Program Refactoring using Functional Aspects

Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
ckaestne@ovgu.de

Don Batory
Dept. of Computer Sciences
University of Texas at Austin, USA
batory@cs.utexas.edu

Abstract

A *functional aspect* is an aspect that has the semantics of a transformation; it is a function that maps a program to an advised program. Functional aspects are composed by function composition. In this paper, we explore functional aspects in the context of aspect-oriented refactoring. We show that refactoring legacy applications using functional aspects is just as flexible and expressive as traditional aspects (functional aspects can be refactored in any order), while having a simpler semantics (aspect composition is just function composition), and causes fewer undesirable interactions between aspects (the number of potential interactions between functional aspects is half the number of potential interactions between traditional aspects). We analyze several aspect-oriented programs of different sizes to support our claims.

Categories and Subject Descriptors D.2.7 [**Software**]: Software Engineering—*Distribution, Maintenance, and Enhancement*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms Design, Languages

Keywords Functional Aspects, Aspect-Oriented Refactoring, Stepwise Refinement, Aspect Interactions, Pseudo-Commutativity

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 [10, 32, 26]. This process improves the structure of a legacy program by gradually untangling code pieces and encapsulating them into aspects while preserving the program's behavior.

The essence of AOR can be expressed mathematically. The following equation expresses behavioral equality between a legacy program P and a program P_n woven with n aspects $(A_1, \ldots, A_n)$, which has been refactored in order to detach aspects incrementally.

$$P = A_1 * A_2 * \cdots * A_{n-1} * A_n * P_n \quad (1)$$

The effect of the operator '$*$' denotes aspect weaving. AOR is the inverse process of aspect weaving. While AOR detaches aspects from a program, aspect weaving combines them back together.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

Copyright © 2008 ACM 978-1-60558-267-2/08/10. . . $5.00

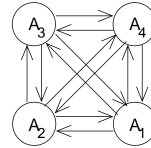


Figure 1. Potential interactions of *traditional aspects*.

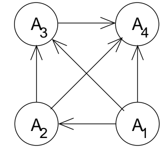


Figure 2. Potential interactions of *functional aspects*.

Aspects in popular aspect-oriented languages like AspectJ[1] or AspectC++[2] are applied *at once*. We call those aspects *traditional*. Unless explicitly declared with `declare precedence` the weaving order is not specified. Furthermore, advice has global effects. That is, a piece of advice in aspect A_i can advise members and introductions of any other aspect (Fig. 1). Inter-type declarations ('introductions') can introduce members into any class of program P_n and into any other aspect A_i. In theory, the number of potential aspect interactions (A_i advises or refers to A_j) grows by $\mathcal{O}(n^2)$, although in practice, a desire is for aspect interactions to be well-understood.

AOR and Stepwise Development

A different perspective on aspects and AOR can be found in the practice of *stepwise refinement* [36, 41, 8, 7, 37]. The idea behind stepwise refinement is to develop a program from a minimal base and successively apply *refinements* that implement different design decisions in distinct development steps.

Usually, refinements are modeled as functions; they receive a program as input and produce a modified program as output. This interpretation is not far from an intuitive understanding of aspects. For example, $P = A_1(P_1)$ denotes an aspect A_1 that takes a program P_1 as input and produces the modified program P. Weaving a sequence of n factored aspects is modeled by consecutive function application:

$$P = A_1(A_2(\ldots A_{n-1}(A_n(P_n))\ldots)) \quad (2)$$

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

In contrast to traditional aspects, we refer to aspects that are transformations and are composed like functions as *functional aspects*. Unlike traditional aspects, functional aspects impose a fixed order on aspect weaving and restrict the scope of aspects to artifacts introduced earlier. This is because a transformation affects and extends only that program that was produced by previous development steps [36]. In the composition $A_1(A_2(P_2))$, transformation A_1 modifies what is produced by applying transformation A_2 to

[1] `http://www.eclipse.org/aspectj/`

[2] `http://www.aspectc.org/`

program P_2. Transformation A_2 is unaware of A_1 and, thus, cannot affect A_1.

This restriction limits the number of potential interactions between different refinements. As shown in Figure 2, modeling aspects as functions leads to a different interaction pattern, one that is theoretically half as complex compared to the traditional one, shown in Figure 1. Therefore, it has been argued that functional aspects may potentially improve program comprehensibility [29]. Modeling aspects as functions is consistent with prior and current research on software design, program generation, and program refactoring [41, 36, 7, 15, 29, 27].

We have extended *ARJ* compiler [2] in order to support and experiment with functional aspects. It can be downloaded at the project's web site.[3]

Perspective, Goals, and Contributions

We start with the premises that functional aspects ease understanding and are more disciplined regarding SWR as discusses in earlier work [29]. We do not discuss advantages or disadvantages of functional aspects compared to traditional aspects (e.g., with regard to readability or maintainability), but note that functional aspects correspond to a form of stepwise refinement, and hence to a time-honored approach to software development. Moreover, we focus on syntactical relationships between functional and traditional aspects and not on whether aspects encapsulate a concern properly or behave according to a specification.

Our work provides a theoretical basis that establishes that functional aspects are as expressive as traditional aspects. The readability and maintainability discussion does not invalidate our results since functional aspects embody a fundamental form of thinking about software development: program extensions (e.g., implemented via aspects) are functions that map programs to programs [41, 8, 36, 7].

However, function evaluation imposes a fixed weaving order, but also a fixed refactoring order. That is, we cannot factor out aspect A_1 in a first step, i.e., $P = A_1(P_1)$, and then factor out aspect A_2, i.e., $P = A_1(A_2(P_2))$, where A_2 affects A_1. So functional aspects may be simpler as they reduce possible interactions, but may be harder to use as the order in which functional aspects are composed must be known a priori. That is, it would seem that A_1 must be refactored first in Equation 2, then A_2, A_3, and so on and as last aspect A_n. Knowing this order up-front is unlikely.

Consequently, we raise a fundamental question regarding functional aspects: does the order in which functional aspects are refactored (i.e., detached from the base program) matter? In this paper, we show three things:

- Refactoring legacy applications using functional aspects is just as flexible and expressive as traditional aspects in that functional aspects can be refactored in any order.[4]
- The semantics of functional aspects is simpler than of traditional aspects since composition of functional aspects is just function composition.
- Using functional aspects causes fewer undesirable aspects interactions than using traditional aspects because the number of potential interactions between functional aspects is half the number of potential interactions between traditional aspects.

