Aspect Refinement in Software Product Lines

Sven Apel, Thomas Leich, and Gunter Saake

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

Abstract. This article investigates aspects in the context of the step-wise development of 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 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. Furthermore, we propose a bounding quantification that reduces the complexity and unpredictability of aspects in incremental software development. A novel bounding mechanism exploits the natural order of the layered architecture introduced by the concept of aspect refinement. Aspect refinement and bounding quantification improve the 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 a concrete program. Doing so, products added to a product line can 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 investigates the relation of *Aspect-Oriented Programming (AOP)* [6] and AHEAD. 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.

Following the AHEAD model we propose that aspects are also artifacts that contribute to features. This is a different view than that of current research on aspects and product lines [4, 10]. These approaches perceive 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 product lines [9, 2, 11]. The reason is that features are often implemented not by single classes or aspects but by the collaboration of sets of them. Common AspectJ-like AOP languages cannot express and encapsulate collaborations and collaboration refinements.

Approaches as *Aspectual Mixin Layers (AMLs)* [2], *Caesar* [11], *Aspectual Collaborations* [7] try to overcome this problem by combining collaborations and aspects. 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 step following the AHEAD model. Aspect refinement opens the door to evolve aspects in a step-wise manner. This improves aspect reuse and evolution.

The integration of aspects into the layered AHEAD architecture allows us 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, decreases the aspect reuse and complicates the evolution of such designs [8, 2, 7]. Based on the idea of aspect refinement, we propose a novel aspect bounding mechanim that takes the layered structure of the feature stack into account. This *bounding 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 Integrating Aspects into AHEAD

This section reviews *Mixin Layers (MLs)* and *Aspectual Mixin Layers (AMLs)*. Both are implementation techniques that follow the AHEAD model. Wheras MLs encapsulate classes and class refinements, AMLs additionally include aspects and their refinements. For explanation, we use FeatureC++[1], a proprietary C++ language extension that supports MLs and AMLs.

2.1 Mixin Layers

MLs are one appropriate technique to implement features [13, 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. Advantages are a high degree of modularity and an easy composition [13].

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. Refinement chains are connected by vertical lines. Mixins that start a refinement chain are called *constants*, all others are called *refinements*. A mixin A that is refined by mixin B is called the *parent* mixin or parent class of mixin B. Consequently, mixin B is the *child* class or child mixin of A. Analogously, we speak of parent and child MLs.

In FeatureC++ MLs are represented by file system directories. Therefore, they have no programmatic representation. Those mixins found inside the directories are assigned to be members of the enclosing MLs.

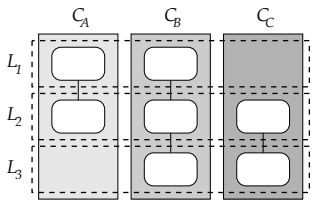

Fig. 1. Stack of MLs.

```
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)
9          super::put(s);
10     }
11 };
```

Fig. 2. Constants and refinements in FeatureC++.

Each constant and refinement is implemented by a mixin inside exactly one source file. The root of a refinement chain is formed by these constants (see Fig. 2, Line 1). Refinements are applied to constants as well as to other refinements. They are declared by the *refines* keyword (Line 5). Usually, they introduce new attributes and methods (Line 6) or extend[2] methods of their parent classes (Lines 7-10). To access the extended method the *super* keyword is used (Line 9). The *super* keyword has a similar semantic as the *proceed* keyword of AspectJ/AspectC++.

For a more detailed introduction to FeatureC++, its capabilities, and its implementation we refer to [2, 1].

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

[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.

2.2 Aspectual Mixin Layers

The key idea behind AMLs is to embed aspects into MLs (see Fig. 3). Each ML contains a set of mixins and a set of aspects. This methodology follows AHEAD which proposes that features of a product consist of different kinds of software artifacts. Aspects are beside classes another kind of software artifact.

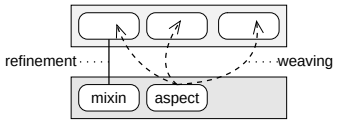

Fig. 3. AMLs: Embedding aspects into MLs.

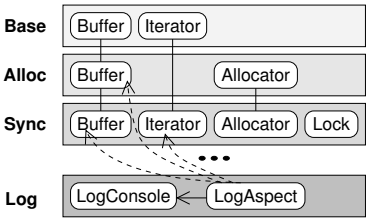

Fig. 4. Implementing a logging feature using AMLs.

Figure 4 shows a stack of MLs 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 as common MLs, the *logging* feature is implemented as an AML. 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. We refer to [2] for more details.

3 Aspect Refinement

The introduction of aspects into AHEAD leads us to the notion of *aspect refinement*. Since aspects are introduced in MLs it is natural to refine them incrementally, too. We perceive this step-wise 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 also refine aspect-specific constructs, in particul pointcuts and advice.

We have used AMLs to implement aspect refinement. 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 that refines a logging aspect – included in a logging feature (see Fig. 4) – by additional join points to extend the set of intercepted methods. Beside this, the logging console (implemented as a mixin) is refined by additional functionality, in particular by a modified output format.

