Published in the Proceedings of the ECOOP Workshop on Aspects, Dependencies, and Interactions (ADI'06), Nantes, France, July 3rd, 2006.

On the Notion of Functional Aspects in Aspect-Oriented Refactoring

Sven Apel[1] and Jia Liu[2]

[1] Department of Computer Science, University of Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de
[2] Department of Computer Sciences, University of Texas at Austin
jliu@cs.utexas.edu

Abstract. In this paper, we examine the notion of functional aspects in context of aspect-oriented refactoring. Treating aspects as functions reduces the potential interactions between aspects significantly. We propose a simple mathematical model that incorporates fundamental properties of aspects and their interactions. Using this model, we show that with regard to refactoring functional aspects are as flexible as traditional aspects, but with reduced program complexity.

1 Introduction

Aspect-oriented refactoring (AOR) is the process of decomposing a legacy program into a well-structured core and a set of aspects that implement concerns that crosscut the core functionality. This process, also referred to *horizontal decomposition* [9], untangles the structure of a legacy program by gradually detaching code pieces and encapsulating them in aspects.

The essence of AOR can be formulated using simple mathematics. The following equation expresses the relation between a legacy program P and a program P' that was refactored in order to detach aspect A:

$$P = A * P' \tag{1}$$

The operator '*' denotes aspect weaving. Aspects are detached in a step-wise manner. In its general form, AOR means decomposing program P incrementally into an untangled core program P' and n aspects A_i:

$$P = A_1 * A_2 * \ldots * A_{n-1} * A_n * P' \tag{2}$$

AOR can be understood as the inverse process of aspect weaving. While the former process detaches aspects of a program, the latter attaches aspects. While introductions of an aspect (as defined in AspectJ) are targeted to the classes of program P', advice can have global effects. That is, an advice in aspect A_i can advise an aspect A_j, for any pair (i, j) (Fig. 1). In theory, the number of potential aspect interactions (A_i advises or refers to A_j) grows by $O(n^2)$, although in practice, a desire is for aspect interactions be either well-understood or non-existent. Note, the *refactoring order* of aspects is not necessarily equivalent to *aspect weaving* order, especially if aspects are independent of each other.

AOR and step-wise development. A different perspective to understand aspects and AOR can be found in the practice of *stepwise software development (SWD)* [7, 8, 3, 2]. The idea behind SWD is to develop a program from a minimal base and successively applying refinements that implement different design decisions, called *development steps*.

Usually, refinements in SWD are modeled as functions; they take a program as input and produce a modified program as output. This interpretation is not far from the intuitive understanding of aspects. For example, aspect A takes program P' as input and produces the modified program P.

$$P = A(P') \tag{3}$$

A sequence of n factored aspects is modeled by function composition, where function application is (a form of) aspect weaving:

$$P = A_1(A_2(\ldots A_{n-1}(A_n(P'))\ldots)) \tag{4}$$

This sequence could be continued by substituting P' for a detached aspect A_{n+1} that takes a refactored program P'' as input.

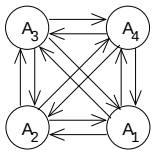

Fig. 1. Potential interactions of *traditional aspects*.

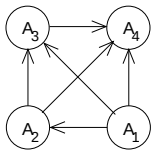

Fig. 2. Potential interactions of *functional aspects*.

At the first glance, the notion of *functional aspects* seems to be only a mathematical or even notational concern. But there are some deeper issues, as we will explain: Notably, SWD and function composition impose a fixed order on weaving. This is because of refinements affect and extend only that program that was produced by previous development steps [7]. This restriction limits the number of potential interactions between different refinements and therewith improves program comprehensibility. As Figure 2 shows, modeling aspects as functions leads to a different interaction pattern, one that is theoretically half as complex compared to the traditional one, shown on Figure 1. As intended for traditional aspects, it is desired for the interactions of functional aspects to be either well-understood or non-existent.

With functional aspects, the weaving order is unidirectional; aspects affect only aspects that were woven previously. This enables us to define the refactoring order as the inverse to the weaving order. This definition is valid since all aspects A_i may only affect subsequently detached aspects A_j with $j > i$.

