Aspectual Mixin Layers: Aspects and Features in Concert

Sven Apel
University of Magdeburg
P.O. Box 4120
39016, Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de

Thomas Leich
University of Magdeburg
P.O. Box 4120
39016, Magdeburg, Germany
leich@iti.cs.uni-magdeburg.de

Gunter Saake
University of Magdeburg
P.O. Box 4120
39016, Magdeburg, Germany
saake@iti.cs.uni-magdeburg.de

ABSTRACT

Feature-Oriented Programming (*FOP*) decomposes complex software into features. Features are main abstractions in design and implementation. They reflect user requirements and incrementally refine one another. Although, features crosscut object-oriented architectures they fail to express all kinds of crosscutting concerns. This weakness is exactly the strength of aspects, the main abstraction mechanism of *Aspect-Oriented Programming* (*AOP*). In this article we contribute a systematic evaluation and comparison of both paradigms, AOP and FOP, with focus on incremental software development. It reveals that aspects and features are not competing concepts. In fact AOP has several strengths to improve FOP in order to implement crosscutting features. Symmetrically, the development model of FOP can aid AOP in implementing incremental designs. Consequently, we propose the architectural integration of aspects and features in order to profit from both paradigms. We introduce *aspectual mixin layers (AMLs)*, an implementation approach that realizes this symbiosis. A subsequent evaluation and a case study reveal that AMLs improve the crosscutting modularity of features as well as aspects become well integrated into incremental development style.

Categories and Subject Descriptors: D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*

General Terms: Design, Languages

Keywords: Feature-Oriented Programing, Aspect-Oriented Programming, Component Techniques, Collaborations

1. INTRODUCTION

Program families [30] and incremental software development [35] have a long tradition and are still subjects of current research. A main objective of research in this field is to simplify the maintenance, reuse, customization, and evolution of software. Two programming paradigms heavily discussed in this context 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 features are mapped one-to-one to modular implementation units (*feature modules*). Since it has emerged that traditional abstractions as classes and objects are too small units of modularity, features contain a set of classes that contribute to the features in *collaborations* [7, 32, 28, 20]. Therefore, refinement of features means refinement of their structural elements.

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. product lines. Although the initial focus does not lie on incremental software development several research efforts go into this direction [23, 28, 10, 24, 20], however, with numerous problems that are discussed here.

Relationship of aspects and features. In this paper we explore the relationship of AOP and FOP an therewith the connection between aspects and features.[1] We do not perceive them as competing approaches but rather as approaches 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. Since features encapsulate collaborations and refine one another, the underlying object-oriented architecture becomes organized at a higher level. It is decomposed along these collaborations. Despite these advantages, FOP has drawbacks regarding (1) the crosscutting modularity, in particular the ability to localize, separate, and modularize certain kinds of crosscutting concerns as well as (2) the ability to seamlessly integrate structural independent features [28, 20]. Both are highly related since an integration of independent features results usually in a crosscutting interconnection of the corresponding structural elements. This is where AOP comes into play.

Aspects modularize concerns that otherwise crosscut other concerns. But they are not adequate to implement all kinds of features. In many cases aspects cannot implement fea-

[1] In the remaining paper we use AOP/FOP and aspects/features synonymously, despite the fact that the former are programming paradigms and the latter their main concepts.

ICSE'06, May 20–28, 2006, Shanghai, China.

tures stand-alone [22]. Other aspects and additional classes are needed. This is because features are mostly implemented by collaborations, but common AOP techniques are not able to express and encapsulate collaborations. A further drawback of current AOP approaches is that aspects 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 [23, 25].

However, we see aspects at the level of classes whereas features organize the overall 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 structure an *aspect-oriented architecture*. Technically this means that aspects are integrated into collaborations and work with classes and other aspects in *concert* to implement features of a software.

The close integration of aspects and features holds several advantages to FOP *and* to AOP: (1) Introducing aspects into feature modules improves their crosscutting modularity as well as the ability to integrate structural independent features. (2) Our approach supports the programmer to implement incremental designs using aspects. Due to their integration into a stack of feature modules we were able to bound aspects to certain layers. In short, this bounding capability decreases unpredictable aspect behavior in the face of adding subsequently unanticipated features.

Our view on the relationship of aspects and features differs from previous work: *Caesar* [28, 27] and related approaches aim at improving AOP by adopting component techniques. These approaches intermix structural elements of aspects and features, e.g. pointcuts and collaborations. Our view is more general and explores the architectural relationship of aspects and features. We discovered that there is a natural connection between both. By applying our ideas to AHEAD [7] – an advanced architectural model for FOP – we place them on a sound algebraic foundation (see Sec. 6).