Generally, aspects can refine the methods of parent aspects as well as the parent's pointcuts. 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 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 instance. See [1] for a details on the implementation in FeatureC++.

Figure 6 depicts an aspect refinement that extends a logging feature, including a logging aspect. It extends a parent method in order to adjust the output format (Line 2-5) and refines a parent pointcut to extend the set of target join points (Lines 6-8). Both is done using the *super* keyword (Line 4,8).

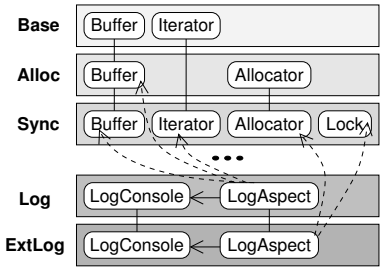

Fig. 5. Refining an AML.

```
1  refines aspect LogAspect {
2      void print() {
3          changeFormat();
4          super::print();
5      }
6      pointcut log() =
7          call("% Buffer::put(...)") ||
8      super::log();
9  };
```

Fig. 6. An aspect embedded into an AML.

4 Bounding Quantification

The close integration of aspects into the incremental development style of product lines and AHEAD 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 lead to errors and unpredicted program behavior. Since common AOP languages cannot distiguish between software artifacts of different development stages they cannot scope their appliance. Integrating aspects into incremental designs implemented by MLs makes it possible to assign the implementation units to development stages and to define a natural order.

In [8] Lopez-Herrejon and Batory propose an alternative aspect composition mechanism. They argue that with regard to software (product line) 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 aspect's implementation time. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. 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 the AMLs of FeatureC++, the user-declared join point specifications of 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 refer to all type names of the parent classes. Figure 7 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 8. It can be seen that the new pointcut matches only types of the current and parent layers (Lines 3-5).

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

Fig. 7. A simple pointcut expression.

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

Fig. 8. Transformed pointcut.

5 Discussion

The integration of aspects into AHEAD contributes several new opportunities to the programmer. It enables the programmer to choose the adequate technique for a given problem: He can use mixins to implement common features in form of MLs and he can use aspects to implement certain crosscutting features. Which technique is most appropriate depends on

the features to be implemented and the features that are already present. [2, 1] give a set of guildelines when using aspects and when using mixins. Note that often the programmer does not use aspects stand-alone but encapsulated in AMLs with mixins in collaboration.

The introduction of aspects has led us to the proposal of aspect refinement. Since aspects are encapsulated in MLs they can be refined, too. We perceive this as a natural consequence of the AHEAD architectural model. 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 refinements would be possible if methods and advice became unified, e.g. by exploiting ideas of *classpects* [12].

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. Furthermore, it opens the door to bounding quantification. Our novel 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. Thus, the integration of aspects into MLs 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 product line 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

Mezini et al. propose *Caesar* that combines aspects and collaboration [11]. 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 Caesar *Aspectual Collaborations (ACs)* [7] and *Object Teams (OTs)* [5] 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 strength compared to its precedessor 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]. Futhermore, AMLs firstly implement aspect refinement with bounding quantification.

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

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 [4]. They distinguish between orthogonal and weak-orthogonal features. But they do not disguish between the structural properties and conceptual differences of aspects and features. The discussion of AMLs and aspect refinement gives the insight that using aspects stand-alone has several weaknesses in implementing features. The integration in collaborations unifies and improves AOP and AHEAD methodolgy.

Loughran et al. support the evolution of program families with *Framed Aspects* [10]. They combine the advantages of frames and AOP in order to serve unanticipated re-

quirements. Framed Aspects are related to AMLs. Both allow to parameterize aspects at instantiation time but AMLs embed aspects into collaborations and supports step-wise aspect refinement.

7 Conclusions

In this paper, we have investigated the relation between the AHEAD architectural model for incremental product line development and AOP. According to AHEAD we perceive aspects also as software artifacts that contribute to features. This is different from the view of former work in this field. The embedding of aspects into the AHEAD layered architecture increases the power of these features. Aspects improve their crosscutting modularity. Based on AMLs we have introduced the notion of step-wise aspect refinement. Aspects encapsulated in AMLs refine other aspects by extending their methods, adding members, and refining pointcuts. The concept of aspect refinement unifies aspects with other software artifacts with regard to step-wise refinement. Integrated in AHEAD, aspects can easily be extended and evolved over several development stages. The novel bounding 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.

Finally, we want to emphasize that the idea of aspect refinement and bounding quantification does no depend on a specific programming language, but is merely a general concept of integrating and handling aspects in context of incremental designs, e.g. software product lines.

References

1. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
2. S. Apel et al. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*, 2005.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
4. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.
5. S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NetObjectDays*, 2003.
6. G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, 1997.
7. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal (Special issue on AOP)*, 46(5), 2003.
8. R. Lopez-Herrejon and D. Batory. Improving Incremental Development in Aspectj by Bounding Quantification. In *SPLAT*, 2005.
9. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.
10. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *AOSD ACP4IS Workshop*, 2004.
11. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
12. H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *ICSE*, 2005.
13. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.