Aspect Refinement and Bounded Quantification in Incremental Designs

Sven Apel, Thomas Leich, and Gunter Saake
Department of Computer Science
Otto-von-Guericke-University Magdeburg
{*apel, leich, saake*}*@iti.cs.uni-magdeburg.de*

Abstract

This article investigates aspects in the context of the incremental software development, i.e. software product lines. Specifically, we propose the integration of aspects into AHEAD*, an architectural model for feature-based product line development. We introduce the notion of* aspect refinement *based on Aspectual Mixin Layers, a novel technique for implementing features. Aspect refinement enables a programmer to evolve aspects over several product line development stages. This is novel since common AOP approaches do not have such an architectural model. We realize the idea of aspect refinement by introducing mixin-based inheritance to aspects. Furthermore, we propose a* bounded aspect quantification *that reduces the complexity and unpredictability of aspects in incremental software development. Our novel bounding mechanism exploits the natural order of the layered architecture introduced by the concept of aspect refinement. Aspect refinement and bounded aspect quantification improve the incremental development of product lines using AOP techniques.*

1 Introduction

Software product lines are subject of ongoing research and will gain momentum in future. Research in this field tries to move software development to the new quality of industrial production. *AHEAD* is an architectural model to implement product lines [3]. The idea of AHEAD is to decompose programs into basic *features* and to compose *stacks of features* to derive concrete programs. Products added to a product line are supposed to reuse features of existing ones and further add new features. This is also called *step-wise refinement*. The steps correspond to the development stages of the evolving product line. *AHEAD* proposes large-scale compositional programming: It generalizes the concept for features and feature refinements. Features consist not only of code but of several types of artifacts, e.g., makefiles, UML-diagrams, documentation. Each artifact inside a feature can refine corresponding artifacts of previous features.

This article explores the relationship of AHEAD and *Aspect-Oriented Programming (AOP)* [9]. AOP is a prominent programming technique that aims on modularizing crosscutting concerns. Due to its success in this respect it is worth while to consider it in context of product line development. We believe that there is no reason to understand aspects and features as concurrent concepts. We propose rather an architectural model to integrate both.

Following the AHEAD model we perceive aspects as artifacts that contribute to features. This is a different view than that of current research on aspects and features [6, 14]. These approaches use aspects as first-class entities to express features. Several studies have shown that this does not hold for a wide range of features, especially not in the context of complex evolved software [13, 1, 15]. The reason is that features are often implemented not by single classes or aspects but by the *collaboration* of sets of them. Common AspectJ[1]-like AOP languages cannot express and encapsulate collaborations and their refinements.

Approaches as *Aspectual Mixin Layers (AMLs)* [1], *Caesar* [15], and *Aspectual Collaborations (ACs)* [11] try to overcome this tension by combining collaborations and aspects. Whereas the latter two intermix structural elements of aspects and collaborations, AMLs propose the integration of aspects and features at architectural level. Since AMLs are based on AHEAD the programmer composes AMLs by specifying and evaluating algebraic equations. This opens the door to automatic optimization and large-scale compositional reasoning. AMLs have similar properties than Caesar and ACs, but however do not focus on on-demand remodularization and dynamic feature deployment.

This article focuses on AMLs because they follow the AHEAD architectural model. We use them to introduce the notion of *aspect refinement*. Since aspects are integrated in the AHEAD layer structure – in our case AMLs – the possibility of refining aspects arises. We perceive that as a natural

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

consequence following the AHEAD model. Aspect refinement facilitates the implementation, evolution, and reuse of aspects in a step-wise manner. Implementing aspect refinement we introduce the known concept of mixin-based inheritance [4] to aspects and AOP. This allows for flexibly altering the inheritance/refinement relations of aspects inside a layered feature stack.

The integration of aspects into the layered AHEAD architecture allows us further to reduce the complexity and unpredictability of aspects. Several studies have revealed that the unpredictable behavior of common aspects, especially in context of incremental designs, i.e. product lines, decreases the aspect reuse and complicates the evolution of such designs [12, 1, 11]. Based on the idea of aspect refinement, we propose a novel aspect bounding mechanism that takes the layered structure of the feature stack into account. This *bounded quantification* scopes aspects so that they only affect features of previous development stages. This prevents inadvertent effects on unanticipated features of subsequent development stages.