Aspect quantification. The differing interaction patterns result from different interpretations of aspect quantification. While traditional aspects are quantified *globally*, functional aspects are quantified *locally* (they affect only previously woven aspects). This results in a fundamental difference between SWD and AOP.

Nevertheless, functional aspects have the strength that the number of possible interactions is half of the number of the traditional approach. Their underlying model has the potential for formalizing the effects of aspects that is in line prior research on software design and program generation [8, 7, 3, 2, 1, 5]. In this paper, we examine functional aspects in the light of AOR. As mentioned, the unidirectionality of functions imposes a fixed weaving order, but also a fixed refactoring order. That is, we cannot factor out an aspect A and subsequently factor out aspect B with B affecting A. This is not allowed with functional aspects. Given these facts, it would seem that refactoring functional aspects would be more difficult than refactoring traditional aspects, because the order in which aspects are composed must known a priori. That is, it would seem that A_1 must be refactored first in (4), then A_2, ... and finally A_n. Knowing this order up-front seems unlikely.

Perspective, goals, and contributions. We believe that the concept of functional aspects matches very closely with prior work in software design and automated software development. Their main benefit is that the number of potential aspect interactions is half of the one of traditional aspects. A fundamental question regarding functional aspects is: does the order in which functional aspects are refactored matter? And can mathematical models of the effects that functional aspects make to a program provide help in answering this question?

In this paper, we explore answers to these questions. Our analysis allows us to demonstrate a flexibility in refactoring functional aspects that is normally attributed to traditional aspects, i.e., that the order in which traditional aspects are refactored does not matter. (Note: that we believe that the refactoring-aspects-order-does-not-matter is an assumption of the AOP community; our results may suggest why this assumption has a basis in fact). We demonstrate how a mathematical model may help to promote refactoring approaches, in particular, we show how to transform functional (and also traditional) aspects in a way that they become commutative with respect to the refactoring order.

2 Aspect Interactions and Dependencies

In our analysis we focus on the following kinds of interactions:

References. Aspects that refer to other aspects depend on these aspects. Such referential dependencies are for example method calls or advising other aspects. Treating aspects as functions implies that referred aspects have to be woven before the referring aspects; as mentioned, with traditional aspects the order does not matter.

Overlapping join points. Aspects interfere with other aspects when advising the same join point. In case of overlapping join point sets, the order of aspects matters, because different orders result in different program semantics. The right order has to be inferred from the domain, i.e. which aspect advises a join point first. With respect to overlapping join point sets, both kinds of aspects behave equally.

3 Algebraic Properties

Dependencies between functional aspects impose a fixed refactoring order. In this section, we show how to alter this order without affecting the program semantics. On one hand, this demonstrates similar flexibility with traditional aspect semantics. On the other hand, it gives evidence that the refactoring order does *not* matter.

Altering the order of refactoring is achieved by respectively *swapping* two aspects, or more precisely swapping the composition order. Functional aspects can be swapped if they are commutative. Otherwise, we exploit a notion which we call pseudo-commutativity to create swappable aspects.

3.1 Commutativity

Two aspects A and B are commutative if they can be swapped, and this does not affect the program semantics. Commutativity is the ideal case and yields the most flexibility in permuting the refactoring order.

$$A(B(P)) = B(A(P)) \tag{5}$$

The left-hand side of the equality means A is factored first, then B; the right-hand side means B is factored first, and then A.

A precondition for commutativity of aspects is that these aspects do not depend on each other – this means there are no referential dependencies and no join point overlapping. Figure 3 lists two aspects *Foo* and *Bar* that are commutative with respect to a program. They do not refer to each other and they advise a disjoint set of join points (calling m and n).[3]

```
1 aspect Foo {                          4 aspect Bar {
2   before() : call(* n()) { ... }      5   before() : call(* m()) { ... }
3 }                                     6 }
```

Fig. 3. Two commutative aspects.