However, our ideas are inspired by this previous work and extend our investigations in FeatureC++ a feature-oriented extension to C++ [3]. In this paper we address the general issues that arise from the individual properties of aspects and features as well as their integration. Our results are independent from a specific language and can be seen as an architectural approach for integrating AOP and FOP.

In this paper we make the following contributions:

- Criteria for evaluating modularization techniques with respect to incremental software development.
- An evaluation of FOP and AOP with regard to modularity, expressiveness, and reusability.
- The integration of aspects and features at architectural level that is based on a symbiosis of AOP and FOP.
- Aspectual mixin layers, an implementation approach that integrates aspects and feature modules in order to exploit their strengths and their synergetic potential.
- An evaluation, a case study, and a discussion of AMLs focusing on their contribution to AOP and FOP, in the broader context of incremental software development.

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++* [33].

2.1 Feature-Oriented Programming

FOP studies the modularity of *features* in product lines [7]. The idea of FOP is to build software (individual programs) by composing features that are first-class entities in design and implementation. Features refine other features incrementally. Hence, the term *refinement* refers to a feature refining others. This *step-wise refinement* leads to conceptually layered software designs.

Mixin layers are one appropriate technique to implement features [32, 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. 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 boxes represent the mixins. Mixins that belong to and constitute together a complete class are called a *refinement chain*.

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

FeatureC++[2] is an extension to C++ that supports FOP. FeatureC++'s mixin layers are represented by file system directories. Thus, they have no textual representation at code level. Those mixins found inside a directory are assigned to be members of the enclosing mixin layer.

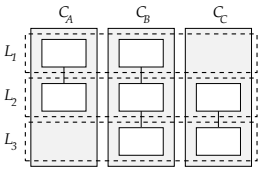

Figure 1: Stack of mixin layers.

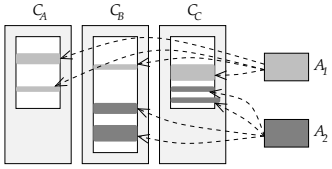

Figure 2: Two aspects extend three classes.

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 (Line 6) or extend methods of their parent classes (Lines 7-9). To access the extended method the *super* keyword is used (Line 8). A more detailed explanation of FeatureC++, its capabilities, and its implementation is given elsewhere [3].

2.2 Aspect-Oriented Programming

AOP aims at separating and modularizing crosscutting concerns [15]. Using object-oriented mechanisms the implementation of crosscutting concerns results in tangled and scattered code [15, 11]. The idea behind AOP is to implement crosscutting concerns as *aspects* whereas the core features are implemented as components. Using *pointcuts*