2 Aspectual Mixin Layers

This section reviews *Mixin Layers (MLs)* and *Aspectual Mixin Layers (AMLs)*. Both are techniques to implement features following the AHEAD model. Whereas MLs encapsulate classes and their collaborations, AMLs additionally include aspects. For explanation, we use FEATUREC++[2], a proprietary C++ language extension that supports MLs and AMLs. It uses *AspectC++*[3] for aspect weaving.

2.1 Mixin Layers

MLs are one appropriate technique to implement features [17, 3]. The basic idea is that features are often implemented by a collaboration of class fragments. A ML is a static component encapsulating fragments of several different classes (mixins) so that all fragments are composed consistently.

Figure 1 depicts a stack of three MLs ($L_1 - L_3$) in top down order. The MLs crosscut multiple classes ($C_A - C_C$). The rounded boxes represent the mixins. Mixins that belong to and together constitute a complete class are called refinement chain. Mixins that start a refinement chain are called *constants*, all others are called *refinements*.

In FEATUREC++ MLs 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 MLs.

[2]http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/
[3]http://www.aspectc.org/

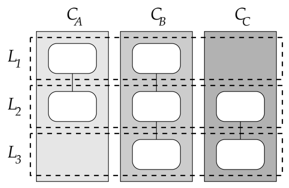

Figure 1. Stack of MLs.

Each constant and refinement is implemented by exactly one mixin. Constant classes form the root of a refinement chain. Refinements are applied to constants as well as to other refinements. Figure 2 depicts a constant (Line 1) and a refinement (Line 5). Programmers declare refinements by the *refines* keyword (Line 5). Usually, they introduce new attributes and methods (Line 6-7) or extend methods of their parent classes (Lines 8-11). To access the extended method the *super* keyword is used (Line 10). The *super* keyword has a similar semantic as the *proceed* keyword of AspectJ.

For a more detailed introduction to FEATUREC++, its capabilities, and its implementation we refer the reader to [1].

```
 1  class Buffer {
 2      char *buf;
 3      void put(char *s) { /*...*/ }
 4  };
 5  refines class Buffer {
 6      int len;
 7      int getLength() { return len; }
 8      void put(char *s) {
 9        if(strlen(s) + len < MAX_LEN)
10          super::put(s);
11      }
12  };
```

Figure 2. Constants and refinements.

2.2 Aspectual Mixin Layers

The key idea behind AMLs is to embed aspects into MLs (see Fig. 3). Each AML contains a set of mixins and a set of aspects.

Figure 4 shows a stack of AMLs that implements some buffer functionality, in particular, a basic buffer with iterator, a separated allocator, synchronization, and logging support. Whereas the first three features are implemented using common mixins, the *logging* feature is implemented using

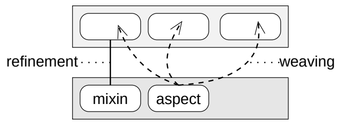

Figure 3. Aspectual Mixin Layers.

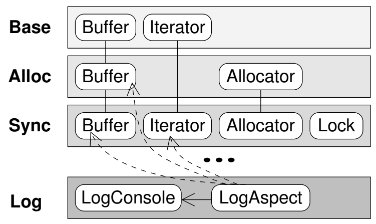

Figure 4. An AML for logging.

one mixin *and* one aspect. The rationale behind this is that the logging aspect captures a whole set of methods that will be refined (dashed arrows).

The embedding of aspects into AMLs has several advantages compared to MLs. Especially the ability of aspects to modularize certain kinds of crosscutting concerns is an improvement. However, these issues are out of scope of this paper. For more details we refer the interested reader to [1].

The close integration of aspects and features reveals that they are not concurrent concepts but enhance and support each other. Features organize the architecture in the large whereas aspects embedded into features improve the crosscutting modularity of these features. This architectural integration is a prerequisite for the ideas of aspect refinement and bounding aspect quantification.

