Technical Report Nr. 10/2005 of the University of Magdeburg, Germany

Mixin-Based Aspect Inheritance

Sven Apel, Thomas Leich, and Gunter Saake

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

Abstract. Introducing mixin-based inheritance to AOP improves the capabilities of aspects to implement incremental designs. Since it enhances the inheritance mechanism with more flexibility, it is a key technology to reuse, compose, and evolve aspects in a step-wise manner over several development stages (*step-wise aspect refinement*). The introduction of mixin capabilities to aspects leads to the unification of the aspect's structural elements with respect to step-wise refinement, i.e. in a chain of refining aspects every element (methods, pointcuts, advice) can be refined by subsequent elements without knowing the exact type of the enclosing aspect. In this context we introduce anonymous calls to parent pointcuts as well as *named advice*, a unification of advice and methods. Making these extensions to aspects, aspect refinement enables the programmer to implement higher-order aspects – aspects that modify other aspects. Furthermore, we show how to tame aspects using a bounded aspect quantification based on the knowledge of incremental designs. We contribute an extended AspectJ grammar, its formal semantics, as well as an implementation approach based on program transformation.

1 Introduction

Aspect-oriented programming (AOP) is a powerful programming paradigm to implement and evolve complex software in a modular a way [1]. In concert with classes, aspects localize, separate, and modularize certain kinds of crosscutting concerns. Without aspects, the implementation of such concerns would be scattered over and tangled with the implementation of other concerns.

This paper aims at several issues that arise from relating AOP and *incremental software development*. Incremental software development (a.k.a. *step-wise refinement*) is a fundamental development approach to software engineering [2–5]. The successive adding of new program elements permits to break down complex software into manageable pieces (*modules*). These subsequently added elements are called *refinements* and are implemented in different stages of the development and evolution of the software product. This methodology promotes reuse, customization, and maintenance and follows directly the prominent principle of *separation of concerns* [3].

Several recent studies have revealed that current AOP mechanisms – despite their advantages – are not well suited to be simply integrated into the methodology of incremental software development: Aspects are problematic with respect

to changing and evolving software [6, 7]. Inadvertent side effects of applying aspects may result in a difficult to comprehend or even unpredictable behavior [8]. Furthermore, it is well-known that aspects are less reusable in context of incremental designs [7, 9] and they are problematic w.r.t. modular reasoning [10, 11]. In a recent study we explained how aspects are related to program features and why aspects should be integrated into well known programming techniques, e.g. collaborations, design patterns, components [12, 13]. In summary, we demonstrated that aspects and features are not competitive but merely are symbiotic techniques to structure software at different scales, i.e. aspects modularize certain kinds of crosscutting concerns whereas features organize collaborations of classes *and* aspects at a higher level of abstraction.

In order to integrate aspects into the methodology of incremental software development, we proposed the notion *aspect refinement* [14]. Aspect refinement is a methodology to incrementally evolve aspects in layered architectures using step-wise refinement. Whereas our former work aims at the architectural integration of aspects and features in context of feature-oriented program families, this paper proposes an explicit implementation of these concepts, but with a broader focus on incremental software development. In order to implement aspect refinement we introduce in this paper the notion of *mixin-based inheritance* [15] – known from the object-oriented world – to AOP. Common aspect languages as AspectJ already support a limited form of aspect inheritance. But the novel notion of *mixin-based aspect inheritance* allows the programmer to flexibly alter the inheritance relationships between aspects and therewith the refinement chain that has evolved over several development stages. Mixin-based inheritance is a central technique to implement aspect refinement. Introducing mixin-based aspect inheritance provides the required flexibility for composing, reusing, and evolving aspects that is necessary to implement and handle highly customizable, incremental designs, i.e. program families. Although we focus in our considerations on extending *AspectJ*[1], our results are applicable to other AOP languages.

Aspect refinement and mixin-based aspect inheritance are highly related to other approaches that aim at step-wise refinement and AOP, i.e. the AHEAD model for large-scale compositional programming [5] and Caesar that supports multi-abstraction and layered aspect components [16]. We believe that our approach is fundamental to the integration of AOP and incremental software development. In a detailed discussion of related work we show the close relationship, the differences, as well as the synergetic potential of integrating our ideas into AHEAD and Caesar.