[3] `http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/`

[4] Note, we believe that the order in which traditional aspects are refactored does not matter is an assumption of the AOP community; our results suggest why this assumption has a basis in fact.

We evaluate our proposal by refactoring functional aspects in a small-sized legacy application and by analyzing several third-party aspect-oriented programs of different sizes.

This paper is an extension of a workshop paper [5] and makes the following contributions (where only the first was presented in [5]):

- We classify aspect interactions and introduce the notion of pseudo-commutativity.
- We present a formal model and algorithm describing the operations required for pseudo-commutativity and the constraints a pseudo-commutative transformation must satisfy.
- Using this formal model, we show that pseudo-commutativity works for all common Java and AspectJ language constructs.
- We explore functional aspects in more realistic software projects and confirm the claims arising from our model.

2. Aspect Interactions

Dependencies between aspects can be caused by two different kinds of aspect interactions: *references* and *shared join points* [13].

1. Aspects *referring* other aspects depend on these aspects. References include calling or advising members or introductions of these aspects. Treating aspects as functions requires that the referenced aspects are woven previously.
2. A *shared join point* is a join point advised by more than one aspect.[5] An aspect interferes with another aspect when the set of join points they advise contains at least one shared join point. In case of such *overlapping join point sets*, the order of weaving aspects matters, because different orders result in different program behavior. The weaving order of functional aspects is enforced by their composition order, hence there is no need for `declare precedence` statements.

3. A Model of Functional Aspects

Dependencies between functional aspects impose a fixed refactoring and weaving order. We show how to alter this order without affecting program behavior. All code examples and formulas in this section refer to functional aspects.

3.1 Commutativity

Two aspects A and B are commutative if they can be swapped without affecting program semantics. Commutativity is the ideal case and yields the most flexibility in permuting refactoring orders:

$$A(B(P)) = B(A(P)) \quad (3)$$

The left-hand side of the (behavioral) 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 interact with each other, i.e., they do not reference each others introductions and they do not share join points.[6]

In Figure 3, we exemplify two independent aspects `Foo` and `Bar` implemented in AspectJ. They do not refer to each other and they advise disjoint sets of join points (calls to the methods `m` and `n`).

[5] Some AOP publications distinguish between 'join points' which occur dynamically during the execution of a program and 'join point shadows', the location of a join point in the static code. We refer to the latter.

[6] Note that there might be aspects that advise the same join point and being swapped do not change the observeable program behavior [38]. However, without knowledge of the programmer this cannot be guaranteed, e.g., think of I/O operations that do not change advised data but whose execution order matters.

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

```
4 aspect Bar {
5   before() : call(* m()) { ... }
6 }
```

Figure 3. Two commutative aspects.

3.2 Pseudo-Commutativity

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

$$A(B(P)) = B'(A'(P)) \tag{4}$$

Equation (4) means that for a pair of functional aspects (A, B) there is a pseudo-commutative pair of functional aspects (A', B'), so that applying either pair results in the same composite transformation.[7] Although the implementations of A and B differ from A' and B', they implement the same concerns but in a different way.[8]

In the following, we illustrate several examples of pseudo-commutativity and how it is used to resolve different kinds of aspect dependencies. Then, we present general guidelines – illustrated by the examples below – for transforming pairs of functional aspects into pseudo-commutative pairs. Finally, we present a formal model of their transformations.

3.2.1 Resolving Referential Dependencies

Let `Foo` and `Bar` be functional aspects, and `Foo` refers to `Bar` by calling the static[9] method `bar` (Fig. 4). Since both are functional aspects, `Bar` has to be woven first because of the referential dependency; the only correct order is *Foo*(*Bar*(*Prog*)).

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

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

Figure 4. Two referentially dependent aspects.

However, applying the notion of pseudo-commutativity, we can create two functional aspects `Foo'` and `Bar'` that implement the same concerns but in a different way, so that the expressions *Bar'*(*Foo'*(*Prog*)) and *Foo*(*Bar*(*Prog*)) result in equivalent programs. Thus, using pseudo-commutativity, we achieve the same flexible refactoring order as with traditional aspects. In Figure 5, we depict one possible pair of `Foo'` and `Bar'`. `Foo'` no longer calls method `bar` in `Bar'` directly (Line 2). Instead, `Bar'` itself triggers the invocation of `bar` by advising the execution of `foo` (Line 8). The advice is executed before that join point in `Foo'` that is responsible for calling `bar`. Note, the basic semantics of `Foo` and `Bar` – advising `n` and `m` as well as `bar` is invoked when `foo` is executed – are preserved by `Foo'` and `Bar'`.

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

```
5 aspect Bar' {
6   static void bar() { ... }
7   before() : call(* m()) { ... }
8   before() : execution(void foo()) { bar(); }
9 }
```

Figure 5. Resolving a referential dependency.

In our example, we removed a method call and replaced it by implicit invocation in another aspect in order to swap both. Now suppose that one aspect refers to another via advice. Interestingly, this is the opposite case of removing a method call reference. Take `Bar'` and `Foo'` as example (Fig. 5) where `Bar'` advises `Foo'`. To remove this reference (in order to swap aspects), we have to find their pseudo-commutative counterparts – these are exactly the original aspects `Foo` and `Bar` (Fig. 4).

Cyclic referential dependencies, in which two aspects mutually reference each other, are not possible with functional aspects. Cyclic referential dependencies of traditional aspects, can be resolved simply by reverting only those references pointing in one direction. That is, we achieve pseudo-commutativity even for aspects that would have cyclic dependencies when implemented traditionally.

Note that the property of pseudo-commutativity exploits the AOP mechanism of *dependency inversion*, i.e., the ability to reverse the dependency between modules [33]. It provides a flexibility needed for AOR using functional aspects.