3 Aspect Refinement

The integration of aspects into the layered architectural development style leads us to the notion of *aspect refinement*. Since aspects are encapsulated in AMLs it is natural to refine them incrementally, too. We perceive this stepwise refinement of aspects as natural consequence of the AHEAD architectural model. AHEAD states that all kinds of software artifacts that contribute to a feature can be refined incrementally. Several ideas of class refinement can be mapped to aspects, e.g. extending methods, introducing members, etc. But more interesting is the fact that it becomes possible to refine also aspect-specific constructs, in particular pointcuts and advice.

Mixins and mixin-based inheritance [4] are fundamental techniques for implementing refinements at code level. They allow to move the selection of the inheritance relation to parent classes from implementation to instantiation time. This increases the degree of flexibility and variability of mixins compared to common classes with common inheritance. Implementing aspect refinement we adopt the concept of mixin-based inheritance to aspects. Thus, aspects become mixins, too. Common aspect inheritance, e.g. as in AspectJ, suffers from the same inflexibility than traditional object-oriented inheritance. It is straightforward that a high degree of flexibility in altering the inheritance/refinement hierarchy yields a high degree of variability, customizability, and reusability. Furthermore, this leads to a unification of aspects and mixins in the context of layered architectures.

In order to prove their feasibility we implement aspect refinement using AMLs. 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 5 shows an AML (*ExtLog*) that refines a logging feature (*Log*). For that purpose it refines the logging aspect by additional join points in order to extend the set of intercepted methods (dashed arrows). Beside this, the logging console (implemented as a mixin) is refined by additional functionality, in particular by a modified behavior and appearance.

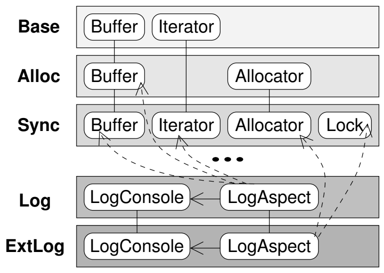

Figure 5. Aspect refinement.

Extending pointcuts increases the reuse of existing join point specifications (as in the logging example). Note that refining/extending aspects is conceptually different than applying aspects themselves. Applying two aspects (child and parent) separately 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. Doing so in the logging example, we have only one logging aspect instance.

Figure 6 depicts an logging aspect (Line 1) and an aspect refinement (Line 9). For simplification both are depicted in one listing. Normally, they would be implemented in two files located in two AMLs. The base aspect owns a method for formatting the logging output (Line 2), a pointcut[4] for defining the intercepted target method calls (Line 3), and an advice for executing the actual logging code (Lines 4-6).

The refining aspect (Line 9) extends the formatting method in order to adjust the output format (Lines 10-13). Moreover, it refines the parent pointcut in order to extend the set of target join points (Lines 14-15). Note, that every time the programmer accesses the parent aspect the *super* keyword is used (Lines 12,15). This is a key advantage of mixin-based inheritance and would not be possible with common AOP languages, e.g. AspectJ.

[4]The character '%' is a wild card and corresponds to the AspectJ's '*'.

```
aspect LogAspect {
    string format() { /*...*/ }
    pointcut log() = call("%␣Buffer::%(...)");
    advice log() : after () {
        cout << JoinPoint::signature() << endl;
    }
};

refines aspect LogAspect {
    string format() {
        /* do something */
        return super::format();
    }
    pointcut log() = call("%␣Allocator::%(...)") ||
                     super::log();
};
```

Figure 6. Refining an aspect by extending methods and pointcuts.

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 *classpects* that unify methods and advice [16] we would be able to refine advice analogously to methods.

4 Bounding Aspect Quantification

The close integration of aspects into the incremental development style of product lines 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 that aspects may affect subsequent integrated features. This can lead to unpredicted effects, e.g. an aspect is unintentionally bound to features of subsequent development stages. This may cause errors and unpredicted program behavior.

4.1 Incremental Development Example

Consider our buffer example. In a particular development stage we add support for serializing and storing basic data types (see Fig. 7). For each supported data type the buffer provides an adequate method to store single items (Lines 2-5).

