Aspectual Mixin Layers

Sven Apel, Thomas Leich, and Gunter Saake

Department of Computer Science
University of Magdeburg, Germany
email: {apel,leich,saake}@iti.cs.uni-magdeburg.de

Abstract. *Feature-Oriented Programming* (*FOP*) is an appropriate programming paradigm to implement program families and incremental designs. Beside numerous strengths in doing that FOP yields several drawbacks, especially regarding the ability to localize and modularize crosscutting concerns. This is exactly the strength of another prominent programming paradigm, *Aspect-Oriented Programming* (*AOP*). In this article we contribute a set of evaluation criteria as well as a detailed evaluation and comparison of AOP and FOP. The evaluation criteria are tailored especially for incremental software development. The evaluation reveals that aspects and features are not concurrent concepts. In fact AOP has several strengths to improve FOP in order to implement crosscutting features. Based on these results we introduce the novel notion of *aspectual mixin layers (AMLs)* that integrate AOP concepts into FOP. Our evaluation of AMLs reveals that they improve the crosscutting modularity of features. In a subsequent case study we clarify and discuss the benefits and limitations of AMLs. Finally, we suggest two potential derivates of AMLs that yield different promising properties.

1 Introduction

Program families [28] and incremental software development have a long tradition and are still subject of current research. A main objective of research in this field is to simplify the maintainance, reuse, customization, and evolution of software. Two important programming techniques used to implement complex software in form of program families are *Feature-Oriented Programming (FOP)* [7] and *Aspect-Oriented Programming (AOP)* [15].

FOP was developed to implement software incrementally in a step-wise manner. Features reflect requirements and program characteristics that are of interest to stakeholders. The main idea is that these features are mapped one-to-one to modular implementation units. *Mixin layers* [31, 7] and *collaborations* [19, 26] are most successful in implementing program families in FOP style. *GenVoca* [6] and its successor *AHEAD* [7] are architectural models that describe and unify such feature implementation techniques. In this article we use the term *feature* and *refinement* synonymously and in the sense of FOP and AHEAD: Features are implemented incrementally by extending and refining existent classes and their collaborations. Hence, features have a direct implementation level representation, e.g. mixin layers.

AOP addresses similar issues but with a different focus: AOP focuses mainly on separating and modularizing crosscutting concerns. It introduces *aspects* which encapsulate code that would be otherwise tangled with other concerns and scattered over the base program. Thereby, separation of concerns is achieved that is important to implement complex software, i.e. program families. Although the initial focus does not lie on incremental software development several research efforts go into this direction [22, 26, 9, 24, 19], however, with numerous problems that are discussed in this article.

1.1 The Relationship of Aspects and Features

In this article we explore the relationship of aspects and features. We do not perceive them as concurrent concepts but rather as concepts that can profit from each other. The idea of FOP is to decompose a system architecture into units that are of interest to the stakeholders. Thus, an object-oriented design becomes fragmented. It is decomposed along collaborations of classes that implement features. Such features are best fit to form the basic building block of program families. However, FOP has drawbacks regarding the crosscutting modularity, in particular the ability to localize, separate, and modularize crosscutting concerns [26]. This is where AOP comes into play.

Aspects implement crosscutting concerns but are not adequate to implement all kinds of features. In many cases aspects cannot implement features standalone [23]. Other aspects and additional classes are needed. This is because features are mostly implemented by collaborations and common AOP techniques are not able to express and encapsulate collaborations. A further drawback of current AOP approaches is that aspects only insufficiently support incremental software development. In a nutshell, aspects are problematic in incremental designs because they cannot be bounded to a certain scope and so they may affect inadvertently unanticipated features.

However, we see aspects at the level of classes whereas features organize the architecture at a higher level. Thinking of aspects and features in this way makes it possible to integrate both. Our idea is that a programmer implements features as units that crosscut an *aspect-oriented* architecture. Technically this means that aspects are integrated into collaborations and work with classes and other aspects together to implement features of software.

The close integration of aspects and features holds several advantages to FOP as well as to AOP. Introducing aspects into collaborations improves the FOP's crosscutting modularity. This improvement eases the development of feature-based software. Furthermore, our approach supports the programmer to implement incremental designs using aspects. Due to their integration into a stack of collaborations we were able to bound aspects to certain layers. In short, this bounding capability decreases unpredictable aspect behavior in the face of adding unanticipated features.

Our view on the relationship of aspects and features differs from previous work: *Caesar* [26], *Aspectual Collaborations* [19], and *Object Teams* [12] and their successors are approaches that aim on improving AOP by integrating support

for an incremental development style and for expressing collaborations. These approaches intermix structural elements of AOP and FOP, e.g. introducing keywords for pointcuts or collaborations. Our view is more general and explores the architectural relationship of aspects and features. We found out that there is a natural connection between both. Moreover, we apply our ideas to the AHEAD architectural model that implicates many advantages (see Section 7).

However, our ideas are inspired by previous work. e.g. Caesar and extend our investigations in FeatureC++ a feature-oriented extension to C++ [3]. In contrast to work on FeatureC++, we address general issues that arise from the evaluation and integration of aspects and features. Our results are independent from a specific language and can be seen as an architectural model for integrating AOP and FOP.

1.2 Contributions

This article contributes the following:

- We define a set of criteria for evaluation of modularization techniques focusing on complex software, i.e. program families.
- We use this set of criteria to evaluate the applicability and efficiency of FOP and AOP for implementing complex software with high demands on reusability of artifacts and a high degree of variability of the architecture and implementation. The evaluation reveals several limitations of both programming paradigms, AOP and FOP, that hinder and complicate incremental software development.
- Based on the evaluation we propose *aspectual mixin layers (AMLs)* that integrate FOP and AOP concepts: Whereas collaborations (mixin layers) are used to implement the skeleton of a design, aspects are used to implement certain crosscutting concerns inside features. The evaluation and discussion of AMLs give insight in the relation between features (and their structural elements) and aspects as well as their synergetic potential.
- Furthermore, we present a case study that clarifies the use of AMLs as well as their contribution to FOP/AOP with respect to incremental software development. It serves as a fundament for a discussion about the advantages and disadvantages as well as promising further research.
- Finally, we introduce two variations of AMLs that yield different promising characteristics regarding the relationship of aspects and features.

2 Background

For a better understanding of the remaining article we briefly review FOP and AOP as well as two representative languages, FeatureC++ [3] and *AspectC++* [32].

2.1 Feature-Oriented Programming

FOP studies the modularity of *features* in program families [7]. The idea of FOP is to build software (individual programs) by composing features. Features are basic building blocks and first-class entities in design and implementation. They represent program properties and satisfy intuitive user-formulated requirements on the software system. Features refine other features incrementally. This *stepwise refinement* leads to conceptually layered software designs.

