On the Duality of Aspect-Oriented and Feature-Oriented Design Patterns

Martin Kuhlemann, Marko Rosenmüller, Sven Apel
School of Computer Science, University of Magdeburg
P.O. Box 4120
39016 Magdeburg, Germany
{kuhlemann,rosenmueller,apel}@iti.cs.uni-magdeburg.de

Thomas Leich
METOP Research Institute
Sandtorstrasse 23
39106 Magdeburg, Germany
thomas.leich@metop.de

ABSTRACT
Design patterns aim at improving reusability and variability of object-oriented software. Despite its success, aspect-oriented programming (AOP) has been discussed recently to improve the design pattern implementations. In another line of research it has been noticed that feature-oriented programming (FOP) is related closely to AOP and that FOP suffices in many situations where AOP is commonly used. In this paper we explore the assumed duality between AOP and FOP mechanisms. As a case study we use the aspect-oriented design pattern implementations of Hannemann and Kiczales. We observe that almost all of the 23 aspect-oriented design pattern implementations can be transformed straightforwardly into equivalent feature-oriented design patterns. For further investigations we provide a set of general rules how to transform aspect-oriented programs into feature-oriented programs.

General Terms
DESIGN, LANGUAGES

Categories and Subject Descriptors
D.1.5 [PROGRAMMING TECHNIQUES]: Object-oriented Programming; D.3.3 [Language Constructs and Features]: Patterns

Keywords
Design patterns, object-oriented programming, aspect-oriented programming, feature-oriented programming

1. INTRODUCTION
Design patterns are a well known and accepted approach to implement variable and reusable software with object-oriented programming (OOP) [9]. Despite its success in software development Hannemann et al. observed a lack of modularity, composability and reusability in the respective object-oriented designs [14]. They trace this lack to the presence of crosscutting concerns. Crosscutting concerns are design and implementation problems that result in code tangling, scattering, and replication of code when a software is decomposed along one dimension [24], e.g., the decomposition into classes and objects in OOP. To overcome this limitation several advanced modularization techniques have been proposed, amongst others aspect-oriented programming (AOP) [17] and feature-oriented programming (FOP) [21]. Both paradigms provide mechanisms to modularize crosscutting concerns. While AOP is based on aspects, advice and inter-type declarations, FOP is based on collaboration design and refinements.

Although there are several success stories of AOP in general [11, 15, 12, 28] and of aspect-oriented design patterns [14, 8], there are many voices that criticize certain modularization mechanisms of AOP [11]. For example, Steimann observed the lack of modularity in aspect-oriented designs due to missing interfaces [23]. Touwé et al. discovered that AOP prevents the reapplication of aspects to a software thus causing an evolution paradox [25]. Another example is the arranged pattern problem described by Gymbel and Brichau [13]. FOP is known to avoid these problems because FOP mechanisms (e.g., mixin composition and collaboration-based design [22, 26]) are more simple and suffice in many situations [2, 3, 19, 18, 4, 6]. Hence, we want to explore if AOP programs can be transformed into FOP programs and if there are benefits regarding complexity.

We show that AOP and FOP address similar issues of software development (i.e., crosscutting concerns) but in different ways. We do not focus on the evaluation of AOP and FOP with respect to OOP nor do we focus on the complexity of specific designs using metrics but we show the duality of AOP and FOP mechanisms. We use the implementation of the GoF design patterns as a case study. Hannemann et al. showed that an aspect-oriented design of the 23 Gang-of-Four design patterns [17] improves modularity, composability, and reusability [14]. They detached code belonging to crosscutting concerns and encapsulated it into aspects.
In this paper we demonstrate the possibility to implement the aspect-oriented design patterns of Hannemann et al. in a collaboration-based and feature-oriented fashion. Due to the duality of some aspect-oriented and feature-oriented modularization mechanisms we were able to transform the implementations for 19 of the 23 aspect-oriented design patterns straightforwardly into feature-oriented counterparts. We discuss our experiences regarding this transformation and illustrate the similarities and differences of the aspect-oriented and feature-oriented solutions.