3.2.2 Resolving Shared Join Point Dependencies

Two aspects interact with each other when they advise overlapping sets of join points. Naturally, the order matters in this situation for both functional *and* traditional aspects. Weaving in different orders results in different execution orders of involved advice and thus in different program behavior. Suppose the two traditional aspects `Foo` and `Bar` advise the same method call as in Figure 6, changing the weaving order would either execute `foo` before `bar` or vice versa.

```
1 void main() { n(); ... }
```

```
2 aspect Foo {
3   void foo() { ... }
4   before() : execution(* main()) {foo();}
5 }
```

```
6 aspect Bar {
7   void bar() { ... }
8   before() : execution(* main()) {bar();}
9 }
```

Figure 6. A shared join point dependency.

We use pseudo-commutativity to resolve shared join point dependencies, i.e., to transform the pair functional aspects (`Foo`, `Bar`) into their pseudo-commutative counter-parts (`Foo'`, `Bar'`). Note, for this transformation, we also have to adapt the program P that contains the shared join point.

As shown in Figure 7, a method `n` is called within `main`. Right before calling method `n` our two concerns are to be executed, first `Foo` and then `Bar`. Factoring both concerns using two aspects `Foo` and `Bar` would be possible only in one order, first `Bar` then `Foo`.

[7] Note, the transformation from traditional aspects to functional aspects described by Lopez-Herrejon et al. [29] is closely related to pseudo-commutativity.

[8] A related idea can be found in design maintenance systems: program transformations implement design decisions in a stepwise manner. Baxter has shown that the order of design decisions can be altered by altering the transformations [8].

[9] We use static methods only for the purpose of a concise example, without loss of generality.

In order to be able to swap `Foo` and `Bar` (i.e., its pseudo-commutative siblings), we prepare the code associated with the target join points themselves, i.e., as illustrated above, we change the base program P (which contains only `main`) to a modified base program P'. A simple way to do so is to add a *hook* for each piece of advice that extends a join point; more sophisticated ways are described later. A straightforward implementation is to add a method call to a new method and move all statements from the original method to the new one. Now, there are two methods that can be advised, the original method `main` and the new `hook` method. The advice that is to be executed first advises the `main` method, the other one the `hook` method, as shown in Figures 7 and 8. If more than two aspects advise the same join points, multiple nested hook methods need to be created. Having two join points by introducing hooks fixes the concern execution order. This allows us to alter the refactoring and the weaving order (i.e., to swap aspects) because now we do not bind both aspects to the execution of `main`, but to calling individual join points. Thus, the weaving order no longer influences the execution order.

```
1 void main() { n(); ... }
```

```
1 void main() { hook(); }
2 void hook() { n(); ... }
```

Figure 7. A shared join point without (top) and with hooks (bottom).

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

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

Figure 8. Binding advice to hooks.

Clearly, there are other ways to achieve the same effect, such as using annotations to mark the relevant positions in the code [24] or to use more expressive pointcut mechanisms [17, 35] and more fine-grained join point models [18, 30]. It is also worth mentioning that code preparation (i.e., including information into the code for subsequent extension/advice) decreases obliviousness [14], but at the same time pinpoints the execution order and makes it explicit. Nevertheless, for the sake of simplicity, we use simple hook methods (i.e., methods that are empty and are inoked only for the purpose of exposing a join/extension point). Hooks are sufficient to illustrate the idea. They can be replaced by annotations or by using a more sophisticated pointcut language obtaining a more elegant solution.

3.3 Formalization and Algorithm

So far, we have illustrated the idea of pseudo-commutativity by means of examples. Now, we sketch a simple formalization, for two reasons. First, it specifies the operations required for pseudo-commutativity and, second, it defines the constraints a pseudo-commutative transformation must satisfy. Using this formalization, we can show that pseudo-commutativity works for all common Java and AspectJ constructs. We refrain from using calculi like Featherweight Java because they are too complex for our purposes and introduce our own model instead.[10]

3.3.1 Basic Definitions

A program p consists of a number of classes from the set of classes $\mathbb{C}$ and aspects from the set of aspects $\mathbb{A}$:

$$p \subseteq \mathbb{C} \cup \mathbb{A} \tag{5}$$

Classes and aspects both have a set of inner program elements. For classes these are methods, fields, inner classes, initializers, etc. Aspects can have additional elements like inter-type declarations or advice. For our purpose, the concrete nature of these inner elements does not matter. We are only interested in two properties that are responsible for dependencies: *references* and *extensions*. References can be caused, for example, by invoking methods, accessing fields, or instantiating objects, inter-type declarations or advice. Extensions, on the other hand, occur when a piece of advice extends a join point.

Classes and aspects are both constructed from an infinite set $\mathbb{L}$ that contains all possible members a given programming language can express, e.g., all possible methods that can be expressed in Java (Equation 6). Furthermore, we introduce a function r that returns the set of aspects that *reference* a program element and a function e that returns the set of aspects that *extend* a program element (Equation 7).[11]

$$\forall A \in p: \ A \subseteq \mathbb{L} \tag{6}$$

$$r: \mathbb{L} \to \mathcal{P}(\mathbb{A}); \quad e: \mathbb{L} \to \mathcal{P}(\mathbb{A}) \tag{7}$$

3.3.2 Commutativity and Pseudo-Commutativity

Two aspects A and B are commutative if three conditions hold. First, program elements in A must not be referenced by aspect B and vice versa. If there was such a reference with functional aspects, it would not be possible to weave the referenced aspect after the referring one. Second, program elements in A must not extend elements in B and vice versa. If there was an extension, this would require the extended aspect to be woven before the extending one. Finally, no program element in the whole program p must be extended by both aspects. If a program element was extended multiple times, the program behavior might depend on the order in which the aspects are woven:

$$\begin{aligned} A(B(P)) = B(A(P)) & \\ \Leftrightarrow \quad & \forall l \in A: B \notin r(l) \wedge B \notin e(l) \\ \wedge \quad & \forall l \in B: A \notin r(l) \wedge A \notin e(l) \\ \wedge \quad & \forall X \in p: \forall l \in X: A \notin e(X) \vee B \notin e(X) \end{aligned} \tag{8}$$

l is a meta-variable for program elements inside an aspect, X is a meta-variable for aspects or classes inside the program.