Mixin layers are one appropriate technique to implement features [31, 7]. The basic idea is that features are seldomly implemented by single classes (or aspects). Often, a whole set of *collaborating* classes contribute to a feature. Classes play different *roles* in different *collaborations*. A mixin layer is a static component encapsulating fragments of several different classes (roles) so that all fragments are composed consistently. Advantages are a high degree of modularity and an easy composition [31].

AHEAD is an architectural model for FOP and a basis for large-scale compositional programming [7]. *AHEAD* generalizes the concept of features and feature refinements. Features do not consist of code only but of several types of artifacts, e.g., makefiles, UML-diagrams, documentation. The ideas elaborated in this article follow the AHEAD model.

FeatureC++[1] is a proprietary C++ language extension that supports FOP. Features are implemented by mixin layers. Figure 1 depicts a stack of three mixin layers ($L_1 - L_3$) in top down order. The mixin layers crosscut multiple classes ($C_A - C_C$). The rounded boxes represent the mixins. Mixins that belong to and constitute together a complete class are called refinement chain.

In FeatureC++ mixin layers are represented by file system directories. Therefore, they have no textual representation at code level. Those mixins found inside the directories are assigned to be members of the enclosing mixin layers.

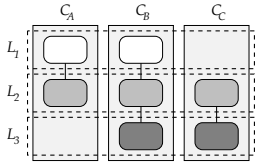

Fig. 1. Stack of mixin layers.

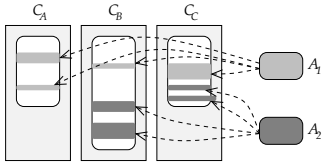

Fig. 2. Aspects extend classes.

Each constant and refinement is implemented as a mixin inside exactly one source file. Constant classes form the root of a refinement chain. Refinements are applied to constants as well as to other refinements. Figure 3 depicts a constant (Line 1) and a refinement (Line 5). Programmers declare refinements using the *refines* keyword. Usually, refinements introduce new attributes and methods

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

```
1  class Buffer {
2    char *buf;
3    void put(char *s) {}
4  };
5  refines class Buffer {
6    int len; int getLength() {}
7    void put(char *s) {
8      if(strlen(s) + len < MAX) super::put(s);
9    }
10 };
```

Fig. 3. Constants and refinements.

(Line 6) or extend[2] methods of their parent classes (see Figure 3, Lines 7-9). To access the extended method the *super* keyword is used (Line 8).

For a more detailed explanation of FeatureC++, its capabilities, and its implementation we refer the interested reader to [3].

2.2 Aspect-Oriented Programming

AOP aims on separating and modularizing crosscutting concerns [15]. Using common object-oriented methods the implementation of crosscutting concerns results in tangled and scattered code [15, 10]. The idea behind AOP is to implement crosscutting concerns as *aspects*. The core features are implemented as components, as in common design and implementation methods. Using *pointcuts* and *advice*, an aspect weaver brings aspects and components together. Pointcuts specify the *join points* of aspects and components, whereas advice define which code is applied to these points. Figure 2 shows two aspects (A_1, A_2) that extend three classes at different join points.

AspectC++[3] is an C++ language extension for AOP. Since it has a similar syntax as the prominent *AspectJ*[4], we omit a detailed introduction and refer the reader to [32].

AspectC++ in a nutshell, Figure 4 shows a logging aspect that intercepts all calls to methods of a buffer class. A pointcut (Line 2) specifies this set of join points and an advice (Lines 4-6) prints out some logging information.

3 Evaluating FOP and AOP

In this section we introduce a set of criteria that forms a basis for a systematic evaluation of modularization techniques with respect to incremental software development. We use these criteria to evaluate and compare FOP and AOP.

[2] We do not use the term 'override' because we want to emphasize that usually method refinements reuse the parent method. This is more an extension than an overriding.

[3] http://www.aspectc.org/

[4] http://eclipse.org/aspectj/

```
1 aspect Logging {
2   pointcut log() = call("% Buffer::%(...)");
3
4   advice log() : before() {
5     cout << JoinPoint::signature() << endl;
6   }
7 };
```

Fig. 4. A logging aspect in AspectC++.

3.1 Evaluation Criteria

The following considerations are based on our experience in building program families [2, 17] as well as on previous work [23, 26, 19]. We are aware that this set of criteria is not complete but we consider only these criteria that are serious to step-wise refinement and in which FOP and AOP differ significantly.

Homogeneous vs. heterogeneous crosscuts. Crosscutting concerns can have two different structures [8]: *Homogeneous concerns* add the same code at different join points whereas *heterogeneous concerns* add different code. A modularization technique should support both types because the former occurs when implementing non-functional features, e.g. tracing, and the latter becomes more important for complex programs, e.g. program families.

Static vs. dynamic crosscutting. Crosscutting concerns affect the base program in two ways: *Static crosscutting* affects the static structure, e.g. static introductions of members in AspectJ or adding classes in AHEAD. *Dynamic crosscutting* depends on and affects the dynamic behavior, e.g. method interceptions. Whereas static crosscutting extends a given program structure, dynamic crosscutting indicates when to apply and execute an extension at runtime.[5] Both types are essential in expressing incremental program refinements because new features usually have to extend the structure *and* the behavior.

Structural dependency. Refinements that depend on the structure of the refined base program are called *hierarchy-conforming*. Such refinements are expressed in terms of the structural elements of the base program. Refinements that alter the structure and rise the abstraction level are *non-hierarchy-conforming*, e.g. a refinement that extends existent classes but introduces new concepts and structural elements that are the base for subsequent features [26]. Non-hierarchy-conforming refinements are important to raise the abstraction level and to reduce complexity during the evolution of complex software systems [26]. They are highly related to the crosscutting phenomena because with common object-oriented techniques several structural elements at different locations are refined using elements that form a new structure.

[5] With dynamic crosscutting we do not refer to dynamic weaving. Whereas the former indicates when to execute/apply an extension during the runtime control flow, the latter is the process of weaving features at load time or runtime.

Whereas the hierarchy-conforming refinements are naturally supported by most language paradigms, e.g. inheritance, it is more difficult to implement non-hierarchy-conforming refinements. Therefore, they should be explicitly supported by modern modularization techniques.

Feature interaction problem. Features interact with each other in many ways. These interactions results in dependencies between different features in this way that features exclude or require other features [13, 34]. Usually the number of interactions grows exponentially with the number of deployed features. This decreases maintainability, variability, and customizability [20]. But even these virtues are especially important for program families.

The *Law of Demeter for Concerns (LoDC)* dictates that features should only know others that contribute to or share their own concerns [18]. This minimizes feature interactions. Advanced modularization techniques have to follow this law by decoupling and scoping features. Since we aim on static customizability and reusability of components at compile time we focus on static and structural interactions that are caused by code modifications and transformations (see also [29, 4]).

Feature optionality problem. This criteria provides information about to what extent a modularization technique is robust against adding or removing optional features [20]. A high tolerance of optional feature compositions leads to a high degree of flexibility and variability. This is a major requirement for incremental designs.