[2]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 };
```

Figure 3: Refining a buffer.

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

Figure 4: A logging aspect in AspectC++.

and *advice*, an aspect weaver glues 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 (dashed arrows).

AspectC++[3] is an C++ language extension for AOP. Figure 4 lists an aspect that implements a logging concern that otherwise would crosscut the basic buffer functionality. The pointcut *log* (Line 2) specifies the set of points in the structure and execution of the base program (*join points*) that are supposed to be extended by the logging concern. An corresponding advice (Lines 3-5) executes the logging code.

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 FOP and AOP.

3.1 Evaluation Criteria

The following considerations are based on our experience in building product lines [2, 18] and on previous work [22, 28, 20]. We are aware that this set of criteria may not be complete, but we consider only those criteria that are crucial to step-wise refinement and in which FOP and AOP differ significantly.

Homogeneous vs. heterogeneous crosscuts. Crosscutting concerns differ in their structure [9]: *Homogeneous crosscuts* apply the same code at different join points whereas *heterogeneous crosscuts* apply different code. A different point of view is that homogeneous crosscuts interact with the base program in a *one-to-n* pattern and heterogeneous crosscuts in a *m-to-n* pattern. If a modularization techniques does not provide explicit mechanisms for encapsulating homogeneous crosscuts, the programmer has to refine each join point with separate structural elements that contain redundant code, e.g. refining a set of methods with a set of new methods that all contain the same code.

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

Static vs. dynamic crosscutting. Crosscutting concerns affect a program in two ways: *Static crosscutting* affects the static structure, e.g. by adding members and classes. *Dynamic crosscutting* affects the dynamic behavior, e.g. method interceptions. Whereas static crosscutting extends a given program structure, dynamic crosscutting indicates when to execute an extension at runtime. Both types are essential because new features usually extend structure *and* behavior.

Structural dependency. A straightforward way to implement a new feature is to express it in terms of the structure of the refined features. That means, new features *superimpose* base features in order to define a set of refinements to the existing structural elements [8, 7]. Such features are called *hierarchy-conforming* because they depend on the given program structure. Although this structure-preserving super-imposition is very useful for comprehensibility, there are certain situations where it is beneficial to alter the structure and to raise the abstraction level [28, 20, 14]. Such features are called *non-hierarchy-conforming*. Suppose a network software that is refined by an application protocol. The interactions at application level can be much easier expressed in terms of producer (server), consumer (client), and product (delivered data) than using the basic network abstractions such as sockets and streams. Implementing such features is strongly related to crosscutting phenomena because with object-oriented techniques several structural elements at different locations (across the network protocol) are refined by a new structure (producer-consumer pattern).

Integrating features that are structural independent results in similar problems, e.g. connecting a component for displaying hierarchical graph structures consisting of nodes and connections with a component abstracting the network structure of a distributed business application. Since there are no structural counterparts between those two components it is very complicated to achieve a clean mapping. Thus, such integration results in crosscutting adaptation code [27]. Whereas the hierarchy-conforming features are naturally supported by most language paradigms, e.g. inheritance, it is more difficult to implement non-hierarchy-conforming features. Both should be explicitly supported by modern modularization techniques.

Feature cohesion. Cohesion is the property of a feature to encapsulate all implementation units that contribute to the feature in one module [22]. This eases the maintainability, clearness, and configurability of software. The one-to-one mapping of requirements to feature modules is an idealized goal [11]. Feature cohesion is the basis for aggregating features to form compound features. Such *hierarchical aggregation* 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 features from scratch.

3.2 Evaluation

Using our criteria, we evaluate FOP and AOP step-by-step. In order to do not complicate the evaluation, we discuss hybrid approaches, e.g. Caesar, in Section 6.

Homogeneous vs. heterogeneous crosscuts. FOP deals with implementing heterogeneous crosscuts. Usually, a programmer refines different classes and methods with different

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. This results in redundant programming effort and duplicated code. This weakness in modularizing homogeneous crosscuts is the strength of AOP.

Using aspects 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 is not well suited to implement heterogeneous crosscuts compared to implementing features as collaborations. Aspects act not at this level of abstraction [28, 20]. They are not intended to organize a collaboration of classes and its subsequent refinement.