Pseudo-commutativity is the transformation of two functional aspects A and B – where A is woven after B – into A' and B' so that $A(B(P)) = B'(A'(P))$. This requires the following three conditions:

1. As A is woven after B, the program elements in A may not be referenced or extended by B ($\forall l \in A: B \notin r(l) \wedge B \notin e(l)$). This condition is already satisfied when A and B are indeed functional aspects.
2. As A' is woven before B', it may not reference or extend program elements from B' ($\forall l \in B': A' \notin r(l) \wedge A' \notin e(l)$).

[10] Featherweight Java provides many typing and evaluation rules that are not necessary for our problem. Furthermore, an extension of Featherweight Java toward AspectJ either complicates the model further [20] or does not cover the core of Java [40].

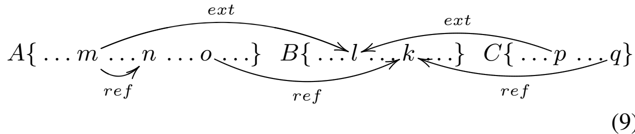
[11] Note that classes have no constructs to extend aspects, references between classes are irrelevant because classes are compiled before weaving. Furthermore, we consider classes referencing aspects bad design since it violates the inversion of control and obliviousness principles. Though, we could model it, it would complicate the formalism without adding anything new.

3. No program elements in the whole program may be extended both by A' and B' ($\forall X \in p : \forall l \in X : A' \notin e(X) \vee B' \notin e(X)$).

3.3.3 Transforming Referential Dependencies

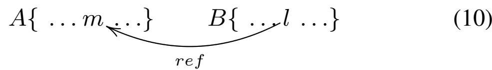
Referential dependencies are caused by violations against Condition 2. That means that a program element in an aspect is referenced or extended by an aspect now woven earlier. In this section, we limit our discussion to methods (that can reference other methods) and advice (that can reference and extend other methods). Later on, we explain whether and how these discussions can be generalized.

For illustration, we extend our notation by further information. Although not needed for our model, it helps to understand pseudo-commutative transformations and may be used eventually for an automation of the transformation steps. Equation 9 illustrates the notation we will use in the remainder. It describes three aspects A, B, and C, each with several program elements like m, n, and o (e.g., methods, fields, inter-type declarations, advice). References are illustrated with 'ref'-arrows below a term. For example, in aspect A, m references n, and o references k in B. Furthermore, extensions are denoted by 'ext'-arrows above a term, for example l in B is extended both by m in A and p in C. We can directly transfer the conditions for pseudo-commutativity to this notation. Conditions 1 and 2 require that all arrows between two aspects point in the same direction (e.g., B must be woven before A and C). Condition 3 is satisfied if above a term there are never two 'ext'-arrows pointing to the same program element (violated in the example because l in B is extended by A and C).



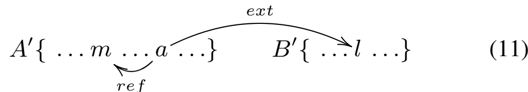
(9)

We start with the two functional aspects A and B in Equation 10, in which the method m in A is referenced by method l in B, which is similar to Figure 4.



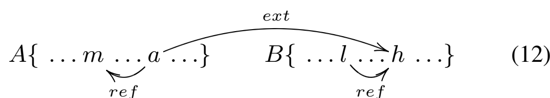
(10)

These two aspects are not commutative because of the reference, but we can transform them into A' and B' to achieve pseudo-commutativity. This pseudo-commutative pair is obtained by creating a piece of advice a in A' and moving the original reference of B to the advice body in A' (as done previously in Figure 5). As we can see in Equation 11, the dependency between both aspects is reverted, instead of a reference from B to A, there is now an extension from A' to B'.



(11)

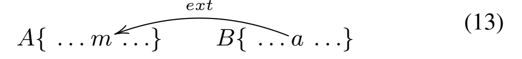
If it is not possible to directly advise method l in B', e.g., because the reference was placed in the middle of the method, it is necessary to expose a new join point by introducing a new hook h, that is called from l and advised by a:[12]



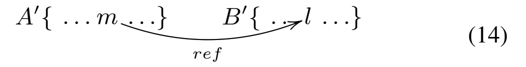
(12)

[12] As explained in Section 3.2, alternative mechanisms such as annotations can be used.

This way it is *always* possible to replace a reference between methods by advice. Similarly, we can always replace a piece of advice with a method call. Consider the aspects A and B in Equation 13, in which advice a extends method m. They can be transformed into A' and B', in which a is replaced by a method l, and m directly calls l (Eq. 14).



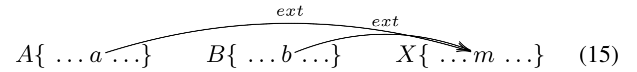
(13)



(14)

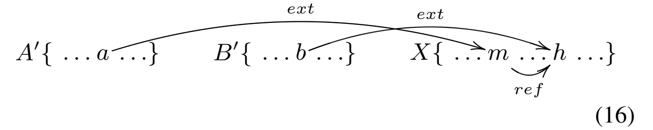
3.3.4 Transformation of Shared Join Points

When two pieces of advice of the aspects A and B extend the same program element X in any class or aspect of the program, in our notation two 'ext'-arrows point to the same element:



(15)

To resolve the dependency, we introduce a new method h in X and move the method body from m to h. We let m call h and let the aspect that was originally woven first (B) advise h instead of m. With the additional method call and the disjoint join points, the two pieces of advice are always executed in the correct order.



(16)

3.3.5 Putting all Transformations Together

When two functional aspects A and B with the original weaving order $A(B(P))$ are transformed in their pseudo-commutative counterparts A' and B' with the weaving order $'B(A'(P))$ the following transformations are applied: First, A and B are copied to A' and B'. Next, every reference from A' to B' is reversed and replaced by an extension from B' to A' as shown above. Second, every extension from A' to B' is replaced by a reference from B' to A' as shown above. Finally, every shared join point is resolved by introducing a hook method and separating both extensions. This means that the individual transformations described above, might need to be applied several times for different program elements in A and B and for different shared join points in the program.