The notion of aspect refinement and the introduction of mixin-based inheritance unify aspects and classes with respect to incremental designs. This unification leads to the opportunity to refine structural elements of aspects similar to those of classes. Known from classes, a refining aspect can add new fields and methods, but also new pointcuts and advice. Moreover, an aspect can extend methods of parent aspects by overriding their methods, adding code, modifying arguments, and calling the parent method. Specifically, code can be added be-

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

fore, after, and around a parent method. Besides those well-known mechanisms, we propose also to refine structural elements specific to aspects, particularly refining pointcuts and advice.

In order to implement the refinement of advice, we propose to unify advice and methods. This idea is adopted from classpects [17] but has a different focus. We do not intend to unify classes and aspects in the sense that they can be uniformly instantiated and invoked. Merely, we introduce the notion of *named advice* that unifies invocations of methods and advice invocations inside aspects. Since named advice are first class entities, programmers can refine them similar to other structural elements, e.g. methods. Thus, it becomes possible to treat all aspect's structural elements equally with respect to refinement.

Having all these extensions to AspectJ, we are able to implement higher-order aspects: A child aspect may modify or extend a previous aspect by refining its structural elements. Furthermore, the integration of aspects into the incremental development style allows for implementing a novel kind of *bounded aspect quantification* that reduces unpredictable effects in evolving designs.

In this paper we make the following contributions:

- We review certain shortcomings of aspect techniques to implement incremental designs.
- We present the notion of mixin-based aspect inheritance in order to implement step-wise aspect refinement.
- We explain how to implement mixin-based inheritance in AspectJ-like languages as well as corresponding mechanisms for refining aspects, methods, pointcuts, and *named* advice.
- We present a bounding mechanism for aspect quantification that exploits the layer structure of incremental designs.
- We give a formal description of a grammar that extends AspectJ, its formal semantics, as well as an approach for extending AspectJ based on program transformation.

2 Aspect Refinement

This section briefly discusses problems of aspects when integrated into incremental designs. Afterwards, we describe the notion of aspect refinement and how it addresses these problems.

2.1 Problems of Aspects in Incremental Software Development

Aspects have numerous strengths with regard to localization, separation, and modularization of crosscutting concerns. However, at this point we want to discuss the weaknesses of current AOP languages in seamlessly integrating aspects into incremental designs. Our considerations extend several recent studies on this field [6–8, 13, 12, 14].

Aspect inheritance. In most AOP languages, aspects look like classes. The have similar structural elements but add also some new ones, i.e. pointcuts and

advice. Interesting for this article is that aspects may inherit from other aspects (and classes). Inheritance is known as a concept for reusing and non-invasively refining parent classes (and aspects) [18]. In the OOP community an extension to common inheritance was proposed in order to increase the compositional flexibility, called *mixin-based inheritance* [15]. In contrast to delegation and dynamic binding, common inheritance dictates a fixed refinement hierarchy. Mixin-based inheritance overcomes this tension by moving the selection of the parent to composition time. This increases reusability in different contexts, allows for incremental refinement, and improves customizability. Unfortunately, current AOP languages do not support mixin composition of aspects.

Constrained aspect extension. Using common aspect inheritance in AspectJ an aspect has to be declared as *abstract* to be able to be refined. This means that adding a child aspect requires the programmer to modify the parent aspect. This and similar requirements[2] cause a fundamental problem of AspectJ-like languages with regard to incremental software development. Implementing an aspect in a particular development stage forces the programmer to decide whether the aspect would be refined in a later stage. Unfortunately, this cannot always be anticipated by the programmer. Thus, the programmer is in a serious dilemma. Declaring the aspect as abstract makes it necessary to add later at least one conrete child aspect. But this may not happen and hence the aspect does not work. If the programmer decides to declare an aspect as concrete (without modifier) he prevents the later refinement of this aspect. In our approach of mixin-based aspect inheritance we address this problem by permitting refinements of concrete aspects.