However, it is possible to bundle a set of static introductions or pairs of pointcuts and advice in one aspect as proposed in [31], but such aspects do not reflect the structure of the refined feature [28]. That means that the logical structure reflecting the domain knowledge is broken. But this structure – that the programmer had in mind during the initial design – is important for the comprehensibility of the overall architecture [8]. However, sometimes there are situations where we want to raise the abstraction level and change the structure, as we will explain.

Suppose a producer-consumer-protocol is refined by functionality that allows for exchanging products. This refinement logically extends the producer *and* the consumer. Encapsulating these pieces into one single aspect moves the code responsible for communication from the producer and consumer away to a third abstraction (aspect). The problem of this separation is that this code is an integral part of the producer and consumer concept. Several studies confirm that using instead super-imposition of the collaborating objects perform better with respect to maintainability and comprehensibility [7, 28, 32, 20, 14, 8].

Another way to implement a heterogeneous crosscut using AOP is to define one refining aspect per join point, instead of refining these different join points with one single aspect. This solution is similar to the FOP approach but uses pointcuts and advice – instead of using inheritance – to refine the parent's structural elements. However, even in the case that there are no additional classes needed this would break feature cohesion because the several aspects that contribute to the feature are not encapsulated in an enclosing feature module. Integrating these aspects in a package does not rectify this since packages cannot be composed and aggregated.

Static vs. dynamic crosscutting. Both, FOP and AOP support static crosscutting, in particular adding methods and attributes. FOP has a more general mechanism for introducing new structural elements since it support also the introduction of new classes. Since in AOP there is no concept of feature module, one cannot specify aspects that introduce new classes. However, an aspect may contain inner classes but as already mentioned there are no mechanisms to compose different aspects and their their inner classes simultaneously.

Furthermore, both paradigms support dynamic crosscutting. Whereas FOP supports only simple method interceptions (that correspond to *execution* pointcuts), AOP can express more advanced dynamic crosscutting, e.g. using *cflow*.

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 super-imposition hinders the raising and altering of the abstraction level. 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. For the same reason the integration of structural independent features leads to complex crosscutting workarounds [27].

AOP allows encapsulating a refinement into aspects. Aspects are able to refine a base program at multiple join points. Although, the quantification of aspects is expressed in terms of syntactical properties of the base program, the aspects themselves do not have to be aligned with the inherent structure of the base program [28]. Thus, the programmer can introduce new abstractions (aspects and classes in collaboration) that build up on present structures but introduce new concepts. This allows for connecting features that differ in their structure [28, 20, 14].

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.

Using AOP, a programmer expresses new features by introducing aspects and classes. In many cases features cannot be expressed using one single aspect, especially not in large evolved programs [22, 28]. 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. A significant body of research confirms that classes or aspects standalone are too small units of modularity and are not suitable for implementing features [7, 27, 20, 34, 6].

Indeed, aspects can be encapsulated in packages or may contain nested classes. Anyways, there is no mechanism for refining, composing, and aggregating such constructs. Besides this lack of encapsulation also the complicated and hard-to-understand precedence rules for ordering the appliance of aspects hinder an easy and consistent composition mechanism to form compound features [23].

A further benefit of FOP which is exclusive to AHEAD is that feature compositions can be described using algebraic equations. An algebraic model poses as a basis and supports automatic composition and optimization as well as compositional reasoning [7].

3.3 Summary

Table 1 summarizes our evaluation results. As we already explained, all these evaluation criteria are crucial to incremental software development. Choosing one modularization technique, FOP or AOP, leads to the known 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.

	homogeneous crosscuts	heterogeneous crosscuts	static crosscutting	dynamic crosscutting	structural dependency	feature cohesion
FOP	○	●	●	◑	○	●
AOP	●	◑	◑	●	◑	◑

● good support
◑ limited support
○ weak/no support

Table 1: Evaluation of FOP and AOP.

4. ON THE SYMBIOSIS OF AOP AND FOP

This section presents an approach that aims at solving the problems discussed. We propose the integration of aspects and features at architectural level as well as a technique to implement the envisioned kind of feature modules. Furthermore, we introduce the notion of *aspect refinement* and a specific *aspect bounding mechanism* that follow logically from the symbiosis of AOP and FOP. Finally, we evaluate our approach using our criteria.

4.1 Integrating Aspects and Features

When designing and implementing software systems in a feature-oriented way, a programmer starts usually by modeling and abstracting real world entities in terms of classes and objects and their manifold collaborations. The result is an object-oriented architecture. Feature-oriented approaches further organize this architecture using features. Features crosscut the architecture horizontally, i.e. they encapsulate those class fragments (and their collaborations) that contribute to the feature. Moreover, subsequent features refine existing features, i.e. their structural elements. Therefore, features are a mechanism that organize architectures at a higher level of abstraction than classes (see Fig. 5).