3.4 Generalization to Other Language Constructs

So far, we have demonstrated that we can transform references between methods and method extensions by pieces of advice. We have also demonstrated that we can resolve shared join points on methods. Our transformations are general and only use meta-variables for program elements. They can be combined, and multiple references or extensions between two aspects can be resolved by repeatedly applying these transformations. This leaves us with a number of further possibilities of how references and extensions can occur in Java and AspectJ not discussed so far:

1. Statements: a sequence of statements inside the middle of a method can always be extracted into its own method (cf. Extract Method Refactoring [15]), thus reducing statement extensions to method extensions.
2. Inter-type declarations: inter-type declarations are treated as members of aspects and can be handled completely like methods or fields.

3. A piece of advice can reference a method: the body of a piece of advice can be extracted into a new method, thus reducing references from aspects to references from methods.
4. A method or piece of advice can reference a field: the field access can be encapsulated in an access method, reducing it to references between methods.
5. Object instantiation: Object instantiation can be encapsulated in a factory method, and thus be reduced to method references.
6. Constructors and static initializers: Constructors and static initializers can both be handled as methods, because AspectJ provides pointcut designators to extend them.

Note, in some cases there might be alternative transformations. For brevity, we only show a straightforward solution.

This is an informal argument that pseudo-commutativity transformations are valid for all considered language constructs. While – because of the complexity of Java and AspectJ – this cannot *prove* completeness, i.e., that pseudo-commutativity is possible for any pair of aspects with arbitrary language constructs, we still demonstrate validity for all common language constructs we came across.

3.5 Summary & Discussion

Functional aspects can be refactored in any order, just as traditional aspects. Functional aspects are either commutative or pseudo-commutative. The latter case occurs when the refactoring order enforces a weaving order that contradicts the initial aspect dependencies. Its worth mentioning that although all refactoring orders are possible, not all might be equally 'difficult' to handle. Even in the small illustrative examples so far, one can argue which version is easier to refactor or where the resulting code is easier to read. For example, comparing Figures 4 and 5, developers would usually prefer the first implementation with less dependency inversion and more direct calls. So, there might be a *'natural order'* [25], in which code would have been incrementally added if we developed it from scratch. To get a deeper understanding of this problem and to show the general applicability of pseudo-commutativity, we have conducted several case studies.

4. Case Studies

As a proof of concept, we extended our *ARJ* compiler [2] to support functional aspects in AspectJ.[13] ARJ is implemented on top of the *abc* compiler framework [6]. The ARJ compiler expects a list of aspects, i.e., an ordered list that enumerates all aspects that shall be woven to a program. Receiving a list of aspects the ARJ compiler is able to determine which functional aspects are allowed to advise which program elements. Specifically, ARJ utilizes *pointcut restructuring* to adjust the set of captured join points in order to enforce the function composition semantics. The actual restructuring mechanism is described elsewhere [22].

In order to explore functional aspects in more realistic software projects, we performed two case studies with Java and AspectJ. In first study we refactored GraphBenchmark a library of graph data structures and algorithms, inspired by an earlier product line of graph algorithms [28]. GraphBenchmark is a rather small software project (793 lines of code – LOC).[14] We started with it because we were familiar with graph data structures and algorithms which helped us to predict aspect interactions. We refactored several functional aspects in different orderings while enforcing functional weaving to demonstrate that by exploiting pseudo-commutativity the refactoring order does not matter.

[13] http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/

[14] For comparability, in all case studies, we removed blank lines and comments and counted only lines with more than three characters.

In subsequent case studies, we collected statistics from four further aspect-oriented programs (one program was refactored by ourselves and three were third-party programs) of different sizes (1,670 – 71,027 LOC) in order to estimate the (worst case) effort required to transform aspects into their pseudo-commutative counterparts. The worst case effort quantifies the overhead in using functional aspects without having to forego the compositional flexibility of traditional aspects.

4.1 Refactoring GraphBenchmark

GraphBenchmark implements a basic graph data structure with weighted and directed edges and 9 algorithms, including depth-first search, shortest path, connected components, and cycle checking. We refactored 11 functional aspects that implement fundamental design decisions.[15] These design decisions cut across several classes, so that refactoring them into aspects improves the code quality of the application. The implementation of GraphBenchmark that we refactored contained 12 Java classes and interfaces implementing 793 LOC[16].

Since we were familiar with the domain, it was straightforward to infer which aspects to detach. Also we knew some dependencies which enabled us to experiment with different refactoring orderings. We considered one refactoring order first (Ordering #1) and then applied pseudo-commutativity to derive the reverse order (Ordering #2). For brevity, we illustrate the refactorings of 4 aspects only: *Number* for a vertex numbering algorithm, *Connected* for a connected components algorithm, *Cycles* for a cycle checking algorithm, and *Search* for two variants of a traversal algorithm, but the results for other aspects are similar.

Ordering #1

We chose $Number(Connected(Cycles(Search(P))))$ as first the refactoring order, which seemed natural because the first three algorithms all use code of *Search* for traversing the vertices of the graph.

While detaching the three aspects *Number*, *Connected*, and *Cycles* we inverted several dependencies between aspects in order not to violate their functional nature. For example, code from *Number* is invoked by the main program P to display the vertex numbers and to trigger the algorithm execution. To reverse this dependency we moved one field to an inter-type declaration and one method call to a piece of advice. Figure 9 shows the (unfactored) class `Vertex` before refactoring and the detached aspect `Number` after refactoring.

Overall, we resolved dependencies at 13 points when factoring out the 4 aspects in order to preserve the function nature (moving 4 fields and 3 methods to inter-type declarations and 6 method calls to pieces of advice).

Furthermore, we resolved several shared join point dependencies: the aspects *Number*, *Connected*, and *Cycles* all advise 2 shared join points in the base program. In Figure 10, we show the code for *Number* as well as for one location of a shared join point. We introduced a hook for each aspect and modified the piece of advice accordingly (Line 11).

