Aspectual Feature Modules

Sven Apel, Thomas Leich, and Gunter Saake, *Member*, *IEEE Computer Society*

Abstract—Two programming paradigms are gaining attention in the overlapping fields of *software product lines (SPLs)* and *incremental software development (ISD)*. *Feature-oriented programming (FOP)* aims at large-scale compositional programming and feature modularity in SPLs using ISD. *Aspect-oriented programming (AOP)* focuses on the modularization of crosscutting concerns in complex software. Although feature modules, the main abstraction mechanisms of FOP, perform well in implementing large-scale software building blocks, they are incapable of modularizing certain kinds of crosscutting concerns. This weakness is exactly the strength of aspects, the main abstraction mechanisms of AOP. We contribute a systematic evaluation and comparison of FOP and AOP. It reveals that aspects and feature modules are complementary techniques. Consequently, we propose the symbiosis of FOP and AOP and *aspectual feature modules (AFMs)*, a programming technique that integrates feature modules and aspects. We provide a set of tools that support implementing AFMs on top of Java and C++. We apply AFMs to a nontrivial case study demonstrating their practical applicability and to justify our design choices.

Index Terms—Feature-oriented programming, aspect-oriented programming, software product lines, incremental software development, collaboration-based design, separation of concerns, crosscutting modularity.

1 Introduction

Software *product lines (SPLs)* [39] and *incremental software development (ISD)* [98] are gaining much attention in software engineering and are subjects of ongoing research. In their overlapping fields, several methods, techniques, and tools have been proposed, for example, *stepwise refinement* [117], *program families* [95], *AHEAD (Algebraic Hierarchical Equations for Application Design)* [24], and *generative programming* [46]. Besides methodological, process-related, and tool-related issues, the capabilities of programming languages are crucial for success in this field. Two novel programming paradigms discussed in this context are *feature-oriented programming (FOP)* [96] and *aspect-oriented programming (AOP)* [63].

FOP aims at the modularity of *features* in SPLs. Programs in SPLs are implemented incrementally by composing code associated with features. FOP has emerged from layered architectures [23] and from domain engineering [46]. It is a paradigm that provides mechanisms for implementing and composing features that represent domain concepts. Features are means to communicate the commonalities and variabilities of the programs or software systems associated with a domain.

There are two key ideas of FOP: 1) Features are mapped one-to-one to modular implementation units called *feature modules* and 2) feature modules incrementally *refine* feature modules already present in a program. Since traditional abstraction mechanisms such as classes are too small units of modularity, a feature module encapsulates a set of classes (and/or class fragments) that define a feature, a so-called *collaboration* [99], [115]. Feature composition is the consistent composition of a feature's structural elements with the elements of other features. Different compositions lead to different programs.

AOP addresses related issues: AOP focuses on localizing, separating, and modularizing crosscutting concerns. A concern is an issue or problem that is of interest to stakeholders. In this sense, concerns are related to features, which we will explore in this paper. Crosscutting concerns are special because their implementation does not align with the hierarchical or block structure of other concerns already present in a program.

Aspects, the main abstraction mechanisms of AOP, encapsulate code that otherwise would be tangled with and scattered across other concern implementations, which is a result of an improper crosscutting modularity [63], [110]. By using aspects, a separation of crosscutting concerns is achieved, which is important for building complex software including SPLs. Although not the initial focus of AOP, several research efforts of the AOP community address SPLs [42], [53], [60], [65], [72], [78] and ISD [13], [77].

1.1 The Relationship of Features and Aspects

In this paper, we explore the relationship of FOP and AOP. At first glance, FOP and AOP differ in their intention. FOP aims at programming language and tool support for domain engineering, that is, at features, which are the fundamental units of abstraction that have a meaning in a domain. AOP aims at the modularization of crosscutting concerns. This also includes concerns that do not usually appear in domain engineering. That is, all features are concerns (possibly crosscutting concerns), but not all concerns are features of a domain.

- S. Apel is with the Department of Informatics and Mathematics, University of Passau, 94030 Passau, Germany. E-mail: apel@infosun.fim.uni-passau.de.
- T. Leich is with the Department of Applied Informatics, Metop Research Institute, Sandtorstrasse 23, 39106 Magdeburg, Germany. E-mail: thomas.leich@metop.de.
- G. Saake is with the School of Computer Science, University of Magdeburg, Universitaetsplatz 2, 39106 Magdeburg, Germany. E-mail: saake@iti.cs.uni-magdeburg.de.

Manuscript received 29 Jan. 2007; revised 2 Aug. 2007; accepted 8 Nov. 2007; published online 16 Nov. 2007.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0024-0107.
Digital Object Identifier no. 10.1109/TSE.2007.70770.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

The different focus of FOP and AOP has led to different terminologies, methods, programming mechanisms, and tools commonly associated with the two programming paradigms. Overall, the situation is quite confusing. We aim at dispelling the confusion by offering a classification of design and implementation problems addressed by FOP and AOP and by a systematic comparison of their capabilities to solve these problems. We compare FOP and AOP at the level of programming language mechanisms, although an isolated discussion is difficult. In order to keep focused, we do not consider developments relating FOP and AOP to software analysis, modeling, and process.

The focus on language mechanisms enables us to compare FOP and AOP on the basis of a small set of evaluation criteria. Of course, some mechanisms are best practice in both paradigms and there are researchers that even classify FOP mechanisms, in particular collaborations, as a subset of AOP [58], [82], [85] or AOP mechanisms, in particular advice, as a subset of FOP [24], [46], [77]. Therefore, we first define what we mean when we talk about FOP mechanisms and AOP mechanisms. Our distinction is motivated by the historical roots of FOP and AOP and by contemporary opinions that are widely shared, as we will explain. Therefore, the results of our analysis should be interpreted in this context.

Commonly, features in FOP are implemented by collaborations. Each feature module encapsulates those fragments of the classes of a program that belong to a feature. FOP decomposes an object-oriented design along a further dimension (beyond classes) into units that are of interest to the stakeholders. Although feature modules are capable of implementing crosscutting concerns, we and others [86] experienced that not all kinds of crosscutting concerns can be modularized appropriately. The classification of crosscutting concerns, which we will present, allows us to understand the nature of these problematic concerns. This is where AOP comes into play.

AOP provides a multitude of mechanisms to modularize crosscutting concerns. Considering all of them here is not possible. Consequently, we limit our attention to what we feel are the most important mechanisms. Our choice is supported by former attempts to capture the essence of AOP [52], [80], [83], [106] and by the fact that *AspectJ*,[1] the most mature and popular AOP language, supports these mechanisms. Particularly, we consider quantification and implicit invocation mechanisms such as pointcuts and advice, as well as static injection mechanisms such as intertype declarations, which we will review in the next section. Note that we do not consider symmetric AOP approaches [57] (for example, *subject-oriented programming* [56], [92] or *aspectual components* [68]) since they are much closer (if not similar) to our notion of FOP than to our notion of AOP. We discuss these, as we call them, *hybrid approaches* in Section 6.

Our study reveals that FOP and AOP provide complementary mechanisms that can profit from each other. Although features are often crosscutting concerns, in general, one cannot implement a feature cohesively and modularly with aspects without restrictions (it may be done in some cases). The difference in the concerns commonly considered in AOP is that features come in large quantities and may have a substantial size. For example, in SPL development, the quantities and sizes of features challenge AOP mechanisms [59], [60]. Aspects are not an adequate alternative of feature modules because, in many cases, multiple aspects and classes are needed to implement a feature with a proper structure [55], [60], [61], [86], [105]. The reason is that AOP mechanisms are not sufficient to express, encapsulate, and compose collaborations of multiple artifacts. Though there are some workarounds (for example, nested aspects and packaging of aspects), they are complicated and unintuitive, as we will explain. The bottom line is that, in order to profit from both FOP and AOP mechanisms, we need to combine them.

1.2 Integrating Features and Aspects

Object orientation decomposes a program along concerns (that may be features), which typically results in a hierarchy of classes. FOP and AOP decompose a program further along those concerns that do not align with the class structure, where decomposition is the intellectual process of structuring software into modules. Feature modules structure an object-oriented design into fragments of classes and their collaborations that belong to a feature. This way, feature modules form a layered architecture that reflects the ISD methodology of FOP. Aspects decompose a program along those concerns that crosscut the class structure and that are supposed to be well modularized. This illustrates that feature-based and aspect-based decomposition may overlap. Nevertheless, the use of aspects only depends on the crosscutting relationship between concerns. If there is no crosscutting, classes and other non-AOP mechanisms are used. Features do not depend on the crosscutting notion. The abstractions of a domain define that features should be implemented modularly. Whether these are crosscutting concerns or not is not important.

Typically, feature-based and aspect-based decomposition lead 1) to a different program structure because of their different intention and 2) to a different implementation because of their different language mechanisms. This can be observed in several case studies on AOP-based (for example, [40], [41], [71], [72], [120]) and FOP-based programs (for example, [23], [35], [66], [112], [119]). Usually, aspects are small units that are used for a few concerns whose implementation would otherwise lead to extreme code scattering and tangling. Features represent any domain abstraction, which results in small-sized, medium-sized, and large-sized feature modules. Using these two kinds of decomposition side by side, aspects might affect code associated with several features and feature modules might contain code from several aspects.

Thinking of aspect-based and feature-based decomposition in this way makes it possible to integrate both. The systematic combination of FOP and AOP leads to a decomposition of software along three dimensions: classes, aspects, and features. Our goal is to eliminate their overlap by assigning aspect-based and feature-based decomposition to different design and implementation problems based on the paradigms' individual strengths and weaknesses. We will show that the problems roughly map to the different

1. http://www.eclipse.org/aspectj/.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

kinds of crosscutting concerns that we have identified. Our key idea is that programmers implement features as units that structure an *aspect-oriented design* instead of an object-oriented design. Feature implementations become enriched by AOP mechanisms because they may contain aspect artifacts. Technically, this means that aspects are integrated into feature modules and work with classes and other aspects in *concert* to implement the features of an SPL.

This kind of integration of feature modules and aspects has two advantages for FOP *and* AOP: 1) an improvement of the crosscutting modularity of features due to the sophisticated modularization mechanisms that aspects provide and 2) an alignment of AOP and ISD achieved by the seamless integration of aspects into the ISD style of feature-based SPLs.

1.3 Contributions

In this paper, we make the following contributions:

- a systematic evaluation and comparison of FOP and AOP, on top of a definition of FOP and AOP, a classification of crosscutting concerns, and a set of evaluation criteria;
- a proposal of the symbiosis of FOP and AOP and a programming technique, called *aspectual feature modules (AFMs)*, that implements the symbiosis differently from previous work;
- a set of programming guidelines for using FOP and AOP mechanisms in concert; and
- an evaluation of AFMs by means of a nontrivial, medium-sized case study that indicates the profitable integration of aspects and feature modules.

This paper revises, extends, and combines our prior work presented in the technical tracks of ICSE '06 [15] and GPCE '06 [8].

2 Background

For a better understanding of the remainder of this paper, we briefly review FOP and AOP, as well as the two representative languages *Jak* [24] and *AspectJ* [62].

2.1 Feature-Oriented Programming

Research on *FOP* investigates the modularity of features in SPLs in which a *feature* is an increment in program functionality [24]. *Feature modules* are modules that realize features at design and implementation levels. Typically, features modules refine the content of other feature modules in an incremental fashion.

An important observation is that features are seldom implemented by single classes, but, instead, are mostly implemented by a whole set of classes that cooperate to complete a task. This coordinated interclass communication is called a *collaboration* [24], [50], [69], [86], [89], [99], [102], [115]. Feature modules facilitate the abstraction, encapsulation, and composition of collaborations.

Classes play different roles in different collaborations [115]. A *role* encapsulates the behavior or functionality that a class provides when a corresponding collaboration with other classes is established, or in the context of FOP, when a corresponding feature is present in a program or software system. That is, a role is the part of a class that implements

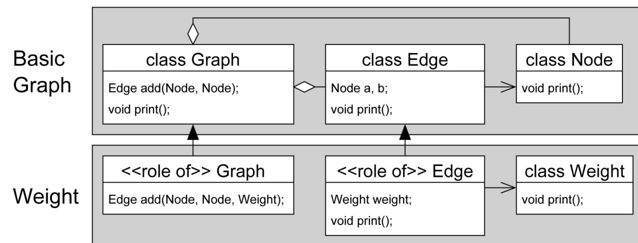

Fig. 1. Collaboration-based design of a graph implementation.

the communication protocol with other classes participating in a particular collaboration. From the FOP perspective, each role is implemented by a refinement. A *refinement* adds new elements to a class and extends existing elements, such as methods. Usually, features extend a program by adding several new classes *and* by applying several new roles to existing classes simultaneously. Hence, the implementation of a feature cuts across several places in the base program.

Fig. 1 depicts the collaboration-based design of a simple program that deals with graph data structures.[2] The feature *BasicGraph* consists of the classes `Graph`, `Node`, and `Edge` that together provide functionality to construct and display graph structures.[3] The feature *Weight* adds roles to `Graph` and to `Edge` and a class `Weight` to implement a weighted graph, that is, a graph that assigns to each edge a specific weight value.

2.2 Jak—FOP for Java

Jak[4] is an extension of Java for FOP. It provides a language construct to express refinements of classes, for example, for implementing roles. Classes in Jak are implemented as standard Java classes. Fig. 2 depicts our feature *BasicGraph* implemented in Jak.[5] It consists of the classes `Graph` (Lines 1-13), `Edge` (Lines 14-18), and `Node` (Lines 19-22). A programmer can add nodes (Lines 3-7) and print out the graph structure (Lines 8-12).

A refinement in Jak encapsulates the changes a feature applies to a class. It is declared by the keyword `refines`. A class composed with a series of refinements forms a new (compound) class. Technically, class refinement can be implemented with various mechanisms (see Section 6).

Fig. 3 depicts our feature *Weight* implemented in Jak: It introduces a class `Weight` (Line 16); it refines the class `Graph` (Lines 1-11) by introducing a new method `add` (Lines 6-10) and by extending an existing method `add` via overriding (Lines 2-5); it refines the class `Edge` (Lines 12-15) by adding a field `weight` (Line 13) and by extending the method `print` (Line 14). The mechanism of extending a method via overriding and invoking `Super`[6] is called a *method extension* (Lines 2-5, 14).

2. The diagram uses the UML notation [30] with some extensions: White boxes represent classes *or* roles; gray boxes denote collaborations; solid arrows denote refinement, that is, to add a new role to a class.

3. We write feature names in *italic* fonts and names of internal elements of features (for example, classes, methods, and fields) in `typewriter` fonts.

4. http://www.cs.utexas.edu/users/schwartz/ATS.html.

5. For brevity, we depict the code of all classes and all refinements belonging to a feature in a single listing.

6. In contrast to Java, the Jak keyword `Super` refers to the class that is refined. For brevity, we write `Super` instead of `Super`($<$ *argument types* $>$), which is actually used in Jak for technical reasons.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

```
 1 class Graph {
 2   Vector nv = new Vector(); Vector ev = new Vector();
 3   Edge add(Node n, Node m) {
 4     Edge e = new Edge(n, m);
 5     nv.add(n); nv.add(m); ev.add(e);
 6     return e;
 7   }
 8   void print() {
 9     for(int i = 0; i < ev.size(); i++) {
10       ((Edge)ev.get(i)).print();
11     }
12   }
13 }
14 class Edge {
15   Node a, b;
16   Edge(Node _a, Node _b) { a = _a; b = _b; }
17   void print() { a.print(); b.print(); }
18 }
19 class Node {
20   int id = 0;
21   void print() { System.out.print(id); }
22 }
```

Fig. 2. A simple graph implementation (*BasicGraph*).

```
 1 refines class Graph {
 2   Edge add(Node n, Node m) {
 3     Edge e = Super.add(n, m);
 4     e.weight = new Weight(); return e;
 5   }
 6   Edge add(Node n, Node m, Weight w) {
 7     Edge e = new Edge(n, m);
 8     nv.add(n); nv.add(m); ev.add(e);
 9     e.weight = w; return e;
10   }
11 }
12 refines class Edge {
13   Weight weight;
14   void print() { Super.print(); weight.print(); }
15 }
16 class Weight { void print() { ... } }
```

Fig. 3. Adding support for weighted graphs via refinements (*Weight*).

A feature module is represented by a containment hierarchy [24]. A *containment hierarchy* contains all artifacts implementing a feature. In Jak, a containment hierarchy is realized by a file system directory, which contains a set of files that store the content of the different artifacts belonging to a feature. Thus, a feature module has no textual representation at the code level. The artifacts, that is, classes and refinements found inside a directory, are the members (assets) of the enclosing feature.

Feature composition superimposes containment hierarchies in which artifacts of the same name and type are composed recursively. Two classes are composed via class refinement and two methods are composed via overriding. For example, the classes `Graph` and `Edge` of feature *BasicGraph* (Fig. 2) are subsequently refined by two refinements of feature *Weight* (Fig. 3). The two refinements override a method each (Lines 2-5 and 14).

2.3 Aspect-Oriented Programming

Aspect-Oriented programming (AOP) is a programming paradigm that aims at the modularization of *crosscutting concerns* (also known as *crosscuts*) [49], [63]. It has been observed that traditional programming paradigms such as OOP do not perform well in modularizing crosscutting concerns [63]. This is because they decompose software along only one (dominant) dimension [110]. The implementation of concerns that do not fit this decomposition leads to *code scattering*, *tangling*, and *replication*.

In our remarks on FOP, we have already considered a kind of crosscutting concern: Collaborations extend a program at different places, thus cutting across the module boundaries introduced by classes. Feature modules implement collaborations. AOP considers crosscutting concerns without special focus on feature modularity or collaborations.

AOP addresses the problems caused by crosscutting concerns as follows: Concerns that can be modularized well using the given decomposition mechanisms of a programming language (also known as *host programming language*) are implemented using these mechanisms. All other concerns that crosscut the implementation of other concerns are implemented as so-called *aspects*. This illustrates that the use of aspects depends on a given decomposition into concern representations.

An aspect enables a programmer to encapsulate code that is associated with one crosscutting concern, thereby eliminating code scattering and tangling. Aspects may affect multiple places in a program via one piece of code, thereby avoiding code replication. An *aspect weaver* merges the separate aspects of a program and the *base program* at predefined *join points*, which is called *aspect weaving*. Join points can be both, syntactical elements of a program (*static join points*) or events in the dynamic execution of the program (*dynamic join points*) [80].

Pointcuts are declarative specifications that select (static and dynamic) join points; pieces of *advice* contain the code that is executed when dynamic join points occur or that is added to static join points in the program. *Intertype declarations* (also known as *static introductions*) inject members statically into existing classes.

Crosscutting concerns that affect static join points are called *static crosscutting concerns* and crosscutting concerns that affect dynamic join points are called *dynamic crosscutting concerns*. Static crosscutting concerns are typically implemented by intertype declarations; dynamic crosscutting concerns are typically implemented by advice.

2.4 AspectJ—AOP for Java

AspectJ is an extension of Java for AOP. In Fig. 4, we depict an aspect `AddColor` that adds facilities for printing colored graph data structures (feature *Color*). It defines an interface `Colored` (Line 2), and it declares that `Node` and `Edge` implement this interface (Line 3); it introduces a field `color`[7] (Line 4), it advises the execution of the method

7. Our notation of intertype declarations differs from AspectJ. The declaration "`int (A || B).i`" means that field `i` is introduced to `A` and `B`.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

```
1 aspect AddColor {
2   interface Colored { ... }
3   declare parents: (Node || Edge) implements Colored;
4   Color (Node || Edge).color = new Color();
5   before(Colored c) : execution(void print()) &&
6     this(c) { Color.setDisplayColor(c.color); }
7   static class Color { ... }
8 }
```

Fig. 4. Adding support for colored graphs via an aspect (*Color*).

```
1 aspect PrintHeader {
2   before() : execution(void *.print()) &&
3     !cflowbelow(execution(void *.print())) {
4       printHeader();
5   }
6   void printHeader() {
7     System.out.print("Graph: ");
8   }
9 }
```

Fig. 5. Advising a method execution dependently on the control flow.

`print` of `Edge` and `Node` (Lines 5-6), and it introduces the class `Color` as a static inner class.

AspectJ provides several mechanisms for the implementation of dynamic crosscutting concerns. For example, in Fig. 5, we depict an aspect that advises the execution of a method only if it occurs in the control flow of another method execution (using `cflowbelow`). This is a dynamic crosscutting concern since it crosscuts the dynamic computation of another concern implementation. It makes the dynamic interaction of two (or more) concerns explicit.

3 Evaluation of FOP and AOP

We use three criteria to evaluate and compare FOP and AOP. The criteria are sufficient to reveal the differences of FOP and AOP.

3.1 Evaluation Criteria

3.1.1 Homogeneous and Heterogeneous Crosscuts

Colyer et al. have introduced the distinction between *homogeneous crosscuts* and *heterogeneous crosscuts* [42].

A homogeneous crosscut extends a program at multiple join points by adding the same *extension*, which is a modular piece of code. In our example, the feature *Color* is a homogeneous crosscut since it introduces the same piece of code to `Edge` and `Node` and extends both their `print` methods.

A heterogeneous crosscut extends multiple join points by adding multiple extensions, where each individual extension is implemented by a distinct piece of code, which affects exactly one join point. In our example, the feature *Weight* is a heterogeneous crosscut since it extends `Graph` and `Edge` at different join points with different pieces of code.

In our evaluation, we examine how FOP and AOP perform with respect to homogeneous and heterogeneous crosscuts.

3.1.2 Static and Dynamic Crosscuts

Mezini and Ostermann have introduced the distinction between *static crosscuts* and *dynamic crosscuts* [86].

A static crosscut extends the structure of a program statically, that is, it adds new classes and interfaces and injects new fields, methods, interfaces, and superclasses, etc. Note that method extensions are not static crosscuts, as we will explain soon. For example, the feature *Weight* introduces one class and injects one method and one field.

A dynamic crosscut affects the runtime control flow of a program. The semantics of a dynamic crosscut can be understood and defined in terms of an event-based model [116]: A dynamic crosscut executes additional code when predefined events occur during the program execution. Such events are dynamic join points [80], [94], [116]. Examples of programming constructs that can implement dynamic crosscuts are Jak method extensions and AspectJ advice.

In our evaluation, we examine how FOP and AOP perform with respect to static and dynamic crosscuts.

3.1.3 Feature Cohesion

Cohesion is the ability of a feature to group and encapsulate all implementation details that define the feature in one unit and to assign a name to it [76], [107]. This enables the programmer to reason about a feature's implementation in one location, separated from other feature implementations. The highest degree of cohesion is achieved by a one-to-one mapping of requirements (features as domain abstractions) to corresponding units at implementation level (features implemented with cohesive modules) [46].

3.2 Evaluation

3.2.1 Homogeneous and Heterogeneous Crosscuts

A significant body of work has shown that collaborations of classes are predominantly of a heterogeneous structure [24], [31], [69], [86], [89], [93], [99], [102], [104], [105], [110], [113], [115]. That is, the roles and classes added to a program differ in their functionality, as in our graph example. A collaboration is a heterogeneous crosscut. A feature module is well qualified to implement a heterogeneous crosscut.

In contrast to feature modules, aspects perform well in extending a set of join points using one cohesive piece of advice or one localized intertype declaration, thus modularizing a homogeneous crosscut. This way, programmers avoid code replication.

Although both approaches support the implementation of crosscuts that the other approach focuses on, they cannot do so elegantly [86]. For example, to implement our feature *Color* (a homogeneous crosscut) in Jak, we would introduce two refinements to the classes `Node` and `Edge`, which contain exactly the same code. Our AspectJ solution proposed previously avoids this code replication (cf., Fig. 4).

Conversely, aspects may implement a collaboration (a heterogeneous crosscut) by bundling intertype declarations and pieces of advice. Actually, there are several ways to do so, which range from implementing one collaboration with one aspect to implementing each role with one aspect.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

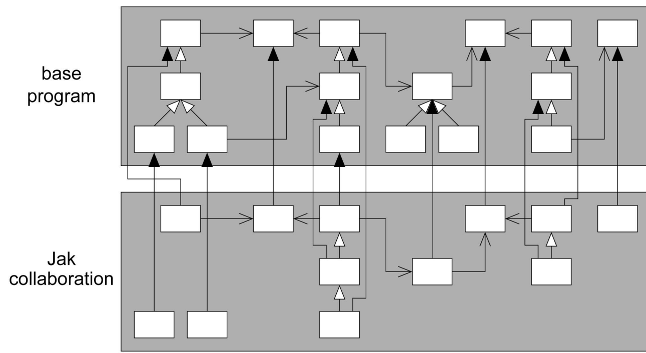

Fig. 6. Implementing a large-scale feature with a feature module.

One may argue that, for our simple example, it does not really matter whether one uses feature modules or aspects. However, the difference between FOP and AOP becomes more apparent when considering features at a larger scale. Suppose a base program consists of many classes and a feature extends most of them. In an FOP solution, the programmer defines, per class to be extended, a new role with the same name (Fig. 6). This way, the programmer is able to retrieve the program structure within the new feature. There is a one-to-one mapping between the structural elements of the base program and the elements of the feature. Of course, this mapping is only relevant for the elements of the base program that are refined.

In an AOP solution, a programmer would either merge all participating roles into one aspect (*first option*) or implement each role with one aspect (*second option*); all cases in between are possible, too, for example, two aspects that implement two roles each.

While implementing a collaboration by a single aspect (first option) is possible [97] (Fig. 7), it flattens the inherent object-oriented structure of the feature and makes it hard to trace the mapping between base program and feature [86], [105]. Note that the difference between FOP and AOP, as shown in Figs. 6 and 7, is not only a matter of visualization. The point is that the inner structure of the aspect does not reflect the structure of the base program; there is a nontrivial mapping between structural elements of the base program and the feature implementation (aspect). The programmer has to trace this mapping continuously, that is, to translate continuously between abstractions of the base program and the feature. The one-to-one mapping of

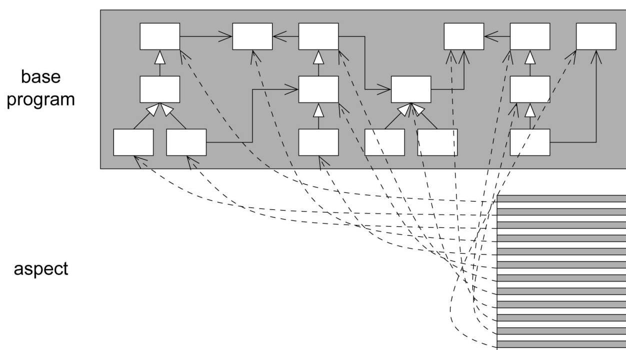

Fig. 7. Implementing a large-scale feature with an aspect.

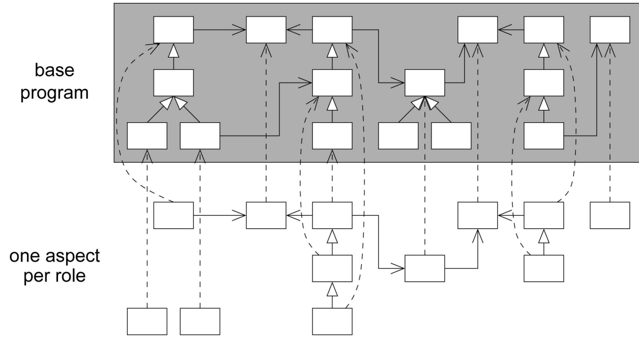

Fig. 8. Using one aspect per role to implement a feature.

the FOP solution is easier to understand, especially for large-scale features.

Implementing each role as a distinct aspect (second option) [55], [61], [100] would enable us to establish a one-to-one mapping between the structural elements of the base program and the elements of the collaboration, provided the programmer follows reasonable naming conventions (Fig. 8). This way, an AOP solution would be very similar to an FOP solution. However, the mapping between classes and roles is not enforced by the programming language and, thus, is left to the discipline of the programmer. This is an important point since the missing linguistic means of AOP languages for expressing collaborations and roles leads to a code in which the role implementations and their interaction tend to vanish.

Furthermore, the way in which inheritance and roles are replaced by aspect weaving does not provide any benefit. It has been argued that such a replacement of object-oriented techniques is questionable [67], [86], [105], especially considering the additional complexity introduced by quantification [3], [106].

The bottom line is that one should use simple mechanisms (class refinement, method overriding) for solving simple problems (heterogeneous crosscuts) and use sophisticated mechanisms (pointcuts and advice) for solving more sophisticated problems (homogeneous crosscuts).

3.2.2 Static and Dynamic Crosscuts

Features and aspects may extend the structure of a base program statically, that is, by injecting new members and introducing new superclasses and interfaces to existing classes. Additionally, features are able to encapsulate and introduce new classes. Traditional aspects, as exemplified by AspectJ, are not able to introduce independent classes—at least not as part of an encapsulated feature (see the discussion below about feature cohesion).[8]

As opposed to AOP, FOP provides no extra language support for implementing dynamic crosscuts. That is, dynamic crosscuts can be implemented, but there are no tailored abstraction mechanisms to express them in an intuitive way, for example, by an event-condition-action

8. Although it is correct that one can just add another class to an environment, for example, using AspectJ, this is at the tool level and not at a model level. The programmer has to build his own mechanisms (outside of the tool) to implement feature modularity [76]. For example, in *FACET*, an AspectJ implementation of a CORBA event channel, the programmers implemented a nontrivial mechanism for feature management [59].

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

```
1  refines class Node {
2    static int count = 0;
3    void print() {
4      if(count == 0)
5        printHeader();
6      count++;
7      Super.print();
8      count--;
9    }
10   void printHeader() { /* ... */ }
11 }
```

Fig. 9. Implementing the extended printing mechanism via refinement.

pattern. Though language abstractions such as `cflow` and `cflowbelow` can be implemented (emulated) by FOP, this usually results in code replication, tangling, and scattering. For example, Fig. 9 depicts our extension of the printing mechanism (cf., Fig. 5) of the class `Node` implemented using FOP; the refinements of the classes `Graph` and `Edge` are analogous. Omitting AOP constructs results in a complicated workaround (underlined) for tracing the control flow (Lines 2, 6, 8) and executing the actual extension conditionally (Lines 4-5). Compared to the FOP solution, the AOP solution captures the intention of the programmer more precisely and explicitly (cf., Fig. 5).

FOP only supports method extensions that are able to implement simple dynamic crosscuts that affect method executions [86]. However, a programmer may want to express a new feature in terms of the dynamic semantics of the base program. Aspects are intended exactly for this kind of crosscut. They provide a sophisticated set of mechanisms to refine a base program based upon its execution, for example, mechanisms for tracing the dynamic control flow and for accessing the runtime context of join points.

The bottom line is that AOP performs well for modularizing *advanced dynamic crosscuts* (that is, dynamic crosscuts that are not method extensions) and FOP only supports method extensions that can be implemented via method overriding.

3.2.3 Feature Cohesion

Features implemented via feature modules have an explicit representation at design and implementation levels. All structural elements that contribute to a feature are encapsulated within a single feature module. Hence, a high degree of feature cohesion is achieved.

Using AOP, a programmer expresses new features by aspects, but, in many cases, features cannot be expressed using one single aspect, especially not in complex programs [76], [86]. Often, the programmer introduces several aspects and additional classes. For example, the feature *Weight* consists of the aspect `AddWeight` and the class `Weight`. One may argue that we could express every feature using only one aspect, but this violates the principle of separation of concerns—it destroys the inner structure of a feature's implementation, as explained above. Classes and aspects are too small units of modularity and, therefore, are not suitable for implementing features [24], [31], [69], [86], [89], [93], [102], [104], [105], [110], [113], [115].

Nevertheless, there are several ways to group collaborations of multiple aspects and classes. For example, aspects could be grouped in a package, in an enclosing aspect or class, or just in a file system directory. However, without FOP, there are no means to compose those constructs to achieve a consistent composition of the features' structural elements. Of course, one could implement all of these things and this has been done in collaboration-based designs for years. The point is that there are solutions available and we show how to reuse them. Note that hybrid approaches like *Caesar* [84], [85], [86] exploit the mechanisms of collaboration-based design such as mixin composition and virtual classes. We discuss them in Section 6.

In summary, feature modules provide appropriate means for the cohesive implementation of program features. An aspect should not implement an entire feature because, in AOP, it is a class-like entity that cannot express a collaboration. Also, a group of aspects and classes is not an option since there are no appropriate mechanisms for their consistent composition.

3.3 Summary, Perspective, and Goals

Table 1 summarizes the results of our comparative evaluation. It reveals that FOP and AOP complement one another. That is, both have strengths where the other is weak. Using both AOP and FOP together offers rewards that neither of them could accomplish in isolation. In our further considerations, we show how these strengths and weaknesses map to programming guidelines that assist programmers in choosing appropriate implementation mechanisms for a given problem.

4 The Symbiosis of FOP and AOP

In this section, we address the following issues: 1) How do we combine FOP and AOP and 2) does their combination outperform FOP and AOP in isolation?

First, we explore the space for achieving a symbiosis of FOP and AOP. Then, we present our approach of integrating feature modules and aspects, which we call *AFMs*.

4.1 Design Space

FOP and AOP can be combined in two ways: 1) Design a programming language that combines the mechanisms of FOP and AOP, which we call an *in-language approach*, and 2) integrate aspects as software artifacts into the development style of FOP and ISD, which we call an *architectural approach*.[9]

The in-language approach enables researchers to explore the language properties of FOP and AOP, as well as their possible integration. In Section 6, we discuss some work that has been taking the in-language approach. As our evaluation will reveal, some language mechanisms of FOP and AOP are redundant. It is an interesting research question as to what a novel language that integrates FOP and AOP should look like, but in an aggregated and

9. Note that it would also be possible to add feature support to AOP, which could be achieved in turn either by an in-language approach or an architectural approach.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

TABLE 1
A Comparison of FOP and AOP with Regard to Our Criteria

evaluation criteria	FOP	AOP
heterogeneous crosscuts	**good support**: feature modules encapsulate and compose collaborations of classes and refinements	**limited support**: aspects bundle sets of intertype declarations and advice, but lack of abstracting and expressing collaborations
homogeneous crosscuts	**no support**: feature modules provides no explicit language constructs for refining multiple join points simultaneously	**good support**: aspects provide wildcards and pattern matching mechanisms to refine multiple join points simultaneously
static crosscuts	**good support**: feature modules can inject new fields, methods, and classes as well as declare new superclasses/interfaces	**limited support**: aspects can inject new fields and methods—but no classes—as well as declare new superclasses/interfaces
dynamic crosscuts	**weak support**: feature modules can implement only basic dynamic crosscuts via overriding (method extensions); there is no support for advanced dynamic crosscuts	**good support**: aspects provide sophisticated mechanisms for advising a program based on its dynamic semantics (method extensions and advanced dynamic crosscuts)
feature cohesion	**high degree**: feature modules encapsulate all artifacts that contribute to a feature	**low degree**: aspects cannot encapsulate collaborations of multiple artifacts that contribute to a feature

stripped-down form. Previous work has not discussed what mechanisms are essential and for which problems which mechanisms should be used. Furthermore, in-language approaches are useful to explore advanced language level issues such as type systems, polymorphism, and soundness. First steps have been made in this direction (Section 6).

The architectural approach takes into account that FOP is also a design method to develop SPLs in an ISD manner. *AHEAD (Algebraic Hierarchical Equations for Application Design)*, an architectural model of FOP, is comprised of all kinds of software artifacts and lays an algebraic foundation for features and ISD [24]. In this sense, aspects are just a new software artifact that should be integrated into the architectural model as well, however, with special characteristics and individual support at the language level. The architectural approach allows us to step back from the implementation and to explore the essential properties of FOP and AOP mechanisms, apart from the specifics of a programming language or environment. First attempts have been undertaken to explore the general characteristics of software composition based on features and aspects using program algebra (Section 6).

Although both approaches promise interesting insights, we concentrate on the architectural approach. One reason is that most of the previous studies have been done in the field of in-language approaches. Another reason is that we are interested in adopting the results of the work on FOP and AOP to other kinds of software artifacts. In principle, software composition based on FOP and AOP should be applicable to any kind of software artifact with a hierarchical structure [24], [38], [110].

In Section 4.5, we compare our architectural approach with an in-language approach and, in Section 6, we review related work on in-language approaches.

4.2 The Integration of Feature Modules and Aspects

When designing and implementing software in a feature-oriented way, a programmer usually starts by modeling and abstracting real-world entities in terms of classes and objects and their collaborations. The result is an object-oriented design (left side in Fig. 10). FOP further structures this design along collaborations that classes undergo. Only the subsets of classes (roles) that participate in a collaboration to implement a certain feature are encapsulated inside the corresponding feature module, that is, features crosscut the object-oriented design (right side in Fig. 10). Subsequent features refine existing features by superimposing their structure [24], [31], [50], [89], [91], [102]. Hence, a feature module is a mechanism that decomposes an object-oriented design along a further dimension, that is, the features of a program.[10]

Our evaluation pointed us to the fact that, in some situations, the implementation of a feature cannot be modularized appropriately by using a traditional feature module. Attempts to do so result in code replication, scattering, and tangling. Typically, these situations are related to crosscutting phenomena. We argue that the shortcomings of FOP revealed by our evaluation are directly responsible for this issue.

To address this issue, we propose to employ AOP since it provides powerful mechanisms to modularize crosscutting concerns. Nevertheless, as our evaluation has revealed, simply using aspects *instead* of feature modules for implementing program features is not appropriate either, for example, because of the lack of feature cohesion and abstraction mechanisms for collaborations. Instead, we

10. Note that decomposition in this context means the intellectual process of a programmer to model real-world entities in objects and to structure this object-oriented design further along features [46].

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

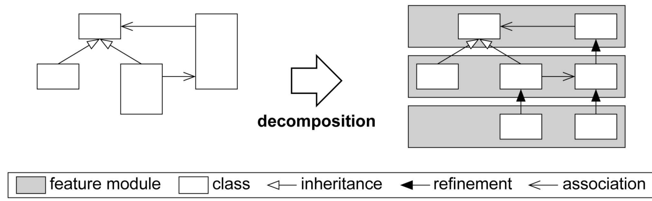

Fig. 10. Feature-oriented decomposition of an object-oriented design.

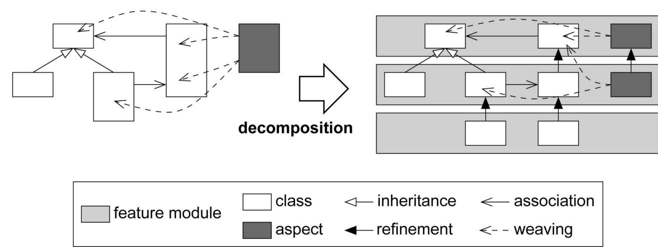

Fig. 11. Feature-oriented decomposition of an aspect-oriented design.

propose using aspects to implement only the concerns that crosscut a given object-oriented design and that cannot be modularized well using feature modules. Thus, a programmer creates an aspect-oriented design, that is, a hierarchy of classes and aspects (left side in Fig. 11).

In order not to forgo the benefits of feature modules, we propose decomposing such aspect-oriented design using the mechanisms of FOP: Although *the aspect-oriented design serves as a substrate, feature modules decompose this design further along the features of the program*. Hence, a feature is implemented by a collaboration of classes *and* aspects (right side in Fig. 11).[11] One benefit of this integration is that we have well encapsulated large-scale feature modules that refine one another incrementally and that dispose of powerful mechanisms for dealing with crosscutting phenomena.

In summary, aspects and feature modules are not competing implementation techniques but decompose a program in different ways. That is, a software is decomposed along three dimensions: classes, aspects, and features. An object-oriented design is the basis; aspects modularize certain kinds of concerns that crosscut the underlying object-oriented design; feature modules decompose the design to impose a structure that is of interest to stakeholders. In this symbiosis, FOP and AOP profit from one another and overcome their individual limitations, as we will illustrate shortly.

4.3 Aspectual Feature Modules

Aspectual feature modules (AFMs) are a concrete approach to implementing the symbiosis of FOP and AOP. AFMs extend the notion of a traditional feature module, for example, known from Jak, by integrating aspects as well as classes and refinements of classes. That is, an AFM encapsulates the roles of collaborating classes *and* aspects that contribute to a feature. Hence, a feature is implemented by a collection of artifacts, among them classes, refinements, and aspects, each artifact appropriate for a specific design or implementation problem.

Note that, typically, an aspect inside a feature module does not implement a role. A refinement is adequate for this task. An aspect is merely a mechanism to extend multiple classes by code that would otherwise be tangled and scattered. In this sense, an aspect may implement a set of roles or fragments of these roles. The advantage of moving away from roles implemented via refinements toward using aspects is to benefit from AOP's capabilities to modularize homogeneous and dynamic crosscutting concerns.

11. Note that the original aspect has been split into two pieces (a base and a subsequent refinement). This is called *aspect refinement* and is not further considered here [13].

Overall, an AFM refines a program in two ways: 1) by using Jak-like refinements or 2) by using aspect-oriented mechanisms, that is, advice and intertype declarations. Probably the most important contribution of AFMs is that programmers may choose the appropriate technique—class refinements or aspects—that fits a given problem best. They can apply a combination of both and decide to what extent either technique is used, according to our programming guidelines (Table 1).

It is worth noting that, much like classes and refinements, an aspect is an integral part of a feature module and is applied and removed together with the feature it belongs to. It may extend classes and refinements of other features already present in a program.

Fig. 12 depicts the feature-oriented design of our graph implementation, consisting of the features *BasicGraph*, *Weight*, and *Color*. *Color* is implemented with an aspect and a class; it is encapsulated in an AFM. As we discussed before, advising executions of the methods `print` in `Node` and `Edge` is a homogeneous crosscut—the same is true for injecting the field `color` to `Node` and `Edge` (cf., Fig. 4). In this situation, it is beneficial to use an aspect because it is able to avoid code replication. Encapsulating the aspect `AddColor` and the class `Color` attains feature cohesion.

As with standard feature modules, an AFM is represented as a containment hierarchy. Besides Java artifacts, it contains also aspects, for example, AspectJ artifacts. After the composition process, we have, in the case of AFMs, a traditional aspect-oriented program (and in the case of traditional feature modules, we have an object-oriented program). Now it becomes clear that it is necessary to weave the aspects and the object-oriented base program in a subsequent step—after the base classes and refinements have been composed. These two steps can be accomplished by different compiler passes or by different tools [7].

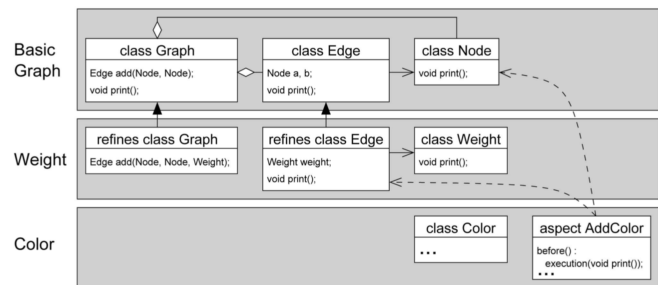

Fig. 12. Implementing the feature *Color* with an aspectual feature module.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

It is worth noting that AFMs are independent of a specific host language. They can be implemented in any pair of feature-oriented and aspect-oriented language that can be woven, for example, Jak and AspectJ, or *FeatureC++* [14] and *AspectC++* [103], etc. Hence, AFMs are invariant with respect to the specifics of the host languages. When the host languages improve, then AFMs improve as well. Thus, AFMs can profit from research on FOP and AOP. With an in-language approach, this would not be possible or only with a major effort.

Nevertheless, once we have chosen a pair of host languages, for example, Jak and AspectJ, AFMs may profit from specific properties of the languages. For example, AspectJ pointcuts may refer to individual roles in a Jak collaboration since Jak uses specific naming conventions. In our graph example, the method `print` of the role of `Edge` in the collaboration *BasicGraph* can be referred to by `Edge_BasicGraph.print`. However, first class language support for advising roles is certainly more appropriate than naming conventions.

4.4 Reviewing the Evaluation Criteria

We apply our evaluation criteria to compare AFMs with traditional FOP and AOP.

4.4.1 Homogeneous and Heterogeneous Crosscuts

The integration of aspects and the traditional constituents of feature modules enables the programmer to choose the appropriate technique for a given problem: The programmer uses aspects to implement homogeneous crosscuts and a set of classes and refinements to implement heterogeneous crosscuts, which are in fact collaborations. As mentioned, this is independent of whether a crosscut is static or dynamic. Aspects, classes, and refinements can be combined at will.

4.4.2 Static and Dynamic Crosscuts

The integration of FOP and AOP allows us to express static crosscuts in two ways using refinements of classes and using intertype declarations in aspects. This introduces a semantic redundancy. As mentioned in the previous paragraph, we propose using aspects to implement static crosscuts that are homogeneous and using refinements to implement static crosscuts that are heterogeneous.

By using aspects, a programmer can implement features depending on the runtime control flow. As with static crosscuts, method extensions can be implemented by aspects (using advice) and by refinements (using method overriding). We handle this analogously to static crosscuts: use aspects for method extensions that are homogeneous and use Jak-like refinements for method extensions that are heterogeneous. Advanced dynamic crosscuts are always implemented using advice because FOP does not provide adequate language mechanisms.

4.4.3 Feature Cohesion

Since we encapsulate aspects in feature modules, we achieve a high degree of feature cohesion. Aspects, as well as their collaborating classes (for example, aspect `AddColor` and class `Color`), are located in one feature module along with other software artifacts. As mentioned, with traditional AOP, programmers would need to provide their own mechanisms for grouping all aspects and classes that belong to a feature (for example, [59]). With AFMs, the programmer is able to explicitly recognize which artifacts belong to a feature, not only at the file system or tool level, but also at the model level.

4.4.4 Summary

The evaluation shows that AFMs outperform FOP and AOP used in isolation by exploiting their strengths. This is embodied in our programming guidelines: 1) use aspects for homogeneous and advanced dynamic crosscuts and 2) use collaborations and refinements for heterogeneous crosscuts and method extensions. An observance of these guidelines improves the crosscutting modularity of AFMs compared to traditional feature modules without destroying the object-oriented structure per se.

4.5 Architectural versus In-Language Approach

Though not the main subject of this paper, we outline the differences in our architectural approach with previous work on in-language approaches, for example, [58], [68], [69], [85].

A first notable difference is that in-language approaches combine several FOP and AOP mechanisms in a single programming language. That is, they unify several language constructs in a well-defined syntax and semantics. Inspired by the work on AHEAD, we chose another approach instead. AFMs are represented by containment hierarchies that may contain different kinds of artifacts, including, among others, classes, refinements, and aspects. Consequently, we do not have a programming language that integrates FOP and AOP mechanisms, but a methodology and framework that handles different artifacts in different ways during composition.

AFMs align well with the AHEAD principles in that they treat classes, refinements, and aspects as different kinds of artifacts that are composed differently. This enables us to use AFMs on top of different languages (C++ and Java are already supported, as explained in Section 4.6). It is even possible to explore principles of FOP and AOP in the composition of noncode artifacts. Therefore, it has been shown that, for the composition of several kinds of noncode artifacts, FOP and AOP mechanisms are useful, for example, for grammar specifications [24], [33], XML documents [6], [112], [118], feature expressions [24], design documents [38], [45], [64], requirement specifications [37], makefiles [24], and feature models [4].

Certainly, in-language approaches can also be migrated to different programming languages, but, since they have special compilers, type systems, etc., this transition is difficult, time consuming, and error prone. The benefit of in-language approaches is exploring advanced programming language concepts such as typing issues and polymorphism inside one well-defined language. AFMs are not intended for such exploration.

A further difference between the architectural approach and the in-language approach is that, in the in-language approach, the actual composition of features is specified inside the source code of the program and, in the architectural approach, outside of the program. With AFMs, to compose a graph application with the features *BasicGraph*, *Weight*, and *Color*, we would specify an external feature expression, which simply enumerates the feature names, for example, we have

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

$$Graph = BasicGraph \bullet Weight \bullet Color.$$

With an in-language approach, the composition would be specified in the source code, for example, we have

collaboration `Graph` **extends** `BasicGraph & Weight & Color`.

The reason for this is that collaborations are first-class language constructs in in-language approaches. In an architectural approach, they are not.

The benefit of an internal specification (in-language approach) is that the runtime entities of a program can reason about collaborations. For example, collaborations could be composed at runtime [44], [93]. However, with this approach, the compositional reasoning and the rest of the program behavior are intermixed. This is similar to metaprogramming in which a metaprogram and a base program are written in the same language and integrated in the same program.

The benefit of an external specification (architectural approach) of a feature composition is that tools and programmers can easily reason about the composition. The reason is that the composition is specified in a separate composition language, as it is good practice in the field of component-based systems [1], [109]. A specification is in some sense an architectural description of a software system. It can itself be a subject of composition and modification [19], [111] and can be optimized on the basis of domain knowledge [24], [46]. A tool just has to interpret and manipulate the specification. This is sometimes called *architectural metaprogramming* [19] since the reasoning about a software system is moved to the architectural level. With an in-language approach, the information about the actual composition is often scattered across the program, for example, in the form of `extends` clauses and/or explicit instantiations of collaborations. Although this allows dynamic reasoning about collaboration, it hinders external compositional reasoning.

In summary, both approaches to the symbiosis of FOP and AOP have their advantages and disadvantages and are useful for exploring different issues. Previous work concentrated on in-language approaches. We chose to explore the architectural approach.

4.6 Tool Support

We provide tool support for AFMs on top of two host programming languages, C++ and Java.

FeatureC++[12] is a language extension of C++ that supports FOP. It comes with a tool for composing feature modules and an FOP compiler for C++ artifacts. Specifically, it introduces class refinement to the C++ language in the form of the syntax presented here, that is, the keywords `refines` and `Super`—with some minor adaptations to the C++ standard. FeatureC++ supports AFMs by integrating *AspectC++* [103] aspects into feature modules.

A further way to implement AFMs is to combine the *AHEAD Tool Suite*[13] and AspectJ. Although the AHEAD Tool Suite is used to compose traditional feature modules, AspectJ weaves the aspects of the individual feature modules to the synthesized class hierarchies.[14] This necessitates some minor tool support (build scripts) and modifications to the aspect code (referring to roles via ⟨class name⟩_⟨collaboration name⟩). The AFMs of our case study are implemented this way.

12. http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/.
13. http://www.cs.utexas.edu/users/schwartz/ATS.html.
14. It is even possible to refine aspects similarly to classes, which is called *aspect refinement* and explained elsewhere [13].

TABLE 2
Aspectual Feature Modules Used in P2P-PL

aspect	description
`Responding`	sends message replies automatically
`Forwarding`	forwards messages to adjacent peers
`MessageHandler`	base aspect for message handling
`Pooling`	stores and reuses open connections
`Serialization`	prepares objects for serialization
`IllegalParameters`	discovers illegal system states
`toString`	introduces several `toString` methods
`LogDebug`	a mix of logging and debugging
`Dissemination`	piggyback meta-data propagation
`FeedbackGenerator`	generates feedback by observing peers
`QueryListener`	waits for query response messages
`CommandLine`	provides command line access
`Caching`	caches peer contact data
`Statistics`	collects and calculates runtime statistics

5 Case Study

This section demonstrates the practical applicability of AFMs to a medium-sized case study. Furthermore, we are interested in how often we need FOP and AOP mechanisms and if the inferred strengths of AFMs pay off in a practical scenario.

5.1 Overview of P2P-PL

As a case study, we use a product line for *peer-to-peer overlay networks (P2P-PL)*, which has been implemented by the first author [11], [12], [34]. Besides the basic functionality as routing and data management in a P2P network [5], P2P-PL supports several advanced features, for example, query optimization, metadata propagation, incentive mechanisms to counter peers that misbehave, called *free riders* [29]. Numerous experiments concerning these features demanded many different configurations to make statements about their specific effects, their variants, and combinations [34]. Hence, P2P-PL seemed to be a good test case for AFMs.

P2P-PL has a fine-grained design and consists of 113 feature modules (traditional and aspectual). P2P-PL was implemented using the AHEAD Tool Suite and AspectJ. As explained in Section 4.6, the AHEAD Tool Suite served for implementing feature modules and AspectJ for composing and weaving aspects within feature modules. The code base of P2P-PL is approximately 6,400 lines of source code.

5.2 Aspectual Feature Modules in P2P-PL

In P2P-PL, 14 of the 113 features (12 percent) use aspects with one aspect per feature (Table 2). The remaining

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

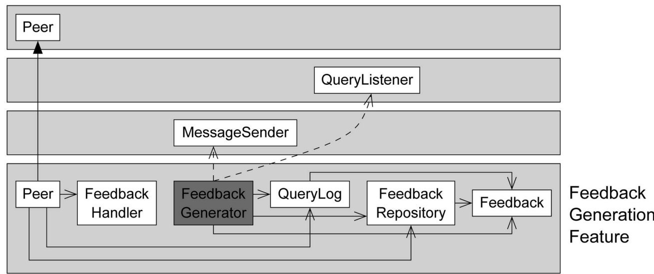

Fig. 13. Feedback generation implemented with an aspectual feature module.

99 features have been implemented with traditional feature modules—without aspects. To give the reader an impression of how aspects and feature modules have been combined in P2P-PL, we explain a simplified example.

5.2.1 An Example—Feedback Generation

The feedback generation feature is part of an incentive mechanism for penalizing *free riders*—peers that profit from the P2P network but do not contribute adequately [29]. The feedback generation feature, on top of the implementation of a peer, identifies free riders by keeping track of whether other peers respond adequately to messages. If this is not the case, an observed peer is considered a free rider. Specifically, the generator observes the traffic of outgoing and incoming messages, and it traces which peers have responded in time. The generator creates positive feedback to reward cooperative peers and negative feedback to penalize free riders. Feedback information is represented by `Feedback` objects and stored in a repository (`FeedbackRepository`); it is passed to other (trusted) peers attached to outgoing messages in order to inform them about free riding. Based on the collected information, a peer judges the cooperativeness of other peers. Messages from peers considered free riders are ignored—only cooperative peers profit by the overall P2P network [29].

The implementation of feedback generation crosscuts the message sending and receiving features of P2P-PL. In Fig. 13, we show an AFM that implements the feedback generation feature. It contains an aspect (dark gray) and introduces four new classes for feedback management. Additionally, it refines the class `Peer` (by a Jak refinement) so that `Peer` owns a log for outgoing queries and a repository for feedback information.

Although the feedback generation feature implements a heterogeneous crosscut, it relies on dynamic context information, that is, it is an advanced dynamic crosscut. Fig. 14 lists an excerpt of the aspect `FeedbackGenerator`. The first piece of advice refines the message sending mechanism by registering outgoing messages in a query log (Lines 2-7). It is executed only if the method `send` was called in the dynamic control flow of the method `forward`. This is expressed using the pointcut `cflow` (Line 5) and avoids advising unintended calls to `send`, which are not triggered by the message forwarding mechanism.[15] The

```
1  aspect FeedbackGenerator { ...
2    after(MessageSender sender, Message msg, PeerId id) :
3      target(sender) && args(msg, id) &&
4      call(boolean MessageSender.send(Message, PeerId)) &&
5      cflow(execution(boolean Forwarding.forward(..))) &&
6      if(msg instanceof QueryRequestMessage)
7      { /* ... */ }
8    after(QueryListener listener) : target(listener) &&
9      execution(void QueryListener.run())
10     { /* ... */ }
11 }
```

Fig. 14. An aspect for feedback generation, part of an AFM (excerpt).

```
1 refines class Peer {
2   FeedbackRepository fr = new FeedbackRepository();
3   QueryLog ql = new QueryLog();
4   Peer() {
5     Super();
6     FeedbackHandler fh = new FeedbackHandler(this);
7     this.getMessageHandler().subscribe(fh);
8   }
9 }
```

Fig. 15. Adding feedback management to the class `Peer`.

second piece of advice intercepts the execution of a query listener task for creating `Feedback` objects (Lines 8-10).

Fig. 15 depicts the refinement of the class `Peer` implemented as a Jak refinement.[16] It adds a feedback repository (Line 2) and a query log (Line 3). Moreover, it refines the constructor by registering a feedback handler in the peer's message handling mechanism (Lines 4-8).

In summary, the feedback generation AFM encapsulates four classes that implement the basic feedback management, an aspect that intercepts the message transfer, and a Jak refinement that extends the class `Peer`. Omitting AOP mechanisms would result in code tangling and scattering since the retrieval of dynamic context information crosscuts other features, for example, clients of the message forwarding mechanism. Implementing this feature as one standalone aspect would not reflect the structure of the P2P-PL framework that includes feedback management. All would be merged in one or more aspect(s) that would decrease program comprehension. Our AFM solution encapsulates all contributing elements cohesively in a collaboration that reflects the intuitive structure of the P2P-PL framework that we had in mind during its design.

5.3 Statistics

5.3.1 Number of Classes, Refinements, and Aspects

The base P2P framework contains only two classes. A fully configured P2P system consists of 127 classes. Thus, refining the base framework into a fully configured system required the incremental introduction of 125 classes. In addition to class introductions, 130 class refinements and

15. The reason for using `cflow` is that the method `send` is called many times inside a peer, but we wanted to advise only those executions of `send` that occur when the peer forwards a message to another peer.

16. The actual syntax for constructor refinement in Jak differs slightly [24].

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

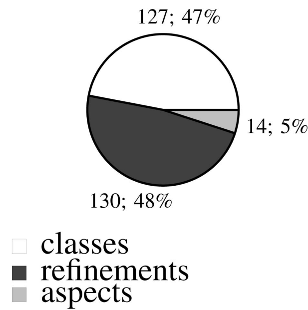

Fig. 16. Number of classes, refinements, and aspects in P2P-PL.

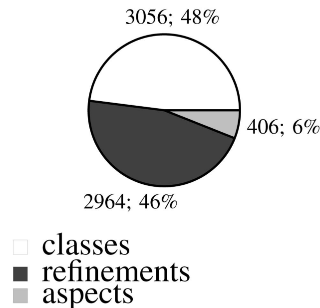

Fig. 17. LOC of classes, refinements, and aspects in P2P-PL.

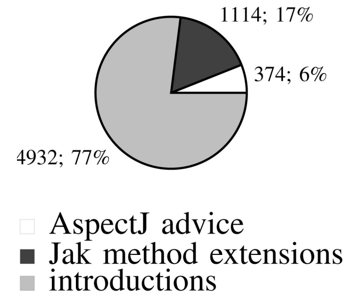

Fig. 18. LOC of static and dynamic crosscuts in P2P-PL.

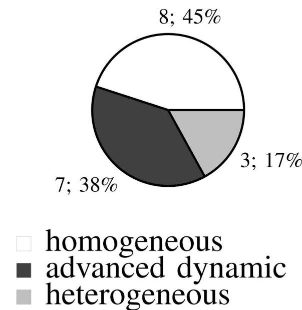

Fig. 19. Number and type of crosscuts implemented by aspects.

14 aspects were used to modularize crosscutting concerns. The main point is that we used primarily classes and refinements rather than aspects for implementing features —about 5 percent of the overall number of mechanisms are aspects (Fig. 16).

5.3.2 LOC Associated with Classes, Refinements, and Aspects

Of the overall P2P-PL code base, aspect code sums up to 6 percent and refinement code to 46 percent and code for classes to 48 percent (Fig. 17). These statistics are in line with the numbers given above on the ratio of implementation mechanism usage.

5.3.3 LOC Associated with Extensions and Introductions

Of the overall code base, 23 percent implement dynamic crosscuts: 6 percent are associated with AspectJ pointcuts and advice and 17 percent with method extensions. The remaining 77 percent are associated with introductions of new functionality (static crosscuts). This suggests that the dominant role of features is to introduce new structures in P2P-PL, rather than extending existing methods or advising join points (Fig. 18).

5.3.4 Number and Properties of Aspects

Of the 14 aspects that were used, eight modularized homogeneous crosscuts, seven aspects implemented advanced dynamic crosscuts (which cannot be implemented by method overriding), and three aspects implemented purely heterogeneous crosscuts (Fig. 19),[17] which we discuss as borderline cases below.

17. In the original study [8], we distinguished between homogeneous crosscuts and those that alter the inheritance relationship. Also note that some aspects were counted for more than one category, for example, homogeneous *and* dynamic.

In summary, 11 of 14 aspects (79 percent) exploit the advanced capabilities of AOP. Using FOP exclusively would result in replicated, scattered, and tangled code, as explained before. Only three aspects implement collaborations that could also be implemented by traditional feature modules. Section 5.4 explains why, in these particular cases, using aspects was appropriate.

5.3.5 Number of Feature-Related Classes and Refinements

With respect to the question of whether aspects are used standalone or with other classes and refinements in concert, we observed that, besides an aspect, each AFM has one to two (up to six) classes and refinements. This demonstrates that AFMs in P2P-PL encapsulate collaborations of aspects, classes, and refinements, rather than single aspects in isolation.

5.4 Lessons Learned

5.4.1 Refinements and Aspects—When to Use What?

A central question for programmers is when to use refinements à la FOP and when to use aspects. Our programming guidelines provide an answer. What we have learned from our case study is that a wide range of problems can be solved by using object-oriented mechanisms and FOP. Specifically, we used FOP for expressing and refining collaborations of classes. Collaborations are typically heterogeneous crosscuts with respect to a base program. Each added feature module reflects a subset of the structure of the base program and adds new and refines existing structural elements. A significant body of prior work advocates this view [24], [31], [50], [69], [86], [89], [93], [102], [104], [105], [110], [113], [115].

Using aspects in isolation for implementing collaboration-based designs, as proposed in [55], [61], [97], [100], was not an option, as explained before.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

Nevertheless, aspects proved to be a useful modularization mechanism. In our study, we have learned that they help in those situations in which traditional OOP and FOP fail:

- By using aspects and their pattern-matching and wildcard mechanisms for homogeneous crosscuts, we could avoid code replication. The aspect-oriented implementation achieves a 5 percent code reduction compared to an equivalent object-oriented or feature-oriented variant.[18]
- In the implementation of seven P2P-PL features, the aspects helped to express the advanced dynamic crosscuts, which are not just method extensions. Aspects perform better in this respect than FOP because they provide sophisticated language-level constructs that capture the programmers intension more precisely and intuitively (for example, `cflow`).

Our case study provides statistics on how often AOP and FOP mechanisms are used. AOP mechanisms were used in 12 percent of all features because they allowed us to avoid code replication, scattering, and tangling. However, aspects occupied only 6 percent of the code base. This is because OOP and FOP mechanisms were sufficient to implement most features (94 percent of the P2P-PL code base). By using AOP for homogeneous crosscuts, we could achieve a code reduction of 5 percent.

We did not address the (important) issue of whether the aspects or feature modules that used standalone would have produced a superior result, which we will address in further work. We merely evaluated the integration of aspects and feature modules and whether the inferred programming guidelines are applicable to a nontrivial software project.

5.4.2 Borderline Cases

We also discovered a few situations in which a decision based on our guidelines was not obvious.

We realized that some homogeneous crosscuts could be modularized alternatively by introducing an abstract base class that encapsulates this common behavior. Although this works, for example, for introducing an integer field for assigning IDs to different types of messages, it does not work for classes that are completely unrelated, as in the case of a logging feature. It is up to the programmer to decide whether the target classes are syntactically and semantically close enough to be grouped via an abstract base class.

Though our study has shown that a traditional collaboration-based design à la FOP works well for most features, we found three heterogeneous features for which it is not clear whether an aspect would not be more intuitive. Two of them are so small that it simply does not matter. The third feature introduces a multitude of `toString` methods to a set of classes. Naturally, each of these methods is implemented differently. Thus, the feature is a heterogeneous crosscut. However, in this particular case, it seems more intuitive to group all `toString` methods in one aspect. We believe that this is caused by the partly homogeneous nature of this crosscut, that is, introducing a set of methods for the same purpose to different classes. Another plausible reason is that this feature is not really a collaboration. That is, the refinements do not add roles to the target classes that establish a collaboration. Exploring this issue will be part of further work.

18. We computed the reduction by subtracting the code size of homogeneous extensions from the hypothetical code size of a set of heterogeneous extensions that implement the same functionality.

6 Related Work

6.1 Implementation of Refinements

Our approach of implementing class refinement is based on Jak and FeatureC++, which use a kind of mixin-based inheritance [32], [114]. We chose a mixin approach because of the tool infrastructure and its success in several domains [10], [20], [21], [27], [35], [66], [115]. Mixin composition is even applicable to the subsequent refinement of aspects [13]. However, we are aware of several alternative mechanisms that might achieve similar results, for example, *virtual classes* [51], [79], *classboxes* [26], *nested inheritance* [87], and *traits* [48].

6.2 Implementation of Feature Modules

Several languages and tools support collaboration-based design. Potentially, all of them could be used to implement feature modules and AFMs; however, each has some limitations.

Several languages support the abstraction and static composition of mixin layers, which are a kind of feature module, for example, *C++ mixin layers* [102], *P++* [101], and *Java layers* [36]. Other approaches exploit related ideas of composing and nesting class hierarchies [43], [50], for example, *Scala* [89], *Jx* [87], *J&* [88], *Classbox/J* [26], *CaesarJ* [17], and *ContextJ* [44], to name a few; all of these are in-language approaches.

A main advantage of AFMs is that they have AHEAD as an architectural model—the approaches mentioned above do not refer to any model. Hence, AFMs build upon the strengths of AHEAD: Aside from classes and aspects, other kinds of software artifacts may also be included in a feature; feature modules are composed declaratively by means of a separate specification (feature expressions) and checked against domain-specific design rules [18]. This opens the door to automatic algebra-based optimization and compositional reasoning [16], [24], [77]. It is not obvious how to carry this over to in-language approaches because the definition of features is done in the same language as their composition. That is, without a separate composition mechanism/language, it is not trivial to implement mechanisms for optimizing and reasoning about composition specifications.

Jiazzi is a component system that supports the composition of collaborations (also in bytecode) via external rules [81]. Since collaborations are represented outside the language, Jiazzi fits the AHEAD architectural model. However, although aspects could possibly be integrated, it is not obvious how to compile them independently, which is a primary goal of Jiazzi.

6.3 Aspects and Collaborations

Several studies suggest exploiting the synergetic potential of mechanisms for aspects and collaborations, for example,

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

Caesar [17], *Aspectual Collaborations* [69], and *Object Teams* [58]. These approaches are closely related to each other since they have common roots [68]. For simplicity, we compare our approach to their general concepts.

All of the approaches mentioned represent collaborations explicitly at the language level and integrate different kinds of mechanisms associated with AOP, for example, pointcuts and advice, *aspectual methods*, *traversals*, *adapters*, and *bindings*. These AOP mechanisms are intended mainly for the modularization of crosscutting concerns that arise from integrating two collaborations. According to the design space of integrating AOP and FOP, the approaches above fall into the first category: They integrate AOP and FOP mechanisms at language level. This is advantageous when exploring issues like typing and polymorphism.

6.4 Multidimensional Separation of Concerns (MDSoC)

MDSoC is a concept and method that aims at the clean separation of multiple, potentially overlapping and interacting concerns simultaneously, with support for on-demand remodularization to encapsulate new concerns at any time [110]. *Hyper/J* supports MDSoC for Java [92]; it introduces the concept of *hyperslices*, which roughly maps to an encapsulated collaboration of classes. It has been observed that features in AHEAD and hyperslices have many commonalities, especially regarding their composition semantics based on superimposition and their mechanisms for composing hyperslices/features [22]. What differs in FOP is that integrating two features that are of a different structure demands a manual integration of the artifacts inside the features, for example, by using wrappers or multiple inheritance [58], [84]. Hyper/J supports declarative composition rules to establish a (possibly complex) mapping between different hyperslices. AHEAD only supports recursive merging of containment hierarchies by type and name.

AFMs, as an extension to traditional feature modules, use aspects to establish the mapping between two unrelated features, as suggested first by Mezini et al. [68], [85]. The code that establishes the mapping is sometimes called *connector* [68] or *adapter* [58]. These are related to the Hyper/J composition rules but at a lower level (programming language level). In this respect, AFMs more closely follow the approach of CaesarJ than of Hyper/J. It remains an open issue which variant of on-demand remodularization and crosscutting integration is preferable.

6.5 Aspects and Information Hiding

One issue of AFMs is that current AOP languages do not respect the principle of information hiding [2], [90], [106], [108]. However, there are several efforts to solve this problem, for example, *open modules* [2], [90], *information hiding interfaces* [54], [108], *harmless advice* [47], or *stratified aspects* [28]. The point here is that AFMs can profit from these developments. Since AFMs do not depend on a specific host language, new languages can easily be integrated. This is a major advantage of AFMs compared to in-language approaches.

6.6 Selected AOP Case Studies

Colyer and Clement refactored an application server using aspects [41]. Specifically, they factored three homogeneous crosscuts and one heterogeneous crosscut. Although the number of aspects is marginal, the size of the case study is impressively high (millions of LOC). Although they draw positive conclusions, they admit (but did not explore) a strong relationship to FOP. This paper demonstrates the useful integration of both worlds.

Zhang and Jacobsen refactored several CORBA ORBs [120]. Using code metrics, they demonstrate that program complexity could be reduced. They propose an incremental process of refactoring which they call *horizontal decomposition*. Liu et al. point out a close relationship to FOP [70]. Our study confirms that, with respect to the implementation of program features, aspects are too small units of modularization [69], [86].

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [40]. They factored four concerns and evolved them in three steps; inherent properties of concerns were not explained in detail.

Kästner et al. refactored the in-memory transactional storage engine Berkeley DB (84 KLOC) into an SPL [60]. Specifically, they detached the code of 38 features and implemented them with AHEAD containment hierarchies. Due to the sheer size of Berkeley DB, several features have been implemented by more than one aspect (overall, 151 aspects). The major fraction of AOP mechanisms used in Berkeley DB implements static and heterogeneous crosscuts and method extensions.

Lohmann et al. examine the applicability of AOP to embedded infrastructure software [71]. They show that AOP mechanisms, if carefully used, do not impose a significant overhead. In their study, they factored in three concerns of a commercial embedded operating system; two concerns were homogeneous and one heterogeneous.

6.7 Selected FOP Case Studies

A significant body of research supports the success of FOP in the implementation of large-scale applications, for example, for the domain of network software [23], databases [23] [25], [66], avionics [20], and command-and-control simulators [21] to mention a few. The AHEAD tool suite is the largest example, with about 100 KLOC [24], [112]. However, none of these studies make quantitative statements about the properties of the implemented features nor do they evaluate the implementation mechanisms used with respect to the structures of the concerns. The features they consider are traditional collaborations with heterogeneous crosscuts, which is in line with our findings in P2P-PL.

Lopez-Herrejon and Batory explore the ability of AOP to implement product lines in an FOP and ISD fashion [73], [75]. They illustrate how collaborations are translated automatically to aspects. They neither address in which situations which implementation technique is most appropriate nor how the aspects generated affect program comprehensibility.

Xin et al. evaluate *Jiazzi* and AspectJ for feature-oriented decomposition [119]. They reimplemented an AspectJ-based

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

CORBA event service [59] by replacing aspects with *Jiazzi units*, which are a form of feature modules. They conclude that Jiazzi provides better support for structuring software and manipulating features, while AspectJ is more suitable for manipulating existing Java code in unanticipated ways. However, they do not examine the structure of the implemented features. Their success in implementing all features of their case study using Jiazzi feature modules hints that most of them (if not all) come in the form of object-oriented collaborations.

We are not aware of further published studies that take both AOP and FOP into account.

7 Conclusion

FOP and AOP are two related programming paradigms discussed in the context of SPLs and ISD. By means of a systematic evaluation, we have revealed that FOP and AOP are complementary paradigms whose combination can overcome their individual limitations. Consequently, we have proposed that AFMs realize the symbiosis of FOP and AOP by integrating aspects into feature modules. This way, the ability of feature modules to modularize crosscutting concerns is improved and aspects become integrated into the stepwise development philosophy of FOP. The evaluation of AFMs has shown that, with regard to our criteria, AFMs perform better than aspects or feature modules in isolation.

We have incorporated the strengths and weaknesses of FOP and AOP into a set of programming guidelines to answer the question of when to use which mechanisms. On one hand, the programmer uses collaborations of classes and refinements in the situations in which they suffice, that is, in implementing heterogeneous and method extensions. On the other hand, the programmer uses aspects to implement homogeneous and advanced dynamic crosscuts, where traditional feature modules fail.

Our case study has demonstrated the practical applicability of the integration of FOP and AOP. We have observed that the dominant role of features is the introduction of new structural elements—adding new classes and new members to existing classes. Refinement of existing methods involved a small fraction of features (17 percent). This is in line with prior studies [73], [75]. Although aspects were used infrequently (6 percent of the code base), they enhanced the crosscutting modularity of features and reduced code replication (by 5 percent). That is, using aspects or feature modules in isolation would not have achieved a clean design or implementation.

The result of our case study provides a single data point. A different line of research confirms our hypothesis that object-oriented collaborations (expressed by classes and refinements) define the predominant way in which features are implemented, where aspects are useful in expressing homogeneous and advanced dynamic crosscuts [7], [9], [74].

In summary, our case study provides supporting evidence that AFMs are applicable to a nontrivial software project and that our programming guidelines assist in choosing and using sufficient implementation mechanisms for a given problem.

Acknowledgments

The authors thank Don Batory, Walter Cazzola, William Cook, Christian Kästner, Christian Lengauer, Roberto Lopez-Herrejon, Olaf Spinczyk, Aleksandra Tesanovic, Sahil Thaker, and Salvador Trujillo for useful comments and fruitful discussions on ideas presented in this article. The presented case study was conducted while Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. The work of Sven Apel and Thomas Leich was funded in part by the German Research Foundation (DFG), project number SA 465/32-1.

References

[1] F. Achermann and O. Nierstrasz, "A Calculus for Reasoning about Software Composition," *Theoretical Computer Science*, vol. 331, nos. 2-3, pp. 367-396, 2005.
[2] J. Aldrich, "Open Modules: Modular Reasoning about Advice," *Proc. European Conf. Object-Oriented Programming*, pp. 144-168, 2005.
[3] R. Alexander, "The Real Costs of Aspect-Oriented Programming," *IEEE Software*, vol. 20, no. 6, pp. 92-93, Nov./Dec. 2003.
[4] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring Product Lines," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 201-210, 2006.
[5] S. Androutsellis-Theotokis and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Computing Surveys*, vol. 36, no. 4, pp. 335-371, 2004.
[6] F. Anfurrutia, O. Díaz, and S. Trujillo, "On Refining XML Artifacts," *Proc. Int'l Conf. Web Eng.*, pp. 473-478, 2007.
[7] S. Apel, "The Role of Features and Aspects in Software Development," PhD dissertation, School of Computer Science, Univ. of Magdeburg, 2007.
[8] S. Apel and D. Batory, "When to Use Features and Aspects? A Case Study," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 59-68, 2006.
[9] S. Apel, D. Batory, and M. Rosenmüller, "On the Structure of Crosscutting Concerns: Using Aspects or Collaborations," *Proc. GPCE Workshop Aspect-Oriented Product Line Eng.*, http://www.softeng.ox.ac.uk/aople/, 2006.
[10] S. Apel and K. Böhm, "Towards the Development of Ubiquitous Middleware Product Lines," *Proc. ASE Workshop Software Eng. and Middleware*, pp. 137-153, 2004.
[11] S. Apel and K. Böhm, "Self-Organization in Overlay Networks," *Proc. CAISE Workshop Adaptive and Self-Managing Enterprise Applications*, vol. 2, pp. 139-153, 2005.
[12] S. Apel and E. Buchmann, "Biology-Inspired Optimizations of Peer-to-Peer Overlay Networks," *Practices in Information Processing and Comm. (Praxis der Informationsverarbeitung und Kommunikation)*, vol. 28, no. 4, pp. 199-205, 2005.
[13] S. Apel, C. Kästner, T. Leich, and G. Saake, "Aspect Refinement—Unifying AOP and Stepwise Refinement," *J. Object Technology (Proc. Int'l Conf. Technology of Object-Oriented Languages and Systems)*, vol. 6, no. 9, pp. 13-33, 2007.
[14] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 125-140, 2005.
[15] S. Apel, T. Leich, and G. Saake, "Aspectual Mixin Layers: Aspects and Features in Concert," *Proc. Int'l Conf. Software Eng.*, pp. 122-131, 2006.
[16] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner, "An Algebra for Feature-Oriented Software Development," Technical Report MIP-0706, Dept. of Informatics and Math., Univ. of Passau, 2007.
[17] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An Overview of CaesarJ," *Trans. Aspect-Oriented Software Development*, vol. 1, no. 1, pp. 135-173, 2006.
[18] D. Batory, "Feature Models, Grammars, and Propositional Formulas," *Proc. Int'l Software Product Line Conf.*, pp. 7-20, 2005.
[19] D. Batory, "From Implementation to Theory in Program Synthesis," *Proc. Int'l Symp. Principles of Programming Languages*, pp. 135-136, 2007.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

[20] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example from Avionics," *Proc. Symp. Software Reusability,* pp. 27-37, 1995.

[21] D. Batory, C. Johnson, B. MacDonald, and D.v. Heeder, "Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 2, pp. 191-214, 2002.

[22] D. Batory, J. Liu, and J. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 48-57, 2003.

[23] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Eng. and Methodology,* vol. 1, no. 4, pp. 355-398, 1992.

[24] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.,* vol. 30, no. 6, pp. 355-371, June 2004.

[25] D. Batory and J. Thomas, "P2: A Lightweight DBMS Generator," *J. Intelligent Information Systems,* vol. 9, no. 2, pp. 107-123, 1997.

[26] A. Bergel, S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 177-189, 2005.

[27] E. Berger, B. Zorn, and K. McKinley, "Composing High-Performance Memory Allocators," *Proc. Int'l Conf. Programming Language Design and Implementation,* pp. 114-124, 2001.

[28] E. Bodden, F. Forster, and F. Steimann, "Avoiding Infinite Recursion with Stratified Aspects," *Proc. Int'l Net.ObjectDays Conf.,* pp. 49-64, 2006.

[29] K. Böhm and E. Buchmann, "Free Riding-Aware Forwarding in Content-Addressable Networks," *VLDB J.,* vol. 16, no. 4, pp. 463-482, 2007.

[30] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide,* second ed. Addison-Wesley, 2005.

[31] J. Bosch, "Super-Imposition: A Component Adaptation Technique," *Information and Software Technology,* vol. 41, no. 5, pp. 257-273, 1999.

[32] G. Bracha and W. Cook, "Mixin-Based Inheritance," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications and the European Conf. Object-Oriented Programming,* pp. 303-311, 1990.

[33] M. Bravenboer and E. Visser, "Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 365-383, 2004.

[34] E. Buchmann, S. Apel, and G. Saake, "Piggyback Meta-Data Propagation in Distributed Hash Tables," *Proc. Int'l Conf. Web Information Systems and Technologies,* pp. 72-79, 2005.

[35] R. Cardone, A. Brown, S. McDirmid, and C. Lin, "Using Mixins to Build Flexible Widgets," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 76-85, 2002.

[36] R. Cardone and C. Lin, "Comparing Frameworks and Layered Refinement," *Proc. Int'l Conf. Software Eng.,* pp. 285-294, 2001.

[37] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters, "Semantics-Based Composition for Aspect-Oriented Requirements Engineering," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 36-48, 2007.

[38] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 325-339, 1999.

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

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

[41] A. Colyer and A. Clement, "Large-Scale AOSD for Middleware," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 56-65, 2004.

[42] A. Colyer, A. Rashid, and G. Blair, "On the Separation of Concerns in Program Families," Technical Report COMP-001-2004, Computing Dept., Lancaster Univ., 2004.

[43] W. Cook, "A Denotational Semantics of Inheritance," PhD dissertation, Dept. of Computer Science, Brown Univ., 1989.

[44] P. Costanza, R. Hirschfeld, and W. de Meuter, "Efficient Layer Activation for Switching Context-Dependent Behavior," *Proc. Joint Modular Languages Conf.,* pp. 84-103, 2006.

[45] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," *Proc. Int'l Conf. Generative Programming and Component Eng.,* pp. 422-437, 2005.

[46] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[47] D. Dantas and D. Walker, "Harmless Advice," *Proc. Int'l Symp. Principles of Programming Languages,* pp. 383-396, 2006.