2. BACKGROUND

In this section we introduce the analyzed techniques AOP and FOP.

2.1 Aspect-Oriented Programming

The purpose of AOP is to modularize software regarding different and independent concerns simultaneously [17]. In AOP the code of crosscutting concerns of an underlying object-oriented structure is encapsulated into aspects. AspectJ is a popular AOP language. We explain its mechanisms in the following paragraphs.

Pointcut and Advice.

The primary mechanism of AOP is the extension of events occurring at runtime, so-called join points [20]. The static representation of a runtime event is called join point shadow. Join point shadows are for example statements of method calls, object creation, or member access. A pointcut defines a set of join points to be advised.

An example for pointcut and advice (PCA) is given in Figure 1. The aspect MyAspect (Lines 11–22) extends the classes Label and Button. The pointcut LabelChangeCall (Line 12) refers to all statements calling methods of the class Label (call pointcut), e.g., call statements for the method setText. Hence, the according piece of advice of Line 14 (before advice) is introduced into the method click of the class Button before the method Label.setText is invoked (Line 7). Advice can also be applied after (after advice) or around (around advice) a join point.

The advice of the pointcut LabelChangeExec (Line 15) refers to the body (execution advice) of the method setText, i.e., the advice is introduced into the method setText of class Label (Line 2).

While pieces of call advice, e.g., LabelChangeCall, are invoked only for referred calls of the method, execution advice, e.g., LabelChangeExec, is applied every time the method is called.

Inter Type Declaration.

Inter type declarations (ITD) are methods or fields that are inserted into OOP classes by an aspect and thus become members of these classes. Additionally, interfaces can be extended with methods and fields. Contrary to Java conventions, AspectJ allows to introduce methods including a method body into interfaces.

In our example of Figure 1 the aspect MyAspect defines two ITD to insert the field Name (Line 17) and the new method printName (Line 18) into a class Creator.

Aspect Fields and Methods.

Aspects can contain members similar to members of a class, i.e., aspects can contain methods, fields, or inner classes and interfaces. These aspect members can be invoked from code inside the aspect, e.g., by advice, but also from external classes. The aspect MyAspect includes one aspect field and one aspect method (Fig. 1 Lines 20–21).

If aspect fields and methods (AFM) are invoked from advice, ITD, or from the classes (using the aspect method aspectOf), and no extra declarations are declared (e.g., per-cflow), then every reference to aspect members refers to the same single aspect instance, thus the aspect is instantiated once.

Parent Declaration.

Aspects can declare a class to implement additional interfaces. Furthermore, aspects can declare a class to inherit from additional classes.

Other AOP.

If a user defined constraint is violated by the classes, the aspect weaver can be instructed to invoke compiler warnings or compiler errors. Precedence declarations define the ordering of advice if join point shadows are advised by more than one aspect.

2.2 Feature-Oriented Programming

FOP aims at feature modularity in software product lines where features are increments in program functionality [21]. Typically, features are not implemented through a single class but through different classes [21]. To add a feature subsequently means to introduce code into existing classes. Code of different classes associated to one feature are merged into one feature module. Feature modules refine other feature modules in a stepwise manner, that is, each refinement superimposes the feature modules already assembled [7].

```java
public class Label {
    public void setText(){
        myLabel.setText("Button clicked");
    }
}

public class Button {
    public void click() {
        myLabel.setText("Button clicked");
    }
}

public aspect MyAspect {
    protected pointcut LabelChangeCall(): call(* Label.*(..));
    protected pointcut LabelChangeExec(): execution(* Label.*(..));
    before():LabelChangeCall();
    before():LabelChangeExec();
    public String Creator.Name;
    public void Creator.printName();
    public Map printer;
    public void getPrinter();
}
```

Figure 1: Application of call and execution advice in AOP.
public class Label {
  public void setText() { /*...*/
}

refines class Label {
  public void setText() {
    Super().setText();
  }
  public void getText() { /*...*/
}

Figure 2: Refinement of a method by a FOP refinement.