Ordering #2

The second ordering is the exact reverse of the first ordering: $Search(Cycles(Connected(Number(P))))$. Especially, the fact that

[15] Both, original and refactored version can be downloaded at http://www.infosun.fim.uni-passau.de/cl/staff/apel/gg4

[16] After refactoring, the program contained 926 LOC, caused by the additional overhead of aspects, i.e., 11 new aspect declarations, each containing package, import, pointcut and advice declarations. It caused by aspects in general, not by functional aspects in particular.

```
1  class Vertex
2    public int num;
3    public void display() {
4      System.out.println(num);
5      System.out.println(comp);
6      System.out.println(cycle);
7    } /* ... */
8  }
```

```
9  aspect Number {
10   public int Vertex.num;
11   before(Vertex v) : this(v) &&
12     execution(void Vertex.display()) {
13       System.out.print(v.num);
14   } /* ... */
15 }
```

Figure 9. Reversing a referential dependency between `Vertex` and `Number` (excerpt).

```
1  class Vertex
2    public void display() {
3      hookDispNum();
4      hookDispConnected();
5      hookDispCycle();
6    } /* ... */
7  }
```

```
8  aspect Number {
9    public int Vertex.num;
10   before(Vertex v) : this(v) &&
11     execution(void Vertex.hookDispNum()) {
12     System.out.print(v.num);
13   } /* ... */
14 }
```

Figure 10. Resolving shared join point dependencies via hooks (excerpt).

the aspects *Number*, *Connected*, and *Cycles* refer all to *Search* makes this ordering unusual. Due to their function nature the aspects cannot refer directly to *Search*. It is interesting to test whether we can transform them by applying pseudo-commutativity and to explore how much code complexity increases. For example, in Figure 11 (Lines 1–6) we exemplify such a reference from aspect *Number* to aspect *Search*. By applying pseudo-commutativity we transformed the code to no longer invoke the method `graphSearch` directly. Instead, the invocation is triggered by the aspect `Search` itself, as shown in Figure 11 (Lines 7–16).

Overall, we transformed all aspects of Ordering #1 to their pseudo-commutative counterparts that are now composed in Ordering #2. All of the 13 referential dependencies are reverted and the 2 shared join point dependencies are resolved as with Ordering #1.

4.1.1 Summary & Discussion

The refactoring detached and encapsulated several design decisions and encapsulates them in aspects. For example, instead of implementing the graph and the main parts of all algorithms with 492 LOC, the class `Graph` only contains the basic graph and benchmark code with 126 LOC and is easier to understand. The behavior of both, the original and the refactored versions, is equivalent.

Within the 4 discussed aspects we applied pseudo-commutative transformations 29 times to resolve referential dependencies and 6 times to resolve dependencies caused by 2 shared join points. This enabled us to reverse the refactoring order without violating the

```
1  /* before resolving reference */
2  aspect Number {
3    void Graph.numVertices() {
4      graphSearch(new NumberWorkSpace());
5    } /* ... */
6  }
```

```
7  /* after resolving reference */
8  aspect Number {
9    void Graph.numVertices() { }
10   /* ... */
11 }
12 aspect Search {
13   before(Graph g) : this(g) &&
14     execution(void Graph.numVertices()) {
15       g.GraphSearch(new NumberWorkSpace());
16   } /* ... */
17 }
```

Figure 11. Resolving referential dependency in Ordering #2 (excerpt).

function composition semantics. Factoring further aspects showed similar results – because their structure is similar. At least for our study, functional aspects are equally flexible to traditional aspects.

It is worth mentioning that we observed that the aspect code of Ordering #2 appears more complex than the code of Ordering #1 due to applying dependency inversion more often (16 instead of 8 times). It is also 15 LOC (2 %) longer. The first ordering appears more natural, i.e., the order in which code would have been incrementally added if we developed it from scratch: the code of *Search* would be implemented first and *Cycles*, *Connected*, and *Number* would be implemented afterward, because they depend on *Search*'s functionality.

Though pseudo-commutativity enables us to permute the refactoring order, there might be orders that are more natural than others, i.e., that minimize the effort in refactoring aspects. We found, a natural order can be determined easily in GraphBenchmark by analyzing the target domain, and a complicated functional aspect implementation might even be considered as an indicator that the current order is far-off a natural order. However, a thorough exploration of these issues should be done in further work.

4.2 Analyzing Existing Aspect-Oriented Programs

To further demonstrate the applicability of pseudo-commutativity and to estimate the maximum effort for a worst case refactoring order with a maximum number of dependency inversion, we analyzed further 4 existing aspect-oriented programs, namely the AspectJ variants of *Berkeley DB*[17] (which we refactored in a previous study [21]), *Prevayler*[18], the *Online Auction System*[19], and *AJHSQLDB*[20]. These programs differ in code size and in their implementation characteristics as summarized in Table 1, which also includes GraphBenchmark as comparison. In the following, we discuss our main observations.

Note, only in Berkeley DB, we actually performed pseudo-commutativity transformation and created functional aspects for one refactoring order. In the remaining three programs, we only analyzed dependencies (references and shared join points). Nevertheless, this is sufficient to determine the maximum effort (worst

[17] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/berkeley/`

[18] `http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem/`

[19] The sources were released kindly by A. Rashid.

[20] `http://sourceforge.net/projects/ajhsqldb/`

case) that would be required if we had to reverse every single dependency, i.e., if we refactored the program in one order and then applied pseudo-commutativity to all aspects to reverse the order.

Usually, when refactoring functional aspects in an order that is close to the natural order [25], the number of required pseudo-commutativity transformations is significantly lower, because many references already point into the correct direction and need not be transformed. Still, the analysis of the maximum number of pseudo-commutativity transformations gives a more accurate impression of the effort required by functional aspects, than the number of transformations for one specific order.

Berkeley DB

Berkeley DB is an embedded database system popular in both commercial and open source applications. To make design decisions explicit and encapsulate their crosscutting implementation we refactored 38 of them into aspects [21]. The size of the detached aspects varies from small (6 LOC) to large aspects (1,867 LOC). Because of their sheer size we split some aspects and grouped them in directories resulting in 151 aspects. Aspects were detached incrementally, while observing dependencies to evaluate function aspects. A comprehensive overview of Berkeley DB and the refactoring process is given elsewhere [23].

Among the 38 features implemented with aspects, we found 53 pairs that interacted by references. Furthermore, there are 28 pairs that interact by overlapping join points. However, this means from 703 (38 choose 2) aspect combinations, only 8 % interact in some form in Berkeley DB. Most pairs of aspects are commutative.