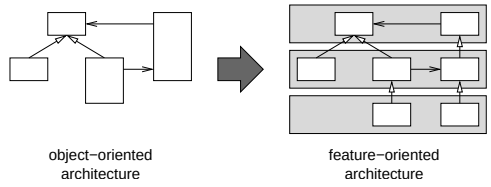

Figure 5: FOP Decomposition.

Our evaluation pointed us to the fact that in certain situations features cannot be appropriately expressed, i.e. inelegant workarounds, code redundancies as well as scattered and tangled code polluted the implementation. Mostly, these situations were related to crosscutting phenomena. We argue that the revealed shortcomings of common FOP approaches are directly responsible for this tension.

To overcome this tension we propose to utilize AOP mechanisms since they are powerful mechanisms to cope with crosscutting concerns. As our evaluation demonstrates, simply using AOP and creating an aspect-oriented architecture is not appropriate because of the mentioned shortcomings (cf. Sec. 3). Therefore, we propose to decompose such architectures using the known mechanisms of FOP: *An aspect-oriented architecture serves as basis whereas features organize the architecture at a higher level of abstraction.* Thus, a feature encapsulates a collaboration of classes *and* aspects (see Fig. 6). In doing that, we have well encapsulated large-scale feature modules that incrementally refine one another as well as powerful mechanisms to deal with crosscutting phenomena. According to this view, aspects are at the level of classes and contribute in collaboration with other artifacts to the implementation of a feature. Feature organize the overall architecture in order to reflect a structure that is of interest for the stakeholders.

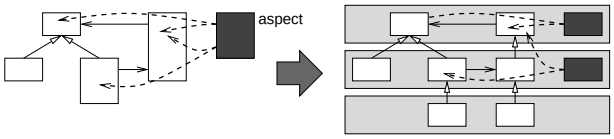

Figure 6: Decomposing an aspect-oriented architecture (dashed arrows mark aspect weaving).

4.2 Aspectual Mixin Layers

In order to provide language mechanisms to realize the architectural integration of aspects and features, we introduce AMLs. AMLs extend the notion of mixin layers [32] by encapsulating besides mixins also aspects (see Fig. 7). In fact an AML encapsulates roles of classes *and* aspects that contribute to a collaboration, i.e. that implements a feature. By applying an AML to a set of features the programmer can refine these features in two ways: (1) by using common mixin-composition[4] or (2) by using aspect-oriented mechanisms, e.g. pointcuts and advice. Probably the most

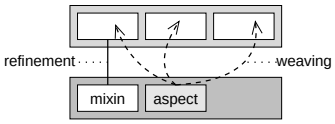

Figure 7: Aspectual mixin layers.

important contribution of AMLs is that programmers may choose the adequate technique – mixins or aspects – that fits a given problem best. Moreover, they can apply a collaboration of both and decide to what extent one technique is used.

However, two crucial questions arise: (1) When to use what mechanism without interspersing both and (2) how to integrate them a technical level.

Regarding the first question our analysis gives the answer: The programmer uses mixins and their collaborations in those situations in that they perform well, i.e. in implementing static, heterogeneous, and hierarchy-conforming crosscutting – in short, the common way of FOP. In contrast to mixins, the programmer uses aspects to implement particular kinds of crosscutting concerns, i.e. in order to modularize homogeneous crosscuts, features that depend highly on the runtime control flow, as well as non-hierarchy-conform crosscutting as in the case of connecting structural independent features. Having these rules a programmer has a guideline to avoid intermixing both refinement mechanisms inconsistently.

Regarding the second question there are two options: The first option composes the mixins first and applies subsequently the aspects. The second composes mixins and aspects layer by layer. Because of technical reasons we chose in FeatureC++ the first option. The second option is worth to be considered but out of scope of this paper.

[4]The programmer may use standard object-oriented mechanisms, too, e.g. delegation, inheritance, etc.

Example. Figure 8 shows a stack of mixin layers that implement buffer functionality, in particular a basic buffer with iterator, an allocator, synchronization and logging support. Whereas the first three features are implemented using mixins only, 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 are refined. This refinement modularizes a homogeneous crosscutting concern and may access the dynamic context □

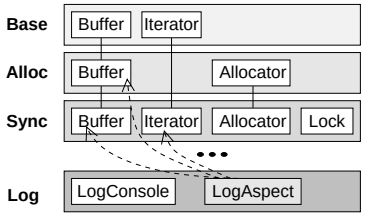

Figure 8: An AML for logging.

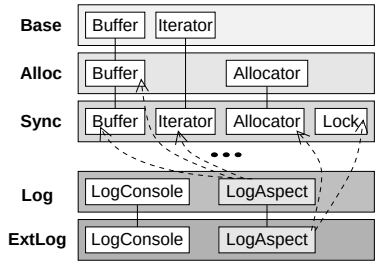

Figure 9: Refining an AML.

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

Figure 10: Refining a logging aspect.

4.3 Aspect Refinement

The introduction of AOP concepts to FOP leads us to the notion of *aspect refinement*. Since aspects are encapsulated in 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.

By refining a pointcut a programmer may subsequently alter, constrain, or extend a set of join points. Hanenberg et al. propose several patterns that could benefit of such refinement, e.g. *composite pointcut*, *pointcut method* [12]. Think of a logging aspect that matches certain points in a given program. Introducing new classes makes it necessary to modify the logging aspect to match join points related to these classes. Following the idea of incremental software development it is recommended to refine the logging aspect subsequently, instead of changing the aspect.

Analogously to pointcuts, it is useful to refine advice, too. Since they encapsulate aspect functionality, they should be subject of subsequent refinement and reuse. Unfortunately, in most aspect languages advice are unnamed entities. Therefore, they cannot be referred to. We recently proposed *named advice* and *advice refinement* that address this issue [4, 5]. However, a further discussion is out of scope.