We systematize the mechanisms of the AHEAD Tool Suite\(^1\), a popular FOP language extension for Java, into the categories of Mixins, Method Extensions, and Other FOP. Additionally, we will describe the OOP technique of Singleton classes as a category since we used singleton classes to transform AFM into FOP.

Mixins.

Feature modules may manipulate properties of classes, e.g., introduce new methods or fields. In Figure 2 the new method `getText` is introduced by the feature module Extension (Line 10).

Method Extension.

FOP allows to extend methods of classes by overriding. An example is depicted in Figure 2. The feature module `BASE` (Lines 1–4) includes a class `Label` that is extended by the refinement `EXTENSION` (Lines 5–11), i.e., the refinement `EXTENSION` superimposes the method `setText` of the class `Label`. The method `setText` of the feature module `EXTENSION` extends the `setText` method of the feature module `BASE` by invoking this superimposed method (using `Super`, Line 8).

Singleton.

A singleton class is an idiom to limit the number of objects of a class. The singleton class is instantiated once and all subsequent requests to this class are forwarded to the unique object \(^2\).

Other FOP.

All classes, that are not nested in other classes are encapsulated in feature modules. The ordering of feature modules defines the ordering of refinements for one single method. Class members qualified as limited visible, e.g., `protected`, cannot be accessed from classes others than the subclasses of the encapsulating class.

3. DUALITY OF AOP AND FOP MECHANISMS

In our analysis we observed that different mechanisms available in AOP and FOP are similar although looking differently:

- The AOP mechanism of ITD is equivalent to the FOP mechanism of Mixins. Both mechanisms extend OOP classes with additional methods and fields.
  - The AOP mechanism of execution advice is equivalent to the FOP mechanism of Method Extension. Both mechanisms extend the body of a method for all subsequent calls.
  - The mechanism of Parent Declaration is representable straightforwardly by the Mixin mechanism due to the possibility of mixins to manipulate the inheritance hierarchy of refined classes.
  - Aspects can be instantiated and referred to from the OOP classes they extend. The method `aspectOf` of an aspect allows to manipulate one instance of the aspect, i.e., different methods referring to one aspect using `aspectOf` work on the same aspect instance.
    This mechanism is representable straightforwardly by the OOP Singleton design pattern \(^2\).

We want to explore the duality of AOP and FOP to get more insight into the used implementation mechanisms. It has been observed that FOP mechanisms suffice frequently while AOP introduce additional complexity \(^2\) \(^3\) \(^9\) \(^18\) \(^22\). Thus we present a set of rules that reflect the duality and describe how to get from an AOP to an FOP implementation and vice versa.

4. TRANSFORMATION METHODOLOGY

In this section we give 7 rules to transform AOP programs into FOP programs.

PCA into Method Extensions.

We propose to transform execution advice into method extensions. If the pointcut of a call advice is not restricted and does not refer to the calling object we propose to transform the call advice into a method extension of the called method. This is possible because this call pointcut is equivalent to an execution pointcut.

For the transformation of PCA, the position of the invocation of the extended method in FOP (Super, Line 8) depends on the kind of advice:

- A `before advice` demands for the invocation of the refined method to be the last statement of the refining method in FOP.
- `After advice` has to be transformed into method extensions while the invocation of the superimposed method is the first statement of the superimposing method.
- `Around advice` defines the invocation of the superimposed method (using a `proceed` statement) in the AOP implementation. This statement has to be transformed into the `Super` statement of FOP, i.e., into the invocation of the superimposed method.

For example, since the call pointcut `LabelChangeEvent` (Fig. 3 Line 2) is not restricted and does not refer to the object calling the method `setText`, we propose to transform the according advice (Line 3) into a FOP method extension of the called method `setText` (Lines 6–9).