In a subsequent step we add synchronization support. Therefore, we introduce an aspect because this is the best way to implement such a homogeneous crosscutting concern (see Fig. 8). The synchronization aspect intercepts calls to the *put* methods (Line 2) and sets lock variables around the execution of these methods (Lines 3-7).

In a third development step we add support for serializing and storing arrays (see Fig. 9). This refinement uses functions of the basic buffer that implements basic data type

```
1 refines class Buffer {
2     void put(int item) { /*...*/ }
3     void put(float item) { /*...*/ }
4     void put(double item) { /*...*/ }
5     /*...*/
6 };
```

Figure 7. Adding support for basic data types.

```
1 aspect SyncAspect {
2     pointcut sync() = call("%␣Buffer::put%(...)");
3     advice sync() : around() {
4         lock();
5         tjp->proceed();
6         unlock();
7     }
8 };
```

Figure 8. Synchronizing access to buffers.

support. This step introduces *put* methods for each supported array type. These methods readout the arrays and call the *put* methods of the parent buffer that handle single data items.

```
refines class Buffer {
    void put(int *array, int len) { /*...*/ }
    void put(float *array, int len) { /*...*/ }
    void put(double *array, int len) { /*...*/ }
    /*...*/
};
```

Figure 9. Adding support for arrays.

Introducing this refinement leads to a problem: The synchronization aspect weaves code (*lock*, *unlock*) not only to the *put* methods of the basic data type buffer but also to all subsequent buffer refinements, e.g. the buffer that supports arrays. During the implementation of the synchronization aspect this was not intended and could not be foreseen. In this example this leads to a doubled synchronization that wastes resources. In more complex buffer hierarchies with concurrent access this may produce unpredictable effects, e.g. deadlocks, inconsistent buffer content, etc.

The programmer has always the opportunity to restrict aspects by fully quantifying the target join points, i.e. by enumerating and narrowing down the set of potential join points; but this decreases the reusability of aspects in other contexts. Furthermore, programmers may define conventions for naming methods, etc. However, it is a non-trivial task to have an overview of complex software. Thus, unanticipated interactions may occur unnoticed.

```
1  aspect SyncAspect_Sync {
2      pointcut sync() = call("%␣Buffer_Sync::put(...)") ||
3                        call("%␣Buffer_Base::put(...)");
4  };
```

Figure 10. Transformed pointcut.

4.2 A Novel Bounding Mechanism

Lopez-Herrejon and Batory state that this problem is not a question of good or bad design but merely it is caused by the aspect quantification mechanisms of current AOP languages [12]. That means that during the evolution of complex software it is unavoidable that such situations occur.

Since common AOP languages cannot distinguish between software artifacts of different development stages they cannot scope their appliance. Integrating aspects into incremental designs allows for assigning the implementation units to development stages and to define a natural order.

To overcome this tension Lopez-Herrejon and Batory propose an alternative aspect composition mechanism [12]. They argue that with regard to software (product line) evolution, features should only affect features of previous development stages. Mapping this to AOP means that aspects should only affect elements assigned to development stages that were already present at the aspect's implementation time. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. As Section 4.1 has shown this decreases aspect reuse and complicates incremental software development. Consequently, AMLs with their support for aspect refinement follow this principle: *aspects affect only artifacts of previous development stages*.

In order to implement this bounding mechanism in FEATUREC++, the aspects 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 match only types of the current and the parent features.

To get back to our buffer example, this novel bounding mechanism transforms the synchronization aspect, i.e. its pointcut, as depicted in Figure 10. The extensions of the class and aspect names, e.g. *_Sync*, are introduced by the FEATUREC++ compiler and are exploited by the bounding mechanism. It can be seen that the new pointcut matches only types of the current and parent layers (Lines 2-3). Whereas this transformation is specific to the FEATUREC++ compiler the idea of restructuring and bounding pointcuts is easily applicable to other AOP languages.

This bounding mechanism is supposed to support the programmer to develop software incrementally. Since our bounding mechanism is not desired in all situations the programmer has always the possibility to intervent the bounding policy and let the introduced aspects affect all development stages. However, this mechanism points to possible problems and assists the programmer to prevent errors.