Example. Figure 9 shows a feature *ExtLog* that refines the feature *Log*. The refinement is implemented as AML. It refines the logging aspect of *Log* by refining its pointcut. In doing that, the set of intercepted methods is extended to *Allocator* and *Lock*. Besides this, the logging console (implemented as a mixin) is refined. Figure 10 lists the code of this refinement to the logging aspect. It 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). The refinement is expressed using *refines*. Similar to methods, *super* is used to refer to the parent pointcuts (Line 4). □

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 aspects. Instead, aspect refinement results in two aspect fragments that are connected via inheritance. Only the final compound aspect is woven to the target program. Applied to our example, we have only one logging aspect.

4.4 Bounding Aspect Quantification

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

A problem of current AOP languages is that that quantification of aspects is *unbounded*. That means aspects can potentially affect all other entities – also entities that are completely unrelated or have been applied in after the introduction of the aspect itself. Currently, there are no adequate mechanisms to scope or bound aspects. In context of incremental software development this means that an aspect may affect subsequent integrated features although the aspect was implemented without being aware of these features. This can lead to unpredictable effects and errors, e.g. an aspect unintentionally interacts with a subsequent features [23, 25].

Lopez-Herrejon and Batory propose an alternative aspect composition mechanism [23]. They argue that with regard to software evolution, features should only affect features of previous development stages. Each incremental refinement to a program is assigned to a development stage. Mapping this to aspects means that aspects should only affect elements of development stages that were already present at the implementation time of these aspects. This corresponds to a *bounded aspect quantification*.

What turns out of our proposed integration of aspects and features in the broader context of an incremental development model is that we are now able to implement such or alternative bounding mechanisms. Since we know which aspects are part of which features and what the parent and child features are, we can infer which aspect is permitted to affect which feature. We implemented a first version of bounded aspect quantification in FeatureC++ using pointcut restructuring, i.e. translating pointcut expressions in that way that they do affect only the 'right' features. However, a deeper discussion is out of scope of this paper and we refer to [4, 5].

4.5 Evaluation of Aspectual Mixin Layers

Taking the ideas of aspect refinement and a bounded aspect 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, we recommend to combine mixins and aspects in one feature module to profit from their collaboration.

Static and dynamic crosscutting. The integration of FOP and AOP concepts allows us 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 propose to use aspects to implement homogeneous crosscuts and mixins to implement heterogeneous crosscuts, which depend on the structure of the parent feature.

By using aspects, a programmer can implement features depending on the runtime control flow, the current state, and the dynamic context. As with static crosscutting method extensions can be implemented with aspects (using *execution*) 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. Aspects inside AMLs can be used to alter the level of abstraction. They can connect features with differing structural assembling. For example, one can use an aspect to connect our former example of a network software and a producer-consumer application protocol. Using common object-oriented mechanisms this mapping could only be established via wrappers and hash-maps to apply the refinements and maintain the connections [28, 20, 14]. This is a non-trivial workaround and results in crosscutting code.

What is important is that a structural differing feature consists of such binding and the new abstractions that usually are implemented as classes. Thus, AML improve the ability to synthesize programs by composing structural different features.

Feature cohesion. Since we encapsulate aspects in feature modules, we achieve a high degree of feature cohesion. 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 feature module. What is novel is that aspects can be refined (composed) to form new aspects and their quantification is bound to the parent features. This allows to reuse existent aspect functionality and avoids unexpected interactions with subsequent features.

Features that contain aspects can be composed using declarative descriptions. This is an improvement of AOP with respect to incremental software development. In summary, AMLs perform better than FOP and AOP standalone because they combine the advantages of both. However, it is up to the programmer to choose the right techniques to implement a given feature.

5. CASE STUDY

In order to survey our approach, we implement a stock information broker [28] using mixin layers and AML.

5.1 A Stock Information Broker

A stock information broker deals with information about the stock market. The main abstraction is the *StockInformationBroker (SIB)* that allows to lookup for information about stocks (see Fig. 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 a *StockInfo (SI)* object that contains the stock quotes.

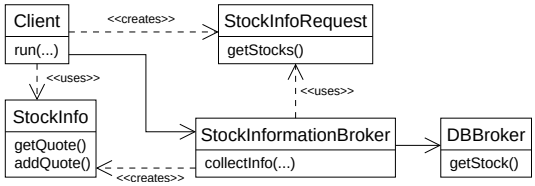

Figure 11: Stock Information Broker.

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.