3.2 Pseudo-Commutativity

Two aspects A and B are pseudo-commutative (1) if they are not commutative, but (2) they can be transformed to A' and B' so that swapping them does not affect the program semantics.

$$A(B(P)) = B'(A'(P)) \tag{6}$$

The above equation means that for each pseudo-commutative pair A and B there is at least one pair A' and B' that, when swapped, does not change the program

[3] For simplicity, we depict both aspects in the same listing. The order in the listing is arbitrary and does not refer to any order.

semantics. Hence, a programmer may implement aspects with this property in several ways, e.g. A or instead A'. If functional aspects are pseudo-commutative, the order of refactoring does not matter. The programmer can always find for each aspect an appropriate implementation that fits the already factored sequence of aspects. In our example, it does not matter if we factor A first and then B, or B' first and then A'. Although the implementations of A and B differ from A' and B', they implement the same concerns, but in a different way.[4]

In the following, we illustrate several examples of pseudo-commutativity.

Resolving referential dependencies. Let *Foo* and *Bar* be aspects, and *Foo* refers to *Bar* by calling the static method *bar* (Fig. 4). Since we treat aspects as functions, *Bar* has to be woven first because of the referential dependency; the only correct order is $Foo(Bar(Prog))$.

```
1 aspect Foo {                                  5 aspect Bar {
2   void foo() { Bar.bar(); }                   6   static void bar() { ... }
3   before() : call(* n()) { foo(); }           7   before() : call(* m()) { ... }
4 }                                             8 }
```

Fig. 4. Two referential dependent aspects.

However, applying the notion of pseudo-commutativity, we are able to derive *Foo'* and *Bar'* so that the expression $Bar'(Foo'(Prog))$ equals $Foo(Bar(Prog))$. By doing this, we achieve the same flexibility as traditional aspects. Figure 5 depicts one possible pair of *Foo'* and *Bar'*.

```
1 aspect Foo´ {                                 5 aspect Bar´ {
2   void foo() { /* ref. removed */ }           6   static void bar() { ... }
3   before() : call(* n()) { foo(); }           7   before() : call(* m()) { ... }
4 }                                             8   before() : execution(void foo()) {
                                                9     bar(); }
                                               10 }
```

Fig. 5. Resolving referential dependencies.

Foo' no longer refers to *Bar'* (Line 2), i.e. *Foo'* does not directly call method *bar*. In order not to affect the program behavior, *Bar'* itself triggers the call of *bar* by introducing an advice (Lines 8-9). The advice is executed before that join point in *Foo'* that is responsible for calling *bar*. Note that the basic meaning or effect of *Foo* and *Bar* (advising *n* and *m* as well as *bar* is invoked when is *foo* is executed) is preserved by *Foo'* and *Bar'*.

In the above example, we removed a method call to another aspect in order to swap both. Now suppose that one aspect refers to another by means of an advice. Interestingly, this is exactly the opposite case of removing a method call

[4] A related idea can be found in design maintenance systems: Program transformations implement design decision in stepwise manner. Baxter has shown that the order of design decisions can be altered by altering the transformations [4].

reference. Take *Bar'* and *Foo'* as example (Fig. 5); *Bar'* advises *Foo'*. To remove this reference (in order to swap them), we have to find their pseudo-commutative counterparts – these are exactly the original aspects *Foo* and *Bar* (Fig. 4).

Cyclic referential dependencies. Cyclic referential dependencies are a special case of referential dependencies. Figure 6 shows that aspect *Foo* refers to *Bar* by advising *bar* (Line 3); the advice invokes method *adviceBar*. Symetrically, *Bar* refers to *Foo* by advising *foo* (Line 7). If we cannot remove such cyclic referential dependencies, we have an example where traditional aspects are more flexible. This is because with traditional aspect semantics, the following two composition orders would be correct: $Foo * Bar * Prog$ and $Bar * Foo * Prog$.

```
1 aspect Foo {
2   void foo() { ... }
3   before() : execution(void Bar.bar()) { adviceBar(); }
4 }
5 aspect Bar {
6   void bar() { ... }
7   before() : execution(void Foo.foo()) { adviceFoo(); }
8 }
```

Fig. 6. Cyclic referential dependencies.