5 Discussion

AMLs propose the close integration of aspects and features. They are no concurrent concepts but enhance each other [1]. The idea of aspect refinement is a direct result of this integration. Since aspects are encapsulated in features they can be refined, too. We perceive this as a natural consequence of the incremental development style. 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 3 has shown that there are indeed certain applications of this concept. FEATUREC++ supports aspect and pointcut refinement already. Advice refinement would be possible if methods and advice became unified, e.g. by exploiting ideas of *classpects* [16].

The advantage of aspect refinement is the possibility to evolve aspects over several development stages. It contributes a unification of aspects and classes in this respect. In order to realize aspect refinement we introduce the capabilities of mixins in flexibly altering the inheritance hierarchy to aspects and AOP. This is an improvement to aspects, similar to mixin-based inheritance with respect to classes.

Aspect refinement opens the door to bound the quantification of aspects. Our novel bounding mechanism allows to scope aspects and prevents thereby 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. Thus, the integration of aspects into AMLs 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 software development.

Although, we used AMLs and FEATUREC++ to implement aspect refinement and the bounding mechanism, the ideas are applicable to other programming languages. The only prerequisite is an explicit layered architecture that integrates aspects.

6 Related Work

Current AOP languages already support aspect inheritance. Aspect inheritance can be used to implement refinements. The drawbacks are that these aspects are not encapsulated in feature modules and that the inheritance hierarchy is fixed at implementation time. Aspect refinement exploits mixin-based inheritance to overcome this tension.

Mezini et al. propose *Caesar* that combines aspects and collaborations [15]: Aspects rely on *aspect collaboration interfaces* that decouple an aspect's implementation from its binding. By defining a binding, the programmer adapts the 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 or mixin-based inheritance.

As with Caesar *Aspectual Collaborations (ACs)* [11] and *Object Teams (OTs)* [8] 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 extend the AHEAD architectural model. Although the others do not propose such model we perceive, if it is, *GenVoca* as their architectural model [2]. AHEAD has several strength 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 [3]. Furthermore, AMLs implement aspect refinement with a bounded aspect quantification for the first time. This improves the capabilities of aspects to implement incremental refinements.

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

A couple of work aims on exploiting generic types for aspects in order to support the evolution of complex software and to serve unanticipated requirements [7, 14, 5, 10]. All these approaches are related to AMLs: All allow to parameterize aspects at instantiation time but only AMLs embed aspects into collaborations and support mixin-based inheritance.

7 Conclusions

In this paper, we examined the relationship between incremental software development and AOP. According to AHEAD we perceive aspects also as artifacts that contribute to features. This is different from the view of previous work. The embedding of aspects into the layered architectural style increases the power of FOP. Aspects improve the crosscutting modularity of features. Based on AMLs we introduced the notion of aspect refinement. Aspects encapsulated in AMLs refine other aspects by extending their methods, adding members, and refining pointcuts. Thus, aspects can easily be extended and evolved over several development stages. The concept of aspect refinement unifies aspects with other software artifacts with regard to stepwise refinement. Realizing aspect refinement we introduced mixin-based inheritance to aspects and AOP. This improves flexibility, customizability, and reusability.

The novel bounded aspect quantification exploits the concept of aspect refinement in order to reduce the complexity and unpredictability of aspects. The naturally layered architecture of AHEAD allows to bound aspects only to artifacts of previous development stages. This reduces unpredictable aspect interactions and improves incremental software development using AOP techniques.

References

[1] S. Apel et al. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*, 2005.

[2] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1(4), 1992.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.

[4] G. Bracha and W. Cook. Mixin-Based Inheritance. In *OOPSLA/ECOOP*, 1990.

[5] J. A. Canal. Parametric Aspects: A Proposal. In *ECOOP RAM-SE Workshop*, 2004.

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

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

[8] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NetObjectDays*, 2003.

[9] G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, 1997.

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

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

[12] R. Lopez-Herrejon and D. Batory. Improving Incremental Development in Aspectj by Bounding Quantification. In *SPLAT*, 2005.

[13] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.

[14] N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *AOSD ACP4IS Workshop*, 2004.

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

[16] H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *ICSE*, 2005.

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