```
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 class Client {
12   StockInformationBroker &m_broker;
13 public:
14   void run(string *stocks, unsigned int num) {
15     StockInfoRequest sir(stocks, num);
16     StockInfo &info = m_broker.collectInfo(sir);
17     /* ... */
18   }
19 };
```

Figure 12: The basic stock information broker.

5.2 Implementing Refinements

In the next step we want to add two refinements: (1) a *pricing* feature that charges the client's account depending on the received stock quotes and (2) an *accounting* feature that monitors the flow of money between client and broker. We survey their implementation using FOP and AMLs.

FOP solution. Figure 13 depicts the *pricing* feature implemented using FOP. *Client* is refined by an account management (Lines 15-24), *SIR* is refined by a price calculation (Lines 1-6), and *SIB* charges the account when passing information to the client (Lines 9-13).

The *accounting* feature is depicted in Figure 14. An *Accounting* class (Line 19) stores and manages information

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

Figure 13: The pricing feature using FOP.

```
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 refines class Client {
12   int account;
13 public:
14   void Client::charge(StockInfoRequest &req) {
15     super::charge(req);
16     Accounting::log(account, req.price());
17   }
18 };
19 class Accounting {
20   void log(int, float) { /*...*/ }
21 };
```

Figure 14: The accounting feature using FOP.

about money transfers between client and broker. *Client* and *SIB* are extended by account ids (Lines 2,12). 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 7,16).

There are several problems to this approach: (1) The *pricing* feature is expressed in terms of the structure of the *base* feature. It would be better to describe the *pricing* feature using abstractions as product, producer, and customer, but that imposes a different structure that is hard to express in FOP. (2) The interface of *collectInfo* was extended. Therefore, the *Client* must inelegantly override the method *run* in order to pass a reference of itself to the *SIB*. (3) The charging of clients is hard-coded in the broker and cannot be altered according to the control flow, e.g. charge only those clients that passed a authentication procedure. (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 *log* is redundant in client and broker.