The 53 interactions were caused by references, either by (1) a method that is called from (or introduced by) by another aspect or by (2) a method that is advised by another aspect. To resolve these references, we applied the notion of pseudo-commutativity, even for one cyclic reference. Only in a few cases, we needed additional hooks, e.g., when the method call was in the middle of another method.

During our refactorings we also found 28 shared join points that were advised by 13 aspects. This number of shared join points is low compared to the overall number of 528 join points that have been advised (5.3 %). Moreover, only 4 shared join points were advised by more than 2 aspects. Remarkably, the refactoring and weaving order does not matter in 13 of these shared join points, e.g., because they are just used to initialize independent variables in a constructor. To resolve these interactions we created 38 hook methods.

Online Auction System & AJHSQLDB

The auction system and the embedded database system AJHSQLDB are especially interesting because their aspects do not reference each other. The auction system has only two inter-type declarations and AJHSQLDB uses none. Except for the two inter-type declarations in the auction system, all aspects only contain pointcuts, advice, and private aspect member methods that are only used inside the declaring aspect itself. Both use several homogeneous pieces of advice that advise many join points with the same code. Especially, AJHSQLDB has several pieces of advice that affect over 50 join points (e.g, exception handling, caching, tracing). But even though 3,307 join points have been extended, only 122 have been shared. Of these 122 shared join points the order seems to matter only on 12 cases, because in all other cases the order is not specified in the original implementation using `declare precedence`. In the auction system, there is no `declare precedence` statement at all, so the order does not matter at any shared join point. Thus, the auction system can already be used with functional aspects without changes and AJHSQLDB can easily be adapted to use functional aspects by applying pseudo-commutativity to resolve only 12 shared join points.

Prevayler

Prevayler is an open source persistence layer for Java, that has been decomposed into 55 aspects to encapsulate crosscutting concerns and design decisions. Some of these aspects strongly interact. There are 92 reference dependencies: 7 pieces of advice advise methods introduced by another aspect, 24 extensions advise method calls inside other aspects and 61 method calls reference methods introduced by other aspects. Additionally, there are 14 shared join points. If we implemented all aspects as functional aspects and then applied pseudo-commutativity to every one of them, we had to apply dependency inversion at most 92 times and add hooks at the 14 shared join points.

4.2.1 Summary & Discussion

The analyzed aspect-oriented programs differ in size and their use of aspect-oriented language features. However, the results concerning functional aspects are similar. The number of reference dependencies and shared join points is rather low. The large majority of aspects do not interact, i.e., they do not reference each other and they advise disjoint sets of join points. The remaining interactions can be easily resolved using the notion of pseudo-commutativity. Although using functional aspects requires some additional effort, because some references might need to be reversed and some shared dependencies might need to be resolved, functional aspects do not hinder the refactoring process. Additionally, we have shown that the maximum number (worst case) of pseudo-commutativity transformations needed when applying pseudo-commutativity to all functional aspects is comparably low.

In our theoretical and empirical analysises we did not address the issue whether functional aspects are more or less readable or maintainable than traditional aspects. Though there are some arguments brought forward in prior work [29], we do not consider them definite. Our work provides a theoretical basis that establishes that functional aspects are as expressive as traditional aspects. Empirical studies should follow to compare readability and maintainability. However, the readability and maintainability discussion does not invalidate our results since functional aspects embody a fundamental form of thinking about software development: program extensions (e.g., implemented via aspects) are functions that map programs to programs. This view has been shown useful in stepwise refinement [41], generative programming [8], program families [36], and software product lines [7].

5. Related Work

There are several proposed methods and principles for AOR, e.g., [10, 32, 26]. Furthermore, recent studies explored the benefits and drawbacks of refactoring software into aspects, e.g., [43, 11, 9]. None consider the potentially global effects aspects can have on a program developed in a series of development steps.

A notable exception is the work of Lopez-Herrejon et al. [29]. They propose a fine-grained model for AOP that assumes a function interpretation of aspects. Our work is based on this prior work and can be understood as a case study of function composition and AOR. However, our model is simpler because we do not study the internal structures of aspects. Moreover, we examine the model's properties in the light of AOR.

Two others studies refactor existing aspects of a program into alternative structures by using different modularization mechanisms [27, 42]. There were no aspects that contradict the function model. Our recent studies on the relationship of aspects and program features support this result [21, 3].

Application	LOC	ASP	ADV	INT	AJP	REF	SJP
GraphBenchmark	926	11	21	29	9	12	12
Berkeley DB	39,906	151	482	574	528	53	28
Auction System	1,670	9	18	2	51	0	2
AJHSQLDB	71,027	31	106	0	3,307	0	122
Prevayler	4,362	55	96	112	101	92	14

LOC: Lines of code; ASP: number of aspects; ADV: pieces of advice; INT: number of introductions;
AJP: advised join points; REF: reference dependencies; SJP: shared join points

Table 1. Statistics of five aspect-oriented programs

Several studies criticized the negative effects of global (unbounded) aspect quantification and put forward several solutions. Lopez-Herrejon et al. [29] and McEachen et al. [31] discuss potential fault scenario arising from inadvertent aspect weaving. *Open modules* [1, 34] and *crosscutting interfaces* [16] propose module interfaces that specify explicitly which join points may be advised – the others are hidden. *Harmless advice* is a restricted form of advice that is designed to obey a weak non-interference property [12].

In the light of functional aspects, the proposal of *aspect refinement (AR)* [2] is related to higher-order functions. Since AR enables to transform existing aspects by applying refinements, these refinements can be understood as higher-order aspects, i.e., higher-order functions. Also the close relationship of AR and higher-order pointcuts and advice [39] has been noted [2].

Some studies propose a more general model of associating aspects to development steps [19, 3, 2]. Thereby multiple aspects (and classes) implement the change a development step applies to a program. It is interesting to explore the implications for our model. As mentioned, given multiple aspects per development step there would be two kinds of weaving: (1) *all-at-once weaving*, which weaves all aspects of a development step in one rush and (2) *stepwise weaving*, which weaves the aspects of a series of development steps one after the other. It has been noted that stepwise weaving is more general than all-at-once weaving because it can express all-at-once weaving but not vice versa [29].