If one call statement is the only statement of a method (e.g., `setText` is the only statement of the method `click`;

\(^1\)http://www.cs.utexas.edu/users/schwartz/ATS.html

\(^2\)http://www.cs.utexas.edu/users/schwartz/ATS.html

\(^3\)http://www.cs.utexas.edu/users/schwartz/ATS.html
public aspect MyAspect {
    protected pointcut
        LabelChangeCall() : call(LabelChangeCall{ /*...*/})
    before(): LabelChangeCall{ /*...*/}
}
public void myLabel.setText("Button clicked");
5. CASE STUDY

Since design patterns solve recurring and diverse problems of software development and incorporate crosscutting concerns [14], they qualify best as a case study for exploring the duality of AOP and FOP as well as for our transformation rules.

To verify our transformation rules, we manually applied them to refactor the AspectJ design pattern implementations by Hannemann et al. into AHEAD counterparts. We executed the transformed FOP code and compared it to the AOP implementation. Despite few exceptions (that will be discussed later) all FOP implementations behave equivalently to their AOP originals.

In summary, 37 of 41 aspects (19 of 23 design patterns) could be transformed straightforwardly using our transformation rules. Some aspects could not be transformed as we will explain soon. Furthermore, the most of the aspect-specific mechanisms of pointcuts (13 of 19) can be transformed into method extensions. Two of the remaining six pointcuts refer to the calling object point to methods that are only forwarding requests, as described above.

Table 1 depicts how often we applied a specific transformation to obtain a FOP counterpart. The last row shows, that only 9 out of 178 AOP elements were not transformed into FOP.

Figure 8 shows the different AOP mechanisms used in the case study of Hannemann et al. **PCA, ITD, AFM, and Parent Declarations** were applied most frequently—in sum 84% (151 of 178 AOP elements). Since we showed that these mechanisms are equivalent to FOP mechanisms (Mixins and Singleton) we argue that AOP and FOP provide similar means in many situations for solving recurring problems of software development.

We observed, that only about 50% of the mechanisms and keywords available in AspectJ (19 of 40) were actually used in the case study of Hannemann et al. [16].

Now we want to explain why we were not able to transform all aspects:

**PCA.**

We could not transform two pointcuts with pieces of advice, because the pointcuts were declared only and advice has not been applied. One aspect in the AOP implementations advises constructor calls that occur within a complex method. (The extension of the constructor of the respective class does not allow to exchange the returned object and thus FOP does not allow to prevent the creation and use of a new object.)

**AFM.**

Two aspect fields and methods referred to advice that we were not able to transform (see **PCA** above). Since the advice cannot be implemented, we could omit these associated members of the aspect.

**ITD.**

AspectJ allows to extend interfaces using method definitions. If a class implements different interfaces, that are extended by aspects using method definitions, multiple inheritance is introduced. Multiple inheritance is not supported in Java and AHEAD. Therefore, we were not able to add three methods to the aimed interfaces.

**Other AOP.**

One **declare warning** statement could not be transformed since FOP does not affect the compilation process and cannot define compiler warnings.

5.1 Limitations of the Transformation Rules

As depicted by several studies both, AOP and FOP, can be used to encapsulate crosscutting concerns. Nevertheless, FOP lacks implementing some AOP mechanisms, e.g., the lack of FOP for extending method calls within complex methods (see above). Figure 9 depicts advice (adopted from tasks) that is applied at join point shadows defined by a non-trivial pointcut. This pointcut refers to the dynamic
Figure 9: Complex pointcut that can not be transformed into FOP directly.

control flow of the program (cflow) which causes the failure of the transformation into FOP, because the points of extension are not evaluable statically.

Furthermore, FOP lacks appropriate implementation of homogeneous crosscutting concerns, i.e., the extension of different join point shadows with identical code. In this case FOP introduces code replication which decreases maintainability. It has been observed that homogeneous pointcuts occur very rare in medium sized software projects – in our case study only 2 of 23 design pattern implementations include homogeneous crosscutting concerns.

In this section we showed that FOP can not implement all sophisticated AOP mechanisms. But this does not mean the failure of FOP. If problems occur where AOP fits best and FOP fits best, we need to transform AOP implementations into FOP. This has not been studied so far. Furthermore, we will apply different software metrics and performance measurements to compare AOP and FOP in this direction in detail.

8. REFERENCES

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