```
1  aspect Charging {
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  };
10 refines class Client {
11   float m_balance;
12 public:
13   float balance();
14   void charge(StockInfoRequest &req) { /* ... */ }
15 };
```

Figure 15: The pricing feature using AMLs.

AML 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 subsequently refined.

The *Charging* aspect intercepts calls to the method *collectInfo* (Lines 2-4) and charges the calling client depending on its request (Lines 5-8). 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, e.g. using *cflow* one can determine if this client has successfully passed an authentication procedure. This makes it possible 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*). Although, there is no direct representation of the producer role, AMLs improve the capabilities to alter the abstraction level.

The *accounting* feature is implemented using an aspect (see Figure 16). The account ids are added using static introductions (Lines 2-3). A pointcut specifies the target methods (Lines 4-5) and an advice adds calls to the *Accounting* class (Line 9). This solution implements the homogeneous *accounting* feature in an elegant way. It encapsulates an aspect and a class in a cohesive module.

```
1  aspect AccountingAspect {
2    pointcut id() = "Client" || "StockInformationBroker";
3    advice id() : int account;
4    pointcut transfers() =
5      call("%%::collect%(...)" || "%%::charge(...)");
6    advice transfers() : after() {
7      StockInfoRequest &req =
8        *(StockInfoRequest*)tjp->arg(JoinPoint::ARGS-1);
9      Accounting::log(tjp->that().account, req.price());
10   }
11 };
12 class Accounting {
13   void log(int, float) { /*...*/ }
14 };
```

Figure 16: The accounting feature using AMLs.

6. RELATED WORK

Aspects, features, and collaborations. Some studies evaluate and discuss AOP and FOP approaches [22, 28, 20]. 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 and features is novel and results in AMLs that exploit their synergetic potential.

Several approaches aim at collaboration-based designs and their symbiosis with AOP mechanisms: *Caesar* [27, 28], *Adaptive Plug-and-Play Components* [26], *Aspectual Components* [19], *Pluggable Composite Adapters* [29], *Aspectual Collaborations* [20], and *Object Teams* [14]. Since these approaches were highly influenced by one another, we compare our approach to their general concepts. We choose Caesar as a representative because it unifies the most essential ideas and it has grown to the most matured approach.

Caesar supports componentization of aspects by encapsulating virtual classes as well as pointcuts and advice in collaborations, so called *aspect components*. Aspect components can be composed via their collaboration interfaces and mixin composition. Besides this, they can be refined using pointcuts in order to implement crosscutting integration. Due to its embedding in classes (*family classes*), collaborations of virtual classes can be used polymorphically

Caesar and AMLs have several similarities. They employ collaborations to form the basic building blocks. Moreover, both integrate AOP concepts, but in different ways. A main advantage of AMLs is that they have AHEAD as an architectural model; The others make no statement about such a model. Hence, AMLs can revert to several advantages of AHEAD: beside classes and aspects also other kinds of software artifacts may be included in a feature; features are described and composed via algebraic equations and checked against domain-specific design rules. This opens the door to automatic algebra-based optimization and compositional reasoning. This is hard to implement in Caesar because Caesar´s aspect components are explicit at language level and their composition is done within source code.

Caesar does not provides mechanisms to refine elements specific to aspects, i.e. pointcuts and advice. But even such aspect refinement is a key to unify AOP and incremental software development. Furthermore, Caesar chooses a different approach to bound and control aspects: Aspects can be explicitly deployed to bound them to a certain scope. Instead, our bounding mechanism operates behind the scenes by exploiting the natural order of the incremental design, i.e. the order of the development stages.

In contrast to AMLs, Caesar provides sophisticated mechanisms for adapting and binding collaborations to a particular application context (*on-demand remodularization*).

Other related work. 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 [10]. They focus on features in general and do not distinguish between the structural properties and conceptual differences of aspects and components.

Several recent approaches enhance aspects with genericity, e.g. *Sally* [13], *Generic Advice* [21], *LogicAJ* [17], *Framed Aspects* [24]. This improves reusability of aspects in different contexts. AMLs and aspect refinement provide an alternative way to customize aspects, i.e. by composing the required refinements to a compound aspect. However, ideas on generic aspects could be combined with our approach.

Hanenberg and Unland discuss the benefits of inheritance in the context of AOP [12]. They argue that aspect inheritance improves aspect reuse and propose several patterns that use aspect inheritance, i.e. *pointcut method*, *composite pointcut*, *chained advice*, and *template advice*. The flexibility aspect refinement can enhance these patterns by simplifying the composition of aspects using mixin techniques.

Aspect-ware interfaces [16] and *open modules* [1] aim at modular reasoning by encapsulating the interactions between two concerns. This reduces unpredictable aspect interactions but also confines the flexibility to implement unanticipated features. AMLs tackle the problem from another side: Bounded quantification exploits the order that is naturally imposed by the incrementally evolved architecture.

7. CONCLUSIONS

This paper contributed a discussion and evaluation of FOP and AOP with respect to incremental software development. We identified an architectural connection between aspects and features. Both complement each other to implement complex software. We 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. However, 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 proposed the architectural symbiosis of both paradigms in order to exploit and combine their strengths. We argued that features are the units of (de)composing aspect-oriented architectures. Hence, we see aspects at the level of classes to implement concerns that would otherwise crosscut other concerns. Feature organize the overall architecture at a higher level to structure the code into cohesive feature modules that reflect the users requirements.

In order to realize such architectural integration we introduced AMLs. AMLs contribute several novel ideas to FOP and AOP. AMLs tackle the FOP problems by introducing aspects into mixin layers. Thus, the programmer has powerful mechanisms to cope with crosscutting concerns, e.g. as in the case of homogeneous crosscuts or structural independent features. This enables the programmer to choose the adequate technique for a given problem. A main advantage is that the programmer does not use aspects stand-alone but encapsulated in AMLs with mixins in concert.

Furthermore, AML contribute a unification of AOP and incremental software development: they integrate aspects into an incremental development model. This allows to refine aspects similar to classes. Aspect refinement is a unification of aspects and classes with regard to mixin composition and incremental software development. It enables the programmer to reuse existent aspect code as well as evolve aspects over time. Refining aspects leads to the observation

that it would be useful to refine all kinds structural elements, which follows the *principle of uniformity* [7]

A further benefit of the architectural integration is that we are able to bound the quantification of aspects depending on their development stage. Aspects can be bound to those features that are known when implementing the aspects. This is supposed to reduce unexpected behavior and interactions as well as increases aspect reuse [23, 25]. This is an important contribution to the unification of AOP incremental software development.

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.

8. 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] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounding Quantification in Incremental Designs. In *APSEC*, 2005.

[5] S. Apel, T. Leich, and G. Saake. Mixin-Based Aspect Inheritance. Technical Report 10, Department of Computer Science, University of Magdeburg, Germany, 2005.

[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] J. Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5), 1999.

[9] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *AOSD*, 2004.

[10] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.

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

[12] S. Hanenberg and A. Schmidmeier. Idioms for Building Software Frameworks in AspectJ. In *AOSD ACP4IS Workshop*, 2003.

[13] S. Hanenberg and R. Unland. Parametric Introductions. In *AOSD*, 2003.

[14] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NetObjectDays*, 2002.

[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] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable Pattern Implementations Need Generic Aspects. In *ECOOP RAM-SE Workshop*, 2004.

[18] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *ADBIS*, 2005.

[19] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical report, College of Computer Science, Northeastern University, 1999.

[20] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.

[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, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.

[23] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *ACM SIGPLAN PEPM Workshop*, 2006.

[24] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR*, 2004.

[25] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *AOSD*, 2005.

[26] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *OOPSLA*, 1998.

[27] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *AOSD*, 2003.

[28] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT FSE-12*, 2004.

[29] M. Mezini, L. Seiter, and K. Lieberherr. Component Integration with Pluggable Composite Adapters. *Software Architectures and Component Technology: The State of the Art in Research and Practice*, 2000.

[30] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE TSE*, SE-5(2), 1979.

[31] E. Pulvermüller, A. Speck, and A. Rashid. Implementing Collaboration-Based Designs Using Aspect-Oriented Programming. In *TOOLS*, 2000.

[32] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.

[33] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's Journal*, 2005(5), 2005.

[34] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.

[35] N. Wirth. Program Development by Stepwise Refinement. *CACM*, 14(4), 1971.