins are more advantageous than the visitor pattern [13].

What are limitations? From our perspective, applying behavioral patterns is limited, because these patterns focus mainly on changing behavior at runtime. Although it has been proven by Rosenmüller et al. that such patterns can be used to support dynamic binding [11], it is generally a very complex task and maybe only possible for certain languages. Furthermore, implementing design patterns with features as an additional dimension could be a complex task, especially from a programmer's comprehension point of view.

4. CONCLUSION AND FUTURE WORK

Design patterns describe recurring problems (and its solution) in object-oriented design. While there is a considerable body of knowledge on design patterns in OOP, only little is known about design patterns in FOP. In this paper, we addressed this topic and reviewed exemplary (OO) design patterns from a feature-oriented point of view. We hav show by example, that design patterns are applicable, but also point to possible limitations and open questions on benefits and application of patterns in FOP.

While the main contribution of this paper is to raise awareness and stimulate discussion, we determined open questions during our review of design patterns in FOP that can guide future research on this topic. In future, we want to analyze existing feature-oriented systems with respect to the occurrence of design patterns to determine whether design patterns are already used in FOP. Furthermore, the concrete realization of design patterns across different feature-oriented languages is part of our future work.

5. REFERENCES

[1] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. ICSE*, pages 221–231. IEEE Computer Society, 2009.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. GPCE*, pages 125–140. Springer-Verlag, 2005.

[3] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

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

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. ICSE*, pages 311–320. ACM Press, 2008.

[8] C. Kästner, S. Apel, and K. Ostermann. The Road to Feature Modularity? In *Proc. FOSD*, pages 5:1–5:8. ACM, 2011.

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

[10] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. ECOOP*, pages 419–443. Springer, 1997.

[11] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proc. GPCE*, pages 3–12. ACM, 2008.

[12] S. Schuster. Design Patterns in Feature-Oriented Programming. Bachelor thesis, TU Braunschweig, 2012.

[13] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-based Designs. *TOSEM*, 11:215–255, 2002.