To remove cyclic referential dependencies the idea of *sandwiching* helps [7]. Although sandwiching is a mechanism to resolve cyclic dependencies in *uses-hierarchies*, it can be adopted to our problem. Using sandwiching, we can split each pair of cyclic referential dependent aspects A and B into A_1, A_2 and B_1 and B_2 so that:

$$A_2(B_2(A_1(B_1(P)))) = B_2(A_2(B_1(A_1(P)))) \tag{7}$$

A_2 refers to B_1 and B_2 to A_1. This is called sandwiching because of A_2 and A_1 embrace B_2, and B_2 and B_1 embrace A_1. After this transformation the ordering of (B_1, A_1) and (B_2, A_2) can be swapped pairwise. Since we eliminated cycles, the remaining referential dependencies can be removed for further swapping, i.e. (B_1,B_2) and (A_1,A_2).

Figure 7 illustrates the sandwiching of the aspects *Foo* and *Bar* (cf. Fig. 6):

$$Foo2(Bar2(Foo1(Bar1(Prog)))) = Bar2(Foo2(Bar1(Foo1(Prog)))) \tag{8}$$

Foo1 defines method *foo* (Line 10); *Bar1* defines method *bar* (Line 13); *Foo2* and *Bar2* encapsulate the dependent code. But now *Foo2* refers to *Bar1* (Lines 2-3), and *Bar2* to *Foo1* (Lines 6-7). Hence, *Foo2* and *Bar2* can be swapped since they only depend on *Foo1* and *Bar1*. Since *Foo1* and *Bar1* do not depend on one another, they can be swapped as well.

However, sandwiching is only the first step to derive the two possible orderings mentioned above: $Foo(Bar(Prog))$ and $Bar(Foo(Prog))$. But now that there are only unidirectional referential dependencies left, we can further transform the aspects by exploiting pseudo-commutativity in order to permute their

```
aspect Foo2 {
  before() : execution(void Bar1.bar()) {
    adviceBar(); }
}
aspect Bar2 {
  before() : execution(void Foo1.foo()) {
    adviceFoo(); }
}
aspect Foo1 {
  void foo() { ... }
}
aspect Bar1 {
  void bar() { ... }
}
```

Fig. 7. Sandwiching for resolving cyclic dependencies.

ordering; we can derive the orderings $Bar2'(Bar1'(Foo2'(Foo1'(Prog))))$ (Fig. 8) and $Foo2''(Foo1''(Bar2''(Bar1''(Prog))))$ (Fig. 9), whereby respectively the parts of *Bar* can be merged again, and symmetrically the parts of *Foo*. After this step, we are able to derive both orders $Bar(Foo(Prog))$ and $Foo(Bar(Prog))$, as with traditional aspects.

```
aspect Bar2´ {
  before() : execution(void Foo1.foo()) {
    adviceFoo(); }
}
aspect Bar1´ {
  void bar() { adviceBar(); ... }
}
aspect Foo2´ {

}
aspect Foo1´ {
  void foo() { ... }
}
```

Fig. 8. Ordering: $Bar2'(Bar1'(Foo2'(Foo1'(Prog))))$.

```
aspect Foo2´´ {
  before() : execution(void Bar1.bar()) {
    adviceBar(); }
}
aspect Foo1´´ {
  void foo() { adviceFoo(); ... }
}
aspect Bar2´´ {

}
aspect Bar1´´ {
  void bar() { ... }
}
```

Fig. 9. Ordering: $Foo2''(Foo1''(Bar2''(Bar1''(Prog))))$.

Resolving join point overlapping. Two aspects interact with each other if they advise overlapping sets of join points. Naturally, the order matters in this situation, for functional *and* traditional aspects. Weaving in different orders results in different execution orders of the connected advice. Suppose *Foo* and *Bar* advice the same method call (Fig. 10); both aspects were already factored out. Changing the refactoring and thereby the following weaving order would result in semantically differing programs because either *foo* is called before *bar* or vice versa. However, with AspectJ it is possible to define a weaving order explicitly, but this can be complicated and not all desired orders can be expressed [5].