Unpredictable behavior. Aspects provide a powerful mechanism for specifying and determining sets of join points which the aspects bind to. Unfortunately, this does not fit to the idea of step-wise refinement and incremental designs. These approaches to software development structure software in different layers that build up on one another. Layers that represent development stages are restricted to affect only those layers of previous development stages [4]. Unfortunately, aspects do not care about this natural order. Often aspects inadvertently affect features of later development stages. This circumstance is directly responsible for unpredictable effects that occur when new classes are added to the architecture and previous aspects interfere with these new elements. The key problem is that programmers cannot anticipate such interactions. This is not a matter of bad design but directly related to the aspect binding mechanism [7, 6]. Current mature aspect languages usually bind aspects based on syntactic information that is less reliable against subsequent changes. This violates the principle of information hiding. Furthermore, there are no sophisticated mechanisms to bound and control aspects. However, some first studies partly address these issues, e.g. [11, 19, 10, 7, 20].

[2] For example, refining a pointcut in AspectC++ requires to declare the parent pointcut as *virtual*.

Advice is not first-class. Pointcuts specify the set of join points an aspect is woven to. Advice execute code at the matched join points. Rajan and Sullivan state that advice are unnecessary different from other structural elements of aspects [17]. Advice is invoked implicitly when a pointcut matches. This prevents other advice or methods to invoke them explicitly. Thus, advice is the only unnamed structural element of aspects. Instead, fields, methods, and pointcuts are named entities that can be referred to by others entities. Therefore, in [17] a unification of classes and aspects is proposed that also includes a unification of advice and methods. In this approach advice can be invoked like methods. We argue that this unification is crucial for implementing aspect refinement and mixin-based inheritance in AOP languages.

2.2 Step-Wise Refinement of Aspects

In order to map aspects to the methodology of incremental software development we proposed the notion of *aspect refinement* [14]. In a preliminary study we focused mainly on the AHEAD architectural model [5] for implementing feature-oriented program families and *aspectual mixin layers* [13] as component technique. In this work we now generalize these ideas to all kinds of incremental designs. We argue that the notion of aspect refinement does not necessarily depend on features, collaborations, or program families, but merely is a fundamental concept for incremental development using aspects.

Since we want to integrate the paradigm of AOP into the methodology of developing software incrementally, it is natural to refine also aspects incrementally. Thus, aspects – as with all other software artifacts – evolve over time, from development stage to development stage. This view is consistent with the *principle of uniformity* that predicts that every kind of software artifact is refineable [21]. An advantage of this view is that several ideas of class refinement can be mapped directly 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, which opens the door to a novel dimension of aspect reuse.

Incremental development example. Figure 1 depicts an example that is used in the remaining article to explain our ideas. It consists of four steps shown in four subfigures (I-IV). Each development step is explained in terms of its AspectJ code and in diagram form. Refinements introduced in a particular development stage are highlighted bold. Weaving is displayed by dashed arrows.

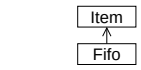
I. In the initial class hierarchy a *Fifo* buffer stores a set of data items. *Fifo* provides a *put* and a *get* method for storing and retrieving data items.
II. In a subsequent step we introduce a synchronization aspect that locks the access to the *put* and *get* methods of *Fifo* via *lock* and *unlock*.
III. Furthermore, we add a *Stack* class that has to be synchronized, too. *Stack* is derived from *Fifo* and the synchronization aspect is refined to match also the methods of *Stack* (*push*, *pop*). This step shows that it could be meaningful to refine aspects. In this example the refining aspect extends the

set of intercepted method calls for synchronization by overriding and reusing the parent pointcut.

IV. Finally, we introduce *Socket* that uses *Fifo* and *Stack* objects. We limit the set of matched join points to those that are inside the control flow of *Socket*. This is implemented as an aspect refinement that restricts the join point set.

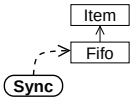
The example clarifies the usefulness of refining aspects in a step-wise manner over several development stages; but it makes also some of the discussed weaknesses clear, e.g. only abstract aspects can be refined. Nevertheless, aspect refinement is a logical consequence of incremental software development mapped to AOP. Refinements can be implemented in numerous ways, e.g. inheritance, wrappers, filters, common AOP mechanisms. However, mixin-based inheritance has certain strengths that qualify it for implementing refinements and, therewith, for solving the mentioned problems. Several studies on mixins and OOP indicate its success in this respect [22–25].

```
1  class Fifo {
2    Vector buf = new Vector();
3    void put(Item e) { buf.add(e); }
4    Item get(int i) { return (Item)buf.get(i); }
5  }
```