Feature cohesion. Cohesion is the property of a feature to encapsulate all implementation units that contribute to the feature in one module [23]. This eases the maintainability, clearness, and configurability of software. The one-to-one mapping of requirements to implemented features would be an idealized goal [10].

Feature cohesion is the basis for aggregating features to form compound features. Such aggregation mechanism allows to reuse *approved* features in order to construct new ones. This eases the implementation and understanding because thinking in terms of existing features is often easier than building each feature from scratch.

3.2 Evaluation

We evaluate AOP and FOP step-by-step using the evaluation criteria. Due to the limited space, we do not consider hybrid approaches, e.g. Caesar. We elaborate on the relationship to our work in the discussion of related work.

Homogeneous vs. heterogeneous crosscuts. FOP deals with implementing heterogeneous crosscuts: By refining several classes and methods with new classes and methods the programmer applies different code to different join points. It is hard to specify a set of join points and to apply the same code to this set because it is not explicitly supported by common FOP approaches. This results in redundant programming effort and duplicated code. This weakness in modularizing homogeneous crosscuts is the strength of AOP.

With AOP the programmer specifies a set of join points in order to apply one advice. This makes it easy to express homogeneous crosscuts. In exchange, AOP has weaknesses in expressing heterogeneous crosscuts. It is possible to bundle a set of static introductions or pairs of pointcuts and advice, but such aspects do not reflect the structure of the refined feature. That means that the logical structure reflecting the domain knowledge is broken. Another way to implement a heterogeneous crosscut using AOP is to define one aspect per join point. This solution breaks the feature cohesion because the added feature consists of several aspects and classes but they are not encapsulated in one implementation unit. This makes it difficult to switch features on and off. Imagine a program family consiting of 10 features. Each feature is implemented in average by 10 classes/aspects. In order to derive a family member with 5 features the programmer has to manually choose 50 aspects/classes and to remove the remaining 50 aspects/classes, respectively.

Using FOP, features are encapsulated in mixin layers. The programmer simply chooses the desired features in a *declarative* way (by specifying equations of features). FOP expresses heterogeneous crosscuts in a more elegant, simpler, and intuitive way than AOP.

Static vs. dynamic crosscutting. Both, FOP and AOP support static crosscutting, in particular adding methods and attributes. FOP has a more general mechanism for introducting new structural elements. Beside methods and attributes also classes (and other elements) can be introduced. With AOP, the programmer has to add new classes manually.

Furthermore, both paradigms support dynamic crosscutting. Whereas FOP supports only simple method interceptions, AOP can express more advanced dynamic crosscutting, e.g. *cflow*, *if*, *execution* pointcuts. Note that dynamic crosscutting can be implemented using static program transformation, e.g. as in AspectJ or FeatureC++.

Structural dependency. In FOP, features that refine other features have to extend the static structure of the base feature. The programmer is forced to express new features in terms of abstractions of the existent features, e.g. by refining existent classes and methods with new classes and methods. This prevents the raising and altering of the abstraction level, e.g. by introducing new elements on the basis of former elements. Indeed, with FOP new classes can be added, but it is not possible to refine/extend multiple existent classes and to introduce a new concept on top of the refined elements.

AOP allows encapsulating a refinement into aspects. Aspects are able to refine a base program at multiple positions but do not depend on its structural properties. Thus, the programmer can introduce new abstractions that build up on present structures but introduce new concepts (cf. [26]).

Feature interaction problem. To minimize feature interactions in FOP, Prehofer suggests to use *lifters*. Lifters encapsulate code that depends on other features [29]. This results in feature that are independent of other features and feature that depend on others. This reduces the amount of depended code, decouples features, but increases the overall number of implementation

units. Prehofer argues that in real applications this number is manageable and therefore this approach helps the programmer to handle interactions.

AOP has no specific mechanisms or idioms to minimize interactions. The property of aspects to crosscut module boundaries and the absence of a scoping mechanism makes it hard to predict interactions with other features. Solving these problems is subject of current research [11, 22, 16, 1]. Lopez-Herrejon and Batory propose an alternative bounding mechanism that scopes aspects in order to affect only features of current and prior development stages [22]. However, current AOP languages do not follow this principle and complicate the development of incremental designs.

Feature optionality problem. To solve the feature optionality problem in FOP once more lifters can help [29]. As with the feature interaction problem the decoupling of features is essential.

AOP tackles the problem from another side. By expressing join points in pointcuts the programmer is able to skillfully define wildcards that are robust against changes of features and their composition. Furthermore, a lot of research effort is done to provide more advanced and robust join point descriptions, e.g. [27].

Feature cohesion. Features implemented as mixin layers are mapped one-to-one to the implementation level. All structural elements that contribute to the feature are encapsulated inside a mixin layer. Hence, a high degree of feature cohesion is achieved. Features can be composed to form new features. This enables the programmer to generate compound features out of atomic features in order to reuse code.

Using AOP, a programmer expresses new features by introducing aspects and classes. In certain cases features cannot be expressed using one single aspect, especially not in large naturally grown programs [23, 26]. Often, the programmer introduces several aspects and additional classes, e.g. a logging aspect and a class responsible for printing log messages. One may argue that he is able to express every feature using one aspect stand-alone, but we argue that this conflicts with the idea of AOP and separation of concerns. Doing all things in one huge class would also not be a good programming style.

With common AOP the introduced aspects and classes that contribute to a feature are not encapsulated in one implementation unit. This complicates the customization, evolution, and comprehensibility of the overall architecture. Common AOP does not support the explicit expression and encapsulation of collaborations, i.e. classes that collaborate to implement a feature. Thus, AOP has a low degree of feature cohesion and therefore aspects cannot be composed to form compound aspects. This decreases the aspect reuse and hinders building software of large-scale building blocks.

A further benefit of FOP is that feature compositions can be described using algebraic equations. An algebraic model poses as a basis and supports automatic composition and optimization [7]. This is especially important for handling large-scale program families.

3.3 Summary

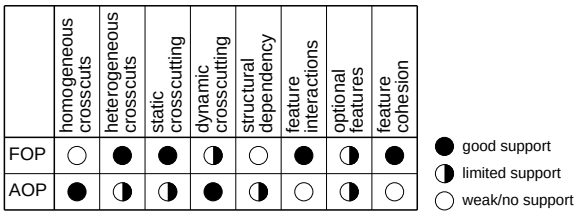

Table 1. Evaluation of FOP and AOP

Table 1 summarizes our evaluation results. As we already explained all these evaluation criteria are crucial to implement incremental designs. Choosing one modularization technique, FOP or AOP, would lead to problems because both have their weaknesses. Table 1 shows that both techniques complement one another, e.g. AOP is strong in modularizing homogeneous crosscuts whereas FOP has its strengths in modularizing heterogeneous crosscuts. Therefore, we propose the integration of both techniques. Whereas FOP helps to decompose a program family in the large, aspects improve the crosscutting modularity of features.

4 Combining Aspects and Features

This section presents AMLs that address several issues discussed in the previous section.

We perceive FOP as a basis methodology for incremental software development using large-scale building blocks. Therefore, we discuss AMLs in terms of how they enhance FOP in improving the crosscutting modularity. Furthermore, we present the new notion of *aspect refinement* and a specific bounding mechanism that follow logically from the integration of AOP and FOP. Afterwards, we evaluate AMLs using our evaluation criteria.

4.1 Aspectual Mixin Layers

The key idea behind AMLs is to embed aspects into mixin layers (see Figure 5). Since mixins implement class fragments the embedded aspects can be seen as fragments, too. FOP with AMLs decomposes an aspect-oriented design into layers that implement features. Thus, each AML contains a set of mixins and a set of aspects. The programmer uses mixins to implement static, heterogeneous, and hierarchy-conforming crosscutting, and he/she uses aspects to express dynamic, homogeneous, and non-hierarchy-conform crosscutting. In other words, mixins refine other mixins and depend, therefore, on the structure of the parent layer.

These refinements follow the static structure of the parent features. Aspects refine a set of parent mixins by intercepting method calls and executions as well as attribute accesses. Therefore, aspects are able to implement advanced dynamic crosscutting and homogeneous, non-hierarchy-conform refinements. With AMLs, the programmer is able to select the adequate technique – mixins or aspects – that fits a given problem best.

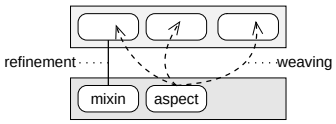

Fig. 5. Aspectual mixin layers.

Figure 6 shows a stack of mixin layers that implements buffer functionality, in particular a basic buffer with iterator, a separated allocator, synchronization, and logging support. Whereas the first three features are implemented using

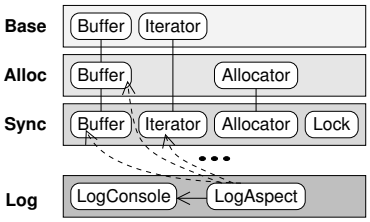

Fig. 6. An AML for logging.

common mixins, the *logging* feature is implemented using mixins *and* aspects. The rationale behind this is that the logging aspect captures a whole set of methods that will be refined (dashed arrows). This refinement is not hierarchy-conforming and depends on the runtime control flow (dynamic crosscutting). Moreover, it modularizes a homogenous crosscutting concern.

4.2 Aspect Refinement

The introduction of AOP techniques and concepts into FOP leads us to the notion of *aspect refinement*. Since aspects are introduced into mixin layers it is natural to refine them incrementally, too. This is complementary to the view that AMLs are units of (de)composition of aspect-oriented architectures. Since in many AOP languages aspects have similar structural elements as classes it is

straightforward to refine these elements in the same way, e.g. introducing methods and attributes or extending methods. Moreover, it becomes possible to refine pointcuts and advice. Refining these aspect-specific elements yields several advantages. By refining a pointcut a programmer can subsequently alter or extend a set of join points. Think of a logging aspect that matches certain points at a given development stage. Introducing new classes leads to modifying the logging aspect to match these new classes. As with classes it is better to refine the logging aspect subsequently than changing the existent aspect. Refining advice is similar to refining methods. In doing that the aspect behavior can be adjusted and extended.

In the following we describe the concept of aspect refinement using FeatureC++: With AMLs aspects can refine other aspects by using the *refines* keyword. To access the methods and attributes of the parent aspect, the refining aspect uses the *super* keyword. Figure 7 shows an AML that refines a logging aspect included in a logging feature by additional join points in order to extend the set of intercepted methods. Beside this, the logging console (implemented as a mixin) is refined by additional functionality, e.g. a modified output format.

Extending pointcuts increases the reuse of existing join point specifications. Note that refining/extending aspects is conceptually different than applying aspects themselves. Applying two aspects modifies the base program in two independent steps. In our logging example this would lead to two different logging instances. Instead, aspect refinement results in two native aspects that are connected via inheritance. Thus, one aspect extends the other and both are applied to the base program. Applied to our logging example, we have only one logging instance.

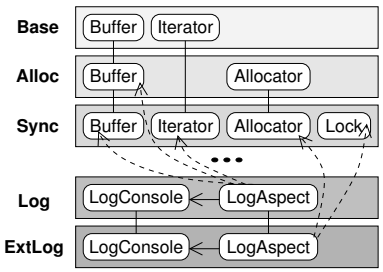

Fig. 7. Refining an AML.

Figure 8 depicts a logging aspect that extends a method of a parent aspect in order to adjust the output format (Line 2) and refines a parent pointcut to extend the set of target join points (Lines 3-4). Both is done using the *super* keyword (Line 2,4).

Since advice is not a first class entity in FeatureC++ and AspectC++ we cannot demonstrate an example of refining an advice. By adopting ideas of

```
1 refines aspect LogAspect {
2   void print() { changeFormat(); super::print();  }
3   pointcut log() = call("% Buffer::put(...)") ||
4                    super::log();
5 };
```

Fig. 8. An aspect embedded into a mixin layer.

classpects that unify methods and advice [30] we would be able to refine advice analogously to methods.

4.3 Bounding Quantification

The close integration of aspects into the incremental development style of FOP and program families leads to a further interesting issue. This integration allows us to tame the unpredictable behavior of aspects.

The problem of current AOP languages is that the binding of aspects is independent of the current development stage. That means an aspect may affect subsequent integrated features although the aspect was implemented without being aware of these features. Moreover, aspects degrade the ability to reason about module interfaces [16]. The programmer may not be aware of the interaction of aspects of previous features with the elements of new features. This can lead to unpredicted effects and errors, e.g. an aspect is unintentionally bound to features of subsequent development stages. Since common AOP languages cannot distinguish between elements of different development stages they cannot scope their appliance. Integrating aspects into incremental designs makes it possible to assign the implementation units to development stages and to define a natural order.

In [22] Lopez-Herrejon and Batory propose an alternative aspect composition mechanism. They argue that with regard to software (program family) evolution, features should only affect features of prior development stages. Mapping this to aspects means that aspects should only affect elements assigned to development stages that were already present at the implementation time of these aspects. Current AOP languages do not follow this principle.

In order to integrate this bounding mechanism into FeatureC++ the user-declared join point specifications must be restructured: Type names in pointcut expressions are translated in order to match only these types that are declared by the current and the parent layers. Each expression that contains a type name is translated into a set of new expressions that refer to all type names of the parent classes. Figure 9 shows a synchronization aspect that is part of an AML. It has two parent layers (*Base*, *Log*) and several child layers. Using this novel bounding mechanism, we transform the aspect and the pointcut as depicted in Figure 10. It can be seen that the new pointcut matches only types of the current and parent layers (Lines 3-5).

This example reflects only our first results. A detailed analysis of complex pointcuts and their transformation is part of further work.

```
1 aspect SyncAspect {
2   pointcut sync() =
3     execution("% Buffer::put(...)");
4 };
```

Fig. 9. A simple pointcut expression.

```
1 aspect SyncAspect_Sync {
2   pointcut sync() =
3     execution("% Buffer_Sync::put(...)") ||
4     execution("% Buffer_Log::put(...)")  ||
5     execution("% Buffer_Base::put(...)");
6 };
```

Fig. 10. Transformed pointcut.

4.4 Evaluation of Aspectual Mixin Layers

Taking the ideas of aspect refinement and bounding quantification into account, we evaluate AMLs using our criteria:

Homogeneous and heterogeneous crosscuts. The integration of aspects and mixins in AMLs enables the programmer to choose the right technique for solving a given problem: The programmer uses aspects to implement homogeneous and mixins to implement heterogeneous crosscuts. Furthermore, it is feasible to combine mixins and aspects in one layer to profit from their collaboration. Aspects and mixins can be refined in subsequent mixin layers.

Static and dynamic crosscutting. The integration of FOP and AOP concepts allows to express static crosscutting in two ways, using mixins and using static introductions in aspects. This introduces a semantic redundancy. As mentioned in the previous paragraph, we suggest to use aspects to implement homogeneous crosscuts and mixins to implement heterogeneous crosscuts, which depend on the structure of the parent feature. Although we have this guideline how and when to use mixins and aspects, a programmer may violate this idiom. In this case, we handle mixins with an higher priority than aspects. The rationale is that we perceive mixins as the skeleton of a software.

Furthermore, the integration of aspects into mixin layers allows expressing advanced dynamic crosscutting. By using pointcuts, e.g. *set*, *get*, *execution*, or *cflow*, a programmer can implement features depending on the runtime control flow, the runtime state, and the current context. As with static crosscutting method extensions can be implemented with aspects (using a *call* pointcut) and mixins (by extending the parent method). We handle this analogously to static crosscutting: using aspects for homogeneous and mixins for heterogeneous crosscuts.

Structural dependency. AMLs can express hierarchy-conforming and non-hierarchy-conforming refinements. AMLs are a super set of mixin layers that can additionally be used to define new features that alter the program structure. AMLs can define aspects that refine entities of the parent features at several locations. Furthermore, they may introduce new concepts that on their part can be refined by other features. It is possible to refine a parent feature by aspects and mixins in concert, e.g. as with the extended logging feature that refines an aspect and a mixin.

Feature interaction problem. FOP solves the problem of unpredictable feature interactions by using lifters. In common AOP languages no such techniques or idioms are intended. Since we do not want to introduce this weakness of AOP into FOP, we utilize a specific bounding quantification mechanism (cf. Section 4.3). It supports a better incremental design and prevents unpredictable aspect composition.

Feature optionality problem. The feature optionality problem is solved by FOP by using lifters. Lifters encapsulate feature dependencies and decouple features. In AOP the well-thought use of wildcards in pointcut designators helps to be reliable against optional features. The integration of mixins and aspects do not worse this problem. On the contrary, the programmer can decide which functionality is implemented using aspects or using mixins. In this way he can select the adequate technique to implement a feature hierarchy that is tolerant and reliable against changes and optional features.

Feature cohesion. Since aspects are encapsulated in mixin layers a high degree of feature cohesion is achieved. Due to the ability to refine aspects they can be reused and combined to form new compound features. Indeed, aspects are encapsulated but still crosscut module boundaries and are not part of the interface of the mixin layer. We refer to *Open Modules* that address this issue [1].

Features that contain aspects can be composed using declarative descriptions. This is an improvement of AOP with respect to incremental software development.

4.5 Summary

Table 2 summarizes the discussion of this section. It compares AMLs with pure FOP and AOP approaches. AMLs are better than FOP and AOP standalone because they combine the advantages of both. However, it is up to the program-

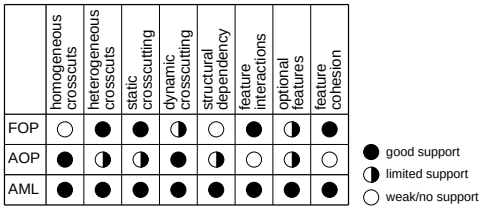

Table 2. Evaluation of AMLs

mer to choose the right techniques to implement a given feature. AMLs are no excuse for bad design or code.

5 Case Study

This section introduces a case study that clarifies the use of AMLs. We choose the stock information broker example, adopted from [26], in order to emphasize the benefits of AMLs compared to common FOP approaches. We show how they perform regarding the criteria discussed in Section 3. Furthermore, the case study serves as a basis for discussing and comparing our ideas. We have implemented the case study using FeatureC++.

5.1 A Stock Information Broker

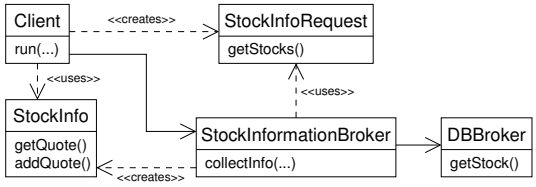

Fig. 11. Stock Information Broker.

A stock information broker provides information about the stock market. The central abstraction is the *StockInformationBroker (SIB)* that allows to lookup for information of a set of stocks (see Figure 11). A *Client* can pass a *StockInfoRequest (SIR)* to the *SIB* by calling the method *collectInfo*. The *SIR* contains the names of all requested stocks. Using the *SIR*, the *SIB* queries the *DBBroker* in order to retrieve the requested information. Then, the *SIB* returns to the client a *StockInfo (SI)* object that contains the stock quotes.

```
1  class StockInformationBroker {
2    DBBroker m_db;
3  public:
4    StockInfo &collectInfo(StockInfoRequest &req) {
5      string *stocks = req.getStocks();
6      StockInfo *info = new StockInfo();
7      for (unsigned int i = 0; i < req.num(); i++)
8        info->addQuote(stocks[i], m_db.get(stocks[i]));
9      return *info; }
10 };
11
12 class Client {
13   StockInformationBroker &m_broker;
14 public:
15   void run(string *stocks, unsigned int num) {
16     StockInfoRequest sir(stocks, num);
17     StockInfo &info = m_broker.collectInfo(sir);
18     /* ... */
19   }
20 };
```

Fig. 12. The basic stock information broker.

All classes are encapsulated in a mixin layer. In other words, this mixin layer implements a basic stock information broker feature. Figure 12 shows a subset of this *base* feature.

A second feature implements a pricing functionality. The *pricing* feature charges the client's account depending on the received stock quotes. A third feature implements an *accounting* functionality that monitors the overall flow of money between client and broker.

5.2 Implementing Refinements

The two features *pricing* and *accounting* shall be implemented as refinements without modifying the *base* feature. The following paragraphs present solutions using common FOP and AMLs.

A Common FOP Solution. Figure 13 depicts the *pricing* feature implemented using common FOP. *Client* is refined by an account management (Lines 17-27), *SIR* is refined by a price calculation (Lines 1-6), and *SIB* charges the account when passing information to the client (Lines 10-14).

The *accounting* feature is depicted in Figure 14. An *Accounting* class (Line 21) stores and manages information about money transfers between client and broker. *Client* and *SIB* are extended by account ids (Lines 2,13). Moreover, they are refined by code that captures transactions that are critical to the money transfer. Corresponding information is passes to the *Accounting* class (Lines 7,17).

There are several problems to this approach: (1) The *pricing* feature is expressed in terms of the structure of the *base* feature. This problem is caused by the inability of FOP to express non-hierarchy-conforming refinements. It would

```
1  refines class StockInfoRequest {
2    float basicPrice();
3    float calculateTax();
4  public:
5    float price();
6  };
7
8  refines class StockInformationBroker {
9  public:
10   StockInfo &collectInfo(Client &c,
11                StockInfoRequest &req) {
12     c.charge(req);
13     return super::collectInfo(req);
14   }
15 };
16
17 refines class Client {
18   float m_balance;
19 public:
20   float balance();
21   void charge(StockInfoRequest &req);
22   void run(string *stocks, unsigned int num) {
23     StockInfo &info =
24       super::m_broker.collectInfo(*this,
25         StockInfoRequest(stocks, num)); .
26   }
27 };
```

Fig. 13. The pricing feature using FOP.

be better to describe the *pricing* feature using abstractions as product, producer, and customer. (2) The interface of *collectInfo* was extended. Therefore, the *Client* must override the method *run* in order to pass a reference of itself to the *SIB*. This is an inelegant workaround, increases the complexity, and violates the principle to do not replace methods of prior development stages. (3) The charging of clients cannot be dynamically altered, e.g. depending on the runtime control flow or the context (e.g. the caller). (4) The *accounting* feature is a homogeneous crosscut that cannot be encapsulated in one location. The introduction of the account ids and the call to the *log* method is redundant in client and broker.

An Aspectual Mixin Layer Solution. Figure 15 depicts the *pricing* feature implemented by an AML. The key difference to the common FOP solution is the *Charging* aspect and the modified *Client* class (*run* is not extended). *SIR* is similar to the FOP version and *SIB* remains unchanged, i.e. it is not extended/refined by a mixin.

The *Charging* aspect intercepts calls to the method *collectInfo* (Lines 2-4) and charges the calling client depending on its request (Lines 6-9). This solves the problem of the extended interface because the client is charged by the aspect instead by the SIB. The client does not need to extend the *run* method.

A further advantage is that the charging of client's accounts can be made dependent to the control flow (using the *cflow* or *if* pointcut). This makes it pos-

```
1  refines class StockInformationBroker {
2    int account;
3  public:
4    StockInfo &collectInfo(Client &c,
5                StockInfoRequest &req) {
6      StockInfo &info = super::collectInfo(req);
7      Accounting::log(account, req.price());
8      return info;
9    }
10 };
11
12 refines class Client {
13   int account;
14 public:
15   void Client::charge(StockInfoRequest &req) {
16     super::charge(req);
17     Accounting::log(account, req.price());
18   }
19 };
20
21 class Accounting {
22   void log(int, float) { /*...*/ }
23 };
```

Fig. 14. The accounting feature using FOP.

sible to implement the charging function variable, e.g., depending on the caller. Finally, our example shows that by using AMLs we are able to refine these classes that play the roles of product (*SIR*) and customer (*Client*). Unfortunately, they do not directly represent the producer role. However, AMLs allow to alter the abstraction level and to implement certain non-hierarchy-conform refinements.

The *accounting* feature is implemented using a simple aspect (see Figure 16). The account ids are added using static introductions (Lines 2-3). A pointcut specifies the target methods (Line 5-6) and an advice adds calls to the *Accounting* class (Line 11). To overcome the problem of the different signatures of *charge* and *collectInfo* we use the AspectC++'s reflective API as well as its static typing support [21].

This solution implements the *accounting* feature, which is a homogeneous crosscut, in an elegant way, using an aspect and a class but encapsulated in a cohesive feature.

6 Discussion

AMLs highlight one way to integrate aspects and features. Our investigations reveal that there is no reason to perceive aspects and features as concurrent concepts. We believe they complement each other.

AMLs contribute novel ideas to FOP in order to improve the crosscutting modularity. AMLs tackle the FOP problems by introducing aspects into mixin layers. This enables the programmer to choose the adequate technique for a given problem: On the one hand, he uses aspects for implementing homogeneous cross-

```
aspect Charging {
  pointcut collect(Client &c, StockInfoRequest &req) =
    call("% StockInformationBroker::collectInfo(...)")
    && args(req) && that(c);

  advice collect(c, req) :
    after(Client &c, StockInfoRequest &req) {
      c.charge(req);
  }
};

refines class Client {
  float m_balance;
public:
  float balance();
  void charge(StockInfoRequest &req);
};
```

Fig. 15. The pricing feature using AMLs.

```
aspect AccountingAspect {
  pointcut id() = "Client" || "StockInformationBroker";
  advice id() : int account;

  pointcut transfers() =
    call("%%::collect%(...)" || "%%::charge(...)");

  advice transfers() : after() {
    StockInfoRequest &req =
      *(StockInfoRequest*)tjp->arg(JoinPoint::ARGS-1);
    Accounting::log(tjp->that().account, req.price());
  }
};

class Accounting {
  void log(int, float) { /*...*/ }
};
```

Fig. 16. The accounting feature using AMLs.

cuts, advanced dynamic crosscutting, as well as features that do not follow the structure of the base elements. Note that the programmer does not use aspects stand-alone but encapsulated in AMLs with mixins in collaboration. On the other hand, the programmer uses common mixins to implement heterogeneous crosscuts that can be expressed in terms of the structure of the base features. The integration of aspects into the incremental development model allows the refinement of aspects similar to classes. Furthermore, we are able to bound their effects to a predefined set of features.

However, the case study has shown that AMLs where not able to express the *pricing* feature in terms of product, producer, and customer. This is because aspectual entities as pointcuts cannot be assigned to mixins, e.g. the *SIB*. A possible solution would be to assign pointcuts and advice to mixins. Thereby,

mixin refine other mixins additionally by introducing pointcuts and advice. [3] discusses such potential variations of the AML approach in the context of FeatureC++..

A different point of view is that FOP using AMLs (de)composes aspect-oriented architecture in order to achieve feature modularity. The introduction of aspects has led us to the proposal of aspect refinement. Since aspects are encapsulated in mixin layers they can be refined, too. Refining aspects leads to the observation that it would be useful to refine pointcuts and advice, besides methods. Since advice are not first-class entities in common AOP approaches we focus preliminarily on pointcut refinements. Section 4 has shown that there are indeed certain applications of this concept.

Finally, the new bounding mechanism allows to scope aspects and to prevent unpredicted aspect interactions and bindings. This is not possible with common AOP languages because the order of refinements cannot be inferred from the program structure, e.g. the classes and aspects. So the integration of aspects into features makes it possible for the first time to bound aspects based on their affiliation to a development stage. This is an important contribution to apply aspects to incremental program family development.

6.1 Extensions and Modifications

During our work on AMLs we have collected several possible extensions and modifications of our ideas to integrate AOP concepts into FOP. The remaining section describes *multi mixins (MMs)* and *aspectual mixins (AMs)* that contribute new ideas to the symbiosis of AOP and FOP. However, the following considerations are of conceptual nature and are not implemented in FeatureC++ nor in another language.

Aspectual Mixins. The idea of AMs is to apply AOP language concepts directly to mixins (see Figure 17). In this approach, mixins refine other mixins as with common FOP, but they also define pointcuts and advices. In other words,

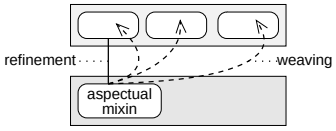

Fig. 17. Aspectual mixins: Unifying aspects and mixins.

aspectual mixins are similar to aspectual mixin layers but integrate pointcuts and advices directly into mixins. The rationale behind this is that in some cases an aspects have to refine directly mixins. Recall the case study where AMLs where not able to assign the producer role to the *SIB* class. Instead, a separate

```
1  refines class StockInformationBroker {
2    pointcut collect(Client &c, StockInfoRequest &req) =
3      call("% StockInformationBroker::collectInfo(...)")
4      && args(req) && that(c);
5    advice collect(c, req) :
6      after(Client &c, StockInfoRequest &req) {
7        c.charge(req);
8    }
9    int broker_id;
10   TransferMap map;
11 };
```

Fig. 18. Aspectual mixins.

aspects encapsulates the charging code. With AMs the SIB can be refined and implement the charging functionality using pointcuts and advice.

Figure 18 depicts an AM that implement the *pricing* feature by adding the charing code to the *SIB* (pointcut and advice).

We admit that this solution differs only in the fact that the charging code has moved to the broker. But (1) in this example the charging functionality can be logically assigned to the broker (which plays the role of the *producer*), and (2) the broker is an extension to an existent broker and can further refine it by, e.g., adding protocol code for tracing the client's transactions (Lines 11,12). We perceive that as an improvement over the AML solution.

AMs can be seen as aspects that inherit from class using mixin-based inheritance. The close connection between mixin and aspect may lead to deeper problems (e.g. regarding the life time dependency of aspects and mixins) which we currently cannot anticipate in their full depth.

However, the possibility to implement mixins, aspects, and aspectual mixins offers the programmer powerful techniques to implement various kinds of features. Each technique has its strengths and weaknesses and it is up to the programmer to select the adequate one. A downer is the fact that the overall approach may become to complex and the programmer is lost. In ongoing work we want to evaluate or ideas in real world case studies.

Multi Mixins. A further variant that comes into mind in order to improve mixins and mixin layers is to extend the capabilities of how to specify the join points where they are bound to. MMs are one approach to do that. They are related to the idea of AMs. The key difference is that MMs specify the set of parent mixins and methods using pointcut-like constructs (see Figure 19).

In a first variant we propose to specify the sets of parent mixins using simple pointcut-like constructs. Figure 20 depicts a multi mixin that implement the *accounting* feature. It refines the client and the broker (Line 1) and adds to both classes an account id (Line 3). Furthermore, it extends two methods (*charge*, *collectInfo*, Line 5-8) to apply the code that passes information to the *Accounting* class (Line 12). Since the two methods have different signatures the refined

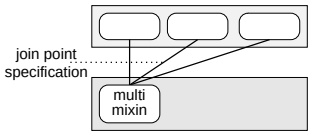

Fig. 19. Multi mixins: One mixin refines a set of parent mixins.

```
1  refines class Client &&
2      class StockInformationBroker {
3    int account;
4  public:
5    void charge(...) && collectInfo(...) {
6      StockInfoRequest &req =
7        *(StockInfoRequest*)argv(argc - 1);
8      Accounting::log(caller.account(), req.price());
9    }
10 };
```

Fig. 20. The accounting feature using multi mixins.

method body accesses the argument via reflective API (Line 6-7) and calls the *log* method (Line 8).

MMs offer a more advanced mechanism to express homogeneous and non-hierarchy-conforming refinements. The strength lies in the pointcut-like mechanism to specify the set of parent mixins and methods (Lines 1,5).

However, in current it is not clear if multi mixins introduce weaknesses in static typing, e.g. by accessing the caller (Line 7), the result, or the arguments (Line 6). Part of future work is how to exploit known ideas of AOP languages and ideas of [21] to guarantee static type checking.

The more complex the mechanism for specifying target methods and mixins become, the more multi mixins mutate to aspects and *classpects* [30]. However, aspects separate pointcuts and advices. classpects, further unify advices and methods. This is also true for multi mixins.

7 Related Work

7.1 Aspects, Features, and Collaborations

[23, 26, 19] feature an evaluation and discussion of common AOP and FOP approaches. They identify several weaknesses concerning the crosscutting modularity, the reuse of features/aspects, the support for dynamic composition, as well as missing module boundaries.

Our evaluation is based on their results but extends them by an explicit evaluation framework with focus on incremental software development. Especially, the direct connection between aspects, mixins, and mixin layer is novel and results in AMLs that exploit their synergetic potential.

Mezini et al. propose *Caesar* that combines aspects and collaboration to address the mentioned issues [26]. Aspects in Caesar rely on *aspect collaboration interfaces* that decouple an aspect's implementation from its binding. By defining a binding a programmer can adapt the aspect's implementation to the application context. This on-demand remodularization improves aspect reuse. Bindings are applied statically at object creation time or during the dynamic control flow. Different aspects can be composed via their collaboration interfaces. Collaborations are refined using pointcut-like constructs.

As with the Caesar approach, *Object Teams (OTs)* [12] and *Aspectual Collaborations (ACs)* [19] encapsulate aspects into modules with expected and provided interfaces. Their focus is similar to Caesar but with drawbacks regarding the aspect reuse (due to missing bidirectional interfaces).

Caesar, ACs, and OTs as well as AMLs have several similarities. All are based on collaborations which represent the basic building blocks and all integrate AOP concepts. The main advantage of AMLs is that they have AHEAD as architectural model. Although the others do not propose such model we perceive, if it is, *GenVoca* as their architectural model. AHEAD has several strengths compared to its predecessor GenVoca: It integrates all kinds of software artifacts and introduces an algebraic model for software structure. This opens the door to automatic algebra-based optimization and compositional reasoning [7].

However, Caesar, ACs, and OTs have a stronger focus on on-demand remodularization and dynamic composition which are not addressed in the AML approach.