We use pseudo-commutativity to resolve join point overlapping. This methodology is useful for both kinds of aspects. Suppose method n is called in a *main*

```
1 aspect Foo {
2   void foo() { ... }
3   before() : call(* n()) { foo(); }
4 }
```

```
5 aspect Bar {
6   void bar() { ... }
7   before() : call(* n()) { bar(); }
8 }
```

Fig. 10. Join point overlapping.

method (Fig. 11). Right before calling method n our two concerns (later on implemented by *Foo* and *Bar*) are to be executed. Factoring both concerns using the above depicted aspects would be possible only in one order, first *Bar* then *Foo*. In order to be able to swap *Foo* and *Bar*, we can prepare the code associated with the target join points themselves. We simply add hooks for each advice that is supposed to bind to the join points. The methodology of preparing code is in line with the observation of Murphy et al. [6].

```
1 void main(String[] argv) {
2   /* Bar */; /* Foo */; n();
3 }
```

Fig. 11. Two concerns intermixed with the base program.

```
1 void main(String[] argv) {
2   hookBar(); hookFoo(); n();
3 }
```

Fig. 12. A target join point for *Foo* and *Bar*.

Figure 12 shows the refactored *main* function. The both hooks *hookFoo* and *hookBar* implement the corresponding concern functionality. Then, when refactoring out *Foo* and *Bar* as aspects the (empty) hook methods mark their join points. The hooks fix the concern execution order. They enable us to alter the refactoring order (first *Foo*, then *Bar*) because now we bind the aspects not to calling n, but to calling the hooks (Fig. 13). Thus, the order of refactoring no longer influences the order of execution, with functional aspects *and* traditional aspects.

```
1 aspect Foo {
2   void foo() { ... }
3   before() : call(* hookFoo()) {
4     foo(); }
5 }
```

```
6 aspect Bar {
7   void bar() { ... }
8   before() : call(* hookBar()) {
9     bar(); }
10 }
```

Fig. 13. Binding advice to hooks.

4 Conclusions

In this paper we examined the flexibility of functional aspects in context of AOR. We presented a simple algebraic approach that models aspects as functions. This is in line with prior work on software design and composition. We explored if functional aspects are as flexible as traditional aspects with respect to altering

the refactoring order. We raised this question because, on one hand, treating aspects as functions reduces program complexity by decreasing the number of potential aspect interactions, and, on the other hand, it seemed that the traditional model was more flexible w.r.t. the ordering of aspects. In order to address these issues, we analyzed the properties and dependencies of functional aspects. We used our mathematical model to abstract over implementation details and to be able to make general statements.

Specifically, we have shown that certain kinds of aspect dependencies caused by references and overlapping join points can be resolved by applying the notion of pseudo-commutativity. Both kinds of aspects are similarly flexible with respect to the order of refactoring. We claim that each pair of aspects with referential dependencies or overlapping join points can be transformed into a corresponding pseudo-commutative pair. While our ideas are preliminary, we believe our theory offers practice potential in future AOR technologies.

With this study we could show that the SWD perspective does not constrain the known techniques of AOR. It has the same power than the traditional approach but reduces potential interactions. It is more disciplined and advantageous with regard to complexity and comprehensibility [5].

Our next task is to decompose larger applications to test the validity of our ideas. This is a subject of on-going work.

Acknowledgments. We thank Don Batory for fruitful discussions and useful comments on drafts of this paper. The first author is sponsored by the German Research Foundation (DFG), project number SA 465/31-1, as well as by the German Academic Exchange Service (DAAD), PKZ D/05/44809. This work was done while Sven Apel was visiting the group of Don Batory at the University of Texas at Austin.

References

1. S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *ICSE*, 2006.
2. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1(4), 1992.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
4. I. Baxter. Design Maintenance Systems. *CACM*, 35(4), 1992.
5. R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *ACM SIGPLAN PEPM Workshop*, 2006.
6. G. C. Murphy et al. Separating Features in Source Code: An Exploratory Study. In *ICSE*, 2001.
7. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE TSE*, SE-5(2), 1979.
8. N. Wirth. Program Development by Stepwise Refinement. *CACM*, 14(4), 1971.
9. C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *OOPSLA*, 2004.