[48] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black, "Traits: A Mechanism for Fine-Grained Reuse," *ACM Trans. Programming Languages and Systems,* vol. 28, no. 2, pp. 331-388, 2006.

[49] T. Elrad, R. Filman, and A. Bader, "Aspect-Oriented Programming: Introduction," *Comm. ACM,* vol. 44, no. 10, pp. 29-32, 2001.

[50] E. Ernst, "Higher-Order Hierarchies," *Proc. European Conf. Object-Oriented Programming,* pp. 303-329, 2003.

[51] E. Ernst, K. Ostermann, and W. Cook, "A Virtual Class Calculus," *Proc. Int'l Symp. Principles of Programming Languages,* pp. 270-282, 2006.

[52] R. Filman and D. Friedman, "Aspect-Oriented Programming Is Quantification and Obliviousness," *Aspect-Oriented Software Development,* pp. 21-35, Addison-Wesley, 2005.

[53] M. Griss, "Implementing Product Line Features by Composing Aspects," *Proc. Int'l Software Product Line Conf.,* pp. 271-288, 2000.

[54] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software,* vol. 23, no. 1, pp. 51-60, Jan./Feb. 2006.

[55] S. Hanenberg and R. Unland, "Roles and Aspects: Similarities, Differences, and Synergetic Potential," *Proc. Int'l Conf. Object-Oriented Information Systems,* pp. 507-520, 2002.

[56] W. Harrison and H. Ossher, "Subject-Oriented Programming: A Critique of Pure Objects," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 411-428, 1993.

[57] W. Harrison, H. Ossher, and P. Tarr, "Asymmetrically versus Symmetrically Organized Paradigms for Software Composition," Technical Report RC22685 (W0212-147), IBM Research Division, 2002.

[58] S. Herrmann, "Object Teams: Improving Modularity for Crosscutting Collaborations," *Proc. Int'l Net.ObjectDays Conf.,* pp. 248-264, 2002.

[59] F. Hunleth and R. Cytron, "Footprint and Feature Management Using Aspect-Oriented Programming Techniques," *Proc. Joint Conf. Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems,* pp. 38-45, 2002.

[60] C. Kästner, S. Apel, and D. Batory, "A Case Study Implementing Features Using AspectJ," *Proc. Int'l Software Product Line Conf.,* pp. 223-232, 2007.

[61] E. Kendall, "Role Model Designs and Implementations with Aspect-Oriented Programming," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 353-369, 1999.

[62] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," *Proc. European Conf. Object-Oriented Programming,* pp. 327-353, 2001.

[63] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming,* pp. 220-242, 1997.

[64] J. Klein, L. Hélouët, and J.-M. Jézéquel, "Semantic-Based Weaving of Scenarios," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 27-38, 2006.

[65] K. Lee, K. Kang, M. Kim, and S. Park, "Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development," *Proc. Int'l Software Product Line Conf.,* pp. 103-112, 2006.

[66] T. Leich, S. Apel, and G. Saake, "Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager," *Proc. East-European Conf. Advances in Databases and Information Systems,* pp. 324-337, 2005.

[67] K. Lieberherr, "Controlling the Complexity of Software Designs," *Proc. Int'l Conf. Software Eng.,* pp. 2-11, 2004.

[68] K. Lieberherr, D. Lorenz, and M. Mezini, "Programming with Aspectual Components," Technical Report NU-CCS-99-01, College of Computer Science, Northeastern Univ., 1999.

[69] K. Lieberherr, D. Lorenz, and J. Ovlinger, "Aspectual Collaborations—Combining Modules and Aspects," *Computer J.,* vol. 46, no. 5, pp. 542-565, 2003.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

[70] J. Liu, D. Batory, and C. Lengauer, "Feature-Oriented Refactoring of Legacy Applications," *Proc. Int'l Conf. Software Eng.*, pp. 112-121, 2006.

[71] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A Quantitative Analysis of Aspects in the eCos Kernel," *Proc. Int'l EuroSys Conf.*, pp. 191-204, 2006.

[72] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat, "Lean and Efficient System Software Product Lines: Where Aspects Beat Objects," *Trans. Aspect-Oriented Software Development*, vol. 2, no. 1, pp. 227-255, 2006.

[73] R. Lopez-Herrejon, "Understanding Feature Modularity," PhD dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 2006.

[74] R. Lopez-Herrejon and S. Apel, "Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies," *Proc. Int'l Conf. Fundamental Approaches to Software Eng.*, pp. 423-437, 2007.

[75] R. Lopez-Herrejon and D. Batory, "From Crosscutting Concerns to Product Lines: A Function Composition Approach," Technical Report TR-06-24, Dept. of Computer Sciences, Univ. of Texas at Austin, 2006.

[76] R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Technologies," *Proc. European Conf. Object-Oriented Programming*, pp. 169-194, 2005.

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

[78] N. Loughran and A. Rashid, "Framed Aspects: Supporting Variability and Configurability for AOP," *Proc. Int'l Conf. Software Reuse*, pp. 127-140, 2004.

[79] O. Madsen and B. Moller-Pedersen, "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 397-406, 1989.

[80] H. Masuhara and G. Kiczales, "Modeling Crosscutting in Aspect-Oriented Mechanisms," *Proc. European Conf. Object-Oriented Programming*, pp. 2-28, 2003.

[81] S. McDirmid, M. Flatt, and W. Hsieh, "Jiazzi: New-Age Components for Old-Fashioned Java," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 211-222, 2001.

[82] S. McDirmid and W. Hsieh, "Aspect-Oriented Programming with Jiazzi," *Proc. Int'l Conf. Aspect-Oriented Software Development*, pp. 70-79, 2003.

[83] K. Mehner and A. Rashid, "Towards a Generic Model for AOP (GEMA)," Technical Report CSEG/1/03, Computing Dept., Lancaster Univ., 2003.

[84] M. Mezini and K. Ostermann, "Integrating Independent Components with On-Demand Remodularization," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 52-67, 2002.

[85] M. Mezini and K. Ostermann, "Conquering Aspects with Caesar," *Proc. Int'l Conf. Aspect-Oriented Software Development*, pp. 90-100, 2003.

[86] M. Mezini and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects," *Proc. Int'l Symp. Foundations of Software Eng.*, pp. 127-136, 2004.

[87] N. Nystrom, S. Chong, and A. Myers, "Scalable Extensibility via Nested Inheritance," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 99-115, 2004.

[88] N. Nystrom, X. Qi, and A. Myers, "J&: Nested Intersection for Scalable Software Composition," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 21-35, 2006.

[89] M. Odersky and M. Zenger, "Scalable Component Abstractions," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 41-57, 2005.

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

[91] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 25-40, 1992.

[92] H. Ossher and P. Tarr, "Hyper/J: Multi-Dimensional Separation of Concerns for Java," *Proc. Int'l Conf. Software Eng.*, pp. 734-737, 2000.

[93] K. Ostermann, "Dynamically Composable Collaborations with Delegation Layers," *Proc. European Conf. Object-Oriented Programming*, pp. 89-110, 2002.

[94] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive Pointcuts for Increased Modularity," *Proc. European Conf. Object-Oriented Programming*, pp. 214-240, 2005.

[95] D. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Eng.*, vol. 5, no. 2, pp. 264-277, 1979.

[96] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," *Proc. European Conf. Object-Oriented Programming*, pp. 419-443, 1997.

[97] E. Pulvermüller, A. Speck, and A. Rashid, "Implementing Collaboration-Based Design Using Aspect-Oriented Programming," *Proc. Int'l Conf. Technology of Object-Oriented Languages and Systems*, pp. 95-104, 2000.

[98] V. Rajlich, "Changing the Paradigm of Software Engineering," *Comm. ACM*, vol. 49, no. 8, pp. 67-70, 2006.

[99] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems," *J. Object-Oriented Programming*, vol. 5, no. 6, pp. 27-41, 1992.

[100] M. Sihman and S. Katz, "Superimpositions and Aspect-Oriented Programming," *Computer J.*, vol. 46, no. 5, pp. 529-541, 2003.

[101] V. Singhal, "A Programming Language for Writing Domain-Specific Software System Generators," PhD dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1996.

[102] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 2, pp. 215-255, 2002.

[103] O. Spinczyk, D. Lohmann, and M. Urban, "AspectC++: An AOP Extension for C++," *Software Developer's J.*, pp. 68-74, 2005.

[104] F. Steimann, "On the Representation of Roles in Object-Oriented and Conceptual Modeling," *Data and Knowledge Eng.*, vol. 35, no. 1, pp. 83-106, 2000.

[105] F. Steimann, "Domain Models Are Aspect Free," *Proc. Int'l Conf. Model Driven Eng. Languages and Systems*, pp. 171-185, 2005.

[106] F. Steimann, "The Paradoxical Success of Aspect-Oriented Programming," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 481-497, 2006.

[107] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems J.*, vol. 13, no. 2, pp. 115-139, 1974.

[108] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information Hiding Interfaces for Aspect-Oriented Design," *Proc. Int'l Symp. Foundations of Software Eng.*, pp. 166-175, 2005.

[109] C. Szyperski, D. Gruntz, and S. Murer, *Component Software—Beyond Object-Oriented Programming*, second ed. Addison-Wesley/ACM Press, 2002.

[110] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. Int'l Conf. Software Eng.*, pp. 107-119, 1999.

[111] S. Trujillo, M. Azanza, and O. Díaz, "Generative Metaprogramming," *Proc. Int'l Conf. Generative Programming and Component Eng.*, 2007.

[112] S. Trujillo, D. Batory, and O. Díaz, "Feature Refactoring a Multi-Representation Program into a Product Line," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 191-200, 2006.

[113] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen, "Dynamic and Selective Combination of Extensions in Component-Based Applications," *Proc. Int'l Conf. Software Eng.*, pp. 233-242, 2001.

[114] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs," *Proc. JSSST Int'l Symp. Object Technologies for Advanced Software*, pp. 22-37, 1996.

[115] M. VanHilst and D. Notkin, "Using Role Components in Implement Collaboration-Based Designs," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 359-369, 1996.

[116] M. Wand, G. Kiczales, and C. Dutchyn, "A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming," *ACM Trans. Programming Languages and Systems*, vol. 26, no. 5, pp. 890-910, 2004.

[117] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM*, vol. 14, no. 4, pp. 221-227, 1971.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.

[118] E. Wohlstadter and K. De Volder, "Doxpects: Aspects Supporting XML Transformation Interfaces," *Proc. Int'l Conf. Aspect-Oriented Software Development*, pp. 99-108, 2006.
[119] 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, Univ. of Utah, 2004.
[120] C. Zhang and H.-A. Jacobsen, "Resolving Feature Convolution in Middleware Systems," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 188-205, 2004.

Sven Apel received the PhD degree in computer science from the University of Magdeburg in 2007. He is a postdoctoral associate with the Chair of Programming at the University of Passau, Germany. His research interests include advanced programming paradigms, software product lines, and algebra for software construction. For more information on the author, see http://www.infosun.fim.uni-passau.de/cl/staff/apel/.

Thomas Leich is currently working toward the PhD degree in computer science at the University of Magdeburg, Germany. He is the head of the Department of Applied Informatics at the Metop Research Institute, Magdeburg, Germany. His research interests are tailor-made and embedded data management and software product lines. For more information on the author, see http://www.metop.de.

Gunter Saake is a full professor of computer science. He is the head of the Database and Information Systems Group at the University of Magdeburg, Germany. His research interests include database integration, tailor-made data management, object-oriented information systems, and information fusion. He is a member of the IEEE Computer Society. For more information on the author, see http://wwwiti.cs.uni-magdeburg.de/~saake/.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on February 13, 2009 at 13:50 from IEEE Xplore. Restrictions apply.