Jiazzi is a component system for Java that allows to express components as collaborations [25]. A special linker is used to compose binary Jiazzi components. Although the author propose to use Jiazzi for AOP this is very complicated and has not the same power than common AOP approaches, e.g. AspectJ. Instead, AMLs support both AspectJ-like aspects and collaborations. As with the other mentioned component systems Jiazzi lacks an architectural model.

7.2 Aspects and Incremental Software Development

Colyer et al. propose the *principle of dependency alignment*: a set of guidelines for structuring features in modules and aspects with regard to program families [9]. They distinguish between orthogonal and weak-orthogonal features. But they do not distinguish between the structural properties and conceptual differences of aspects and features. The discussion of AMLs reveals that using aspects stand-alone has several weaknesses in implementing features. The integration in collaborations unifies and improves AOP and FOP.

Loughran et al. support the evolution of program families with *Framed Aspects* [24]. They combine the advantages of frames and AOP in order to serve unanticipated requirements. Framed Aspects are related to AMLs. Both allow to parameterize aspects at instantiation time but AML embed aspects into collaborations.

Kendall explores the connection between role modeling and AOP [14]. Aspects can implement roles, but, however, she does not consider the encapsulation

of aspects into collaborations. Thus, her approach has several shortcomings regarding cohesive role refinements.

7.3 AOP and Separation of Concerns

AMLs are related to *multi-dimensional separation of concerns (MDSC)* and *Hyper/J* that implements MDSC for Java [33]. This approach to software development is as general as that of AHEAD and AMLs [5]. It addresses all kinds of software artifacts. Hyper/J uses external rules to compose the different concerns. AMLs with its implementation level mechanisms based on roles and aspects, as well as their external composition using algebraic equations, is more flexible and leaves space for optimizations and compositional reasoning. However, the separation of features along different dimensions as proposed in MDSC has strengths compared to common collaboration-based designs, e.g. handling of optional features and feature interactions.

The *Law of Demeter for Concerns (LoDC)* states that concerns should only know other concerns that contribute to its own functionality [18]. Following this principle (1) eases the incremental evolution of software by adding concern by concern and (2) minimizes the number of feature interactions. AMLs follows LoDC and enables a clear encapsulation of concerns. The supported bounding mechanism scopes aspects in order to reduce unpredictable feature interactions.

Classpects combine capabilities of aspects and classes to unify the design of layered module systems [30]. They are related to AMLs, whereas classpects unify advice and methods (advice can be explicitly invoked), but do not support mixin-based refinements.

AspectJ-like languages can express mixins, too. Using static introductions, several classes (and methods) can be refined. In the face of heterogeneous crosscuts, for each target class a new aspect must be introduced. Otherwise, one aspect declares all introductions. The problem of the first approach is that it breaks feature cohesion. Moreover, the target classes are defined at development time. Therefore, an easy exchange of the target features is not possible (because class names change which is not the case with mixins). The second approach merges multiple refinement chains into one aspect. This may destroy the logical structure. Our approach can be seamlessly integrated into mixin layers and follows an architectural model. Moreover, it supports incremental software development using a novel bounding mechanism.

7.4 Aspects and Modular Reasoning

Several research efforts are made on the fact that aspects crosscut module boundaries. *Aspect-ware interfaces* [16] and *open modules* [1] support modular reasoning by encapsulating the interactions between two concerns. This reduces unpredictable aspect/feature interactions but also reduces the flexibility to implement unanticipated features. AMLs tackle the problem from another side: The AMLs bounding quantification exploits the order that is naturally introduced by different development stages. Furthermore, it is based on an algebraic model that

is more extensible [22]. However, in future work we aim on integrating ideas on modular reasoning into AMLs.

8 Conclusions

This article has contributed a discussion and evaluation of FOP and AOP with respect to the implementation of incremental designs and program families. We have identified the architectural connection between aspects and features. Both complement each other to implement complex software. We have introduced a set of criteria that poses as a basis for the evaluation and comparison of modularization techniques with focus on incremental software development. Whereas common AOP has weaknesses regarding heterogeneous crosscuts, feature cohesion, and implementing software incrementally, FOP has shortcomings in the crosscutting modularity. On the contrary, both contribute essential ideas and approved techniques to incremental software development at different levels of abstraction.

Since both paradigms have their strengths and weaknesses, we have proposed the symbiosis of both in order to exploit and combine their strengths. Based on a detailed evaluation, we have introduced the novel notion of aspectual mixin layers. AMLs enhance common mixin layers by integrating AOP techniques, i.e. by integrating aspects into mixin layers. This leads to an improvement of FOP regarding crosscutting modularity. Furthermore, it contributes an architectural model that integrates aspects and features as well as the novel AOP concepts of aspect refinement and bounding quantification.

A case study has revealed the benefits and limitations of AMLs. AMLs improve the crosscutting modularity of FOP but, unfortunately, they introduce known AOP weaknesses, e.g. no strict module boundaries. However, further improvements of aspect techniques will also advance AMLs.

A final discussion of alternative variants of AMLs revealed that there are numerous ways to integrate aspects and features. A deeper examination is part of further work.

9 Acknowledgments

The authors would like to thank Don Batory, Olaf Spinczyk, Aleksandra Tesanovic, Walter Cazzola, and Ingolf Geist for fruitful discussions on the ideas developed in this article.

This work is partially funded by the Metop Research Institute at Magdeburg, Germany.

References

1. J. Aldrich. Open Modules: Modular Reasoning about Advice. In *ECOOP*, 2005.
2. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *ASE'04 SEM Workshop*, volume 3437 of *LNCS*. 2005.

3. S. Apel et al. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*, 2005.
4. D. Batory, 2005. Personal correspondence.
5. D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. *ACM SIGSOFT*, 2003.
6. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1(4), 1992.
7. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
8. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *AOSD*, 2004.
9. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.
10. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
11. R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *GPCE 2002*, 2002.
12. S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NetObjectDays*, 2003.
13. D. Keck and P. Kuehn. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE TSE*, 1998.
14. E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *OOPSLA*, 1999.
15. G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, 1997.
16. G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *ICSE*, 2005.
17. T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *ADBIS*, 2005.
18. K. Lieberherr. Controlling the Complexity of Software Designs. In *ICSE*, 2004.
19. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.
20. J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature-Oriented Software Designs. In *ICFI*, 2005.
21. D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *GPCE*, 2004.
22. R. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *SPLAT*, 2005.
23. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.
24. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *AOSD ACP4IS Workshop*, 2004.
25. S. McDirmid and W. Hsieh. Aspect-Oriented Programming in Jiazzi. In *AOSD*, 2003.
26. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
27. K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *ECOOP*, 2005.
28. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE TSE*, SE-5(2), 1979.
29. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, 1997.

30. H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *ICSE*, 2005.
31. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.
32. O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's Journal*, (5), 2005.
33. P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.
34. P. Zave. Feature Interactions and Formal Specifications in Telecommunications. *IEEE Computer*, XXVI(8), 1993.