By abstracting our formalization from methods and advice to all language constructs that can reference or extend other language constructs, we can easily transfer the idea of pseudo-commutativity to stepwise refinement using other languages or tools like Jak [7] or FSTComposer [4]. In the context of these approaches, the language construct of advice is not available, but methods can be extended by method refinements. At the same time, it formalizes the concept of pseudo-commutativity for *aspectual feature modules* [3].

Finally, our results have been used to explore a notion of pseudo-commutativity in the context of feature interactions [25].

6. Conclusions

In this paper, we explored whether functional aspects are as flexible as traditional aspects with respect to altering the refactoring order. We raised this question because, on one hand, earlier research has shown that treating aspects as functions is beneficial as it reduces program complexity by decreasing the number of potential aspect interactions [29] and aligning with prior work on software design [36, 41, 8, 7, 29], and, on the other hand, it seemed that the traditional model was more flexible with regard to the order in which aspects are factored.

We showed this is not the case: functional aspect interactions caused by references and shared join points can be resolved by pseudo-commutativity, the ability to swap functional aspect composition orders by altering aspect definitions. We explained that every pair of aspects with referential dependencies or overlapping join points can be transformed into a corresponding pseudo-commutative pair. Our work provides a theoretical basis that establishes that functional aspects are as expressive as traditional aspects.

Our case studies support that functional aspects indeed are applicable to AOR. Even though different applications use aspects differently, we could use functional aspects in a straightforward way by applying pseudo-commutativity transformations a few times. Even in our large case studies, the estimated maximum effort for using functional aspects is manageable small.

Our work shows that stepwise refinement does not constrain the known techniques of AOR. It has the same expressiveness than the traditional approach but reduces potential interactions. It is more disciplined with regard to composition.

An avenue of further work is to automate pseudo-commutative transformations. Our formalization and algorithms suggest that this should be possible. Furthermore, empirical studies should follow to compare readability and maintainability of functional and traditional aspects in general and with respect different refactoring orders in particular.

Acknowledgments

We thank Jia Liu for fruitful discussions and useful comments on drafts of this paper. This work was supported in part by the German Research Foundation (DFG), project number AP 206/2-1 and Batory's NSF's Science of Design Project #CCF-0438786 and #CCF-0724979.

References

[1] J. Aldrich. Open Modules: Modular Reasoning about Advice. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 3586 of *LNCS*, pages 144–168. Springer-Verlag, 2005.

[2] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect Refinement - Unifying AOP and Stepwise Refinement. *J. Object Technology – Special Issue: TOOLS EUROPE 2007*, 6(9):13–33, 2007.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[4] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l. Symp. Software Composition*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.

[5] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proc. ECOOP Workshop Aspects, Dependencies, and Interactions*, pages 1–9. Computing Department, Lancaster University, 2006.

[6] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An Extensible AspectJ Compiler. *Trans. Aspect-Oriented Software Development*, 1(1):293–334, 2006.

[7] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[8] I. Baxter. Design Maintenance Systems. *Comm. ACM*, 35(4):73–89, 1992.

[9] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 50–59. ACM Press, 2003.

[10] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 123–134. ACM Press, 2005.

[11] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[12] D. Dantas and D. Walker. Harmless Advice. In *Proc. Int'l. Symp. Principles of Programming Languages*, pages 383–396. ACM Press, 2006.

[13] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, 2002.

[14] R. Filman and D. Friedman. Aspect-Oriented Programming Is Quantification and Obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[16] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[17] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 60–69. ACM Press, 2003.

[18] B. Harbulot and J. Gurd. A Join Point for Loops in AspectJ. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 63–74. ACM Press, 2006.

[19] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. *SIGPLAN Not.*, 37(7):38–45, 2002.

[20] F. Kammüller and H. Sudhof. Composing Safely — A Type System for Aspects. In *Proc. Int'l. Symp. Software Composition*, volume 4954 of *LNCS*, pages 231–247. Springer-Verlag, 2008.

[21] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features using AspectJ. In *Proc. Int'l. Software Product Line Conf.*, pages 222–232. IEEE CS Press, 2007.

[22] C. Kästner, S. Apel, and G. Saake. Implementing Bounded Aspect Quantification in AspectJ. In *Proc. ECOOP RAM-SE Workshop*, pages 111–122. School of Computer Science, University of Magdeburg, 2006.

[23] K. Kästner. Aspect-Oriented Refactoring of Berkeley DB. Diploma thesis, School of Computer Science, University of Magdeburg, 2007.

[24] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice, and Pointcuts. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 3586 of *LNCS*, pages 195–213. Springer-Verlag, 2005.

[25] C. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*. ACM Press, 2008.

[26] R. Laddad. *Aspect-Oriented Refactoring*. Addison-Wesley, 2006.

[27] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proc. Int'l. Conf. Software Engineering*, pages 112–121. ACM Press, 2006.

[28] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l. Conf. Generative and Component-Based Software Engineering*, volume 2186 of *LNCS*, pages 10–24. Springer-Verlag, 2001.

[29] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proc. Int'l. Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 68–77. ACM Press, 2006.

[30] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proc. Asian Symp. Programming Languages and Systems*, volume 2895 of *LNCS*, pages 105–121. Springer-Verlag, 2003.

[31] N. McEachen and R. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 192–200. ACM Press, 2005.

[32] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 111–122. ACM Press, 2005.

[33] M. Nordberg. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Dependency Management, pages 557–584. Addison-Wesley, 2005.

[34] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 39–50. ACM Press, 2006.

[35] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.

[36] D. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Trans. Softw. Eng.*, SE-5(2), 1979.

[37] V. Rajlich. Changing the Paradigm of Software Engineering. *Comm. ACM*, 49(8):67–70, 2006.

[38] M. Rinard, A. Salcianu, and S. Bugrara. A Classification System and Analysis for Aspect-Oriented Programs. In *Proc. Int'l. Symp. Foundations of Software Engineering*, pages 147–158. ACM Press, 2004.

[39] D. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 158–167. ACM Press, 2003.

[40] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *Proc. Int'l. Conf. Functional Programming*, pages 127–139. ACM Press, 2003.

[41] N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14(4):221–227, 1971.

[42] B. Xin, S. McDirmid, E. Eide, and W. Hsieh. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, The University of Utah, 2004.

[43] C. Zhang and H. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 188–205. ACM Press, 2004.