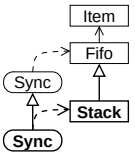
(I)

```
1  abstract aspect FifoSync {
2    pointcut syncPC() : execution(* Fifo.get(..)) ||
3                        execution(* Fifo.put(..));
4    Object around() : syncPC() {
5      lock(); Object res = proceed(); unlock();
6      return res;
7    }
8  }
```



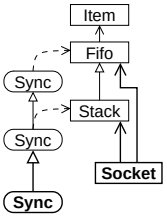
(II)

```
1  class Stack extends Fifo {
2    void push(Item i) { super.put(i); }
3    Item pop() { return super.get(buf.size() - 1); }
4  }
5  abstract aspect StackSync extends FifoSync {
6    pointcut syncPC() : FifoSync.syncPC() ||
7                        execution(* Stack.pop()) ||
8                        execution(* Stack.push());
9  }
```



(III)

```
1  class Socket {
2    void receive() {
3      Fifo fifo = new Fifo();
4      Stack stack = new Stack(); /* ... */
5    }
6  }
7  aspect SyncSocketsOnly extends StackSync {
8    pointcut syncPC() : StackSync.syncPC() &&
9                 cflow(execution(* Socket.*(..)));
10 }
```



(IV)

Fig. 1. An incremental development example (AspectJ code).

3 Mixin-Based Inheritance for Aspects

This section introduces mixin-based aspect inheritance. First we review the original approach of mixin-based inheritance; then we show how it can be adopted for implementing refinements of aspects. Specifically, we explain the different opportunities to refine aspects such as adding fields and methods, extending existing methods, refining pointcuts and advice. We emphasize the last two because they are crucial for aspect refinement.

3.1 Mixin-Based Inheritance and Program Refinements

Program refinements (refinements for short) add new, or alter and extend existing functionality. In context of OOP that means adding new fields and methods as well as extending existing methods. Inheritance is an appropriate mechanism to implement incremental program refinements [18]. Different object-oriented languages use different mechanisms for inheritance [15]. *Mixin-based inheritance* introduced by Bracha and Cook [15] is a more flexible approach to inheritance. They introduce the notion of *mixins* that can be parameterized with parent classes. Mixins are *abstract subclasses* that extend a family of parent classes. That means they can be applied to different classes implementing *reusable* extensions and are therefore qualified to implement refinements. Mixin composition is also called *instantation of mixins* and should not be confused with instantiating a class to get an object. The resulting composition that forms the final class is called *inheritance chain* or *refinement chain*. Figure 2 shows a set of mixins and four generated refinement chains.

The Jak language – Refinements for Java. Jak extends Java with mixin capabilities. It is part of the *AHEAD Tool Suite* for large-scale compositional programming and step-wise refinement [5]. Jak serves as our archetype for enhancing a language with constructs for implementing refinements via mixins. We adopt its constructs and map them to aspects. Refinements are declared using the *refines* keyword. Figure 3 depicts *Fifo* (Line 1) and a refinement (Line 5) that introduces a maximum size (Line 6). The refinement extends *put* with a check of the current size against the maximum size (Lines 7-9). It can be seen

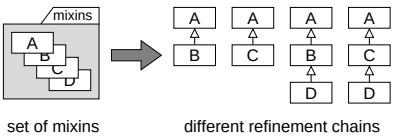


Fig. 2. Composing mixins in different ways to form alternative refinement chains.

```
1  class Fifo {
2    void put(Item e) { }
3    Item get(int i) { }
4  }
5  refines class Fifo {
6    static int max = 100;
7    void put(Item e) {
8      if(fill < max)
9        super.put(e);
10   }
11 }
```

Fig. 3. Refinements in Jak.

that in contrast to common subclassing a Jak mixin (that is an abstract subclass) has no explicit name (Line 5). It simply implements a refinement to *Fifo*. Although, in the original approach mixins are named entities we consider only unnamed mixins for implementing refinements. The advantage of mixins is that different refinements (to *Fifo*) can be composed to a compound class (the final *Fifo* class) in *different permutations*. The names of the mixins do not matter.

Mixin-based inheritance is more flexible than common inheritance. A programmer can select a set of program refinements and compose them to form a final class (compound class). In the remaining paper we use mixin-based inheritance to implement incremental refinements of aspects. Each mixin of a refinement chain corresponds to a particular development stage. We further assume that mixins are composed at compile time.

3.2 Aspects and Mixin Composition

Introducing mixin capabilities to aspects means that a programmer can compose aspects via mixin-based inheritance. Thus, aspects can refine other aspects by adding and extending functionality. To express aspect refinements we adopt the syntax of the Jak language. Hence, aspects as well as their refinements have equal names. Figure 4 shows a synchronization aspect (Line 1) and a refinement (Line 5) that adds a new field and that extends the locking methods to count the number of threads that access the locked critical section. Refinements are distinguished based on their affiliation to development stages.

The semantics of such mixin-based inheritance is as follows. An aspect together with all of its refinements constitute the final aspect that is woven *once* to the final program. Hence, the elements of the refinement chain have to be composed. A possible implementation is to translate mixin-based inheritance into common aspect inheritance: Aspect refinements are implemented as (non-abstract) subaspects of their parents. Section 6 explains those and alternative transformations in more detail.

With such kind of mixin-based inheritance one does not need to declare the parent aspect as abstract. This might seem to be of no great advantage but this eliminates the dilemma to anticipate subsequent added features.

3.3 Adding Members and Extending Methods

Aspects may extend parent aspects by adding new members. Figure 4 depicts an aspect and a refinemet that adds a field (Line 6), a pointcut (Line 9), and an advice (Lines 10-12). Known from classes, aspects may also extend existing methods. An extension to a method overrides, adds some code, and usually calls the extended method (Lines 7,8).[3] Refining aspects is in parts already supported by common aspect inheritance but without the flexibility of mixin composition and anonymous refinements.

[3] Notice that a second way to refine a method is to use the common pointcut-advice-mechanism. However, this is out of scope of this paper.

```
1  aspect Sync {
2    void lock() { /* locking access */ }
3    void unlock() { /* unlocking access */ }
4  }
5  refines aspect Sync {
6    int threads;
7    void lock() { threads++; super.lock(); }
8    void unlock() { threads--; super.unlock(); }
9    pointcut syncPC() : execution(* Fifo.get(..)) || execution(* Fifo.put(..));
10   Object around() : syncPC() {
11     lock(); Object res = proceed(); unlock(); return res;
12   }
13 }
```

Fig. 4. Adding members and extending methods.

3.4 Pointcut Refinement

When inheriting from a parent aspect a child aspect can override and extend the parent pointcuts. Hanenberg et al. propose several design patterns that utilize those techniques to improve reusability and extensibility of aspects, e.g. *composite pointcut*, *pointcut method*, *template advice* [26].

Recall our example aspect that synchronizes calls to *Fifo* (cf. Fig. 1). For this aspect we defined two refinements, an aspect that extends this set by all method calls to *Stack* (III) and an aspect that constrains the set of join points to calls that originate from *Socket* (IV). Both aspects were derived using common aspect inheritance. They override the parent pointcut and define the new one by reusing the parent pointcut (*syncPC*) and by adding new pointcut expressions that extend or constrain the set of matched join points. Hence, the child aspects reuse the parent aspect's functionality for synchronization. Recall that applying these two refinements extends the synchronization aspect and does not create two new aspects. Only the final compound aspect is woven to the target program.

In common AspectJ parent pointcuts have to be accessed by their full-qualified name, e.g. *FifoSync.syncPC*. Thus, the programmer is forced to hard-wire the parent and the child aspect. This tight coupling decreases reusability. Having mixin-based inheritance the child aspect does need to know the concrete parent aspect. Figure 5 depicts the synchronization aspect for *Fifo* and our refinements regarding *Stack* and *Socket*, but now using mixin-based aspect inheritance. This example clarifies an important advantage of mixin-based aspect inheritance related to pointcut reuse. Using the *super* keyword the programmer can refer to the parent's pointcut without knowing its exact type at implementation time. This improves flexibility in composing, reusing, and evolving aspects. Figure 6 shows how a pointcut triggers a corresponding advice (I). Refining that pointcut alters the triggering mechanism (II): the most refined pointcut triggers the connected advice, albeit the advice was defined in a parent aspect.

3.5 Advice Refinement

Before explaining advice refinement it is necessary to introduce the notion of *named advice*.

```
1  aspect Sync {
2    pointcut syncPC() : execution(* Fifo.get(..)) || execution(* Fifo.put(..));
3    Object around() : syncPC() { /* synchronization code */ }
4  }
5  refines aspect Sync {
6    pointcut syncPC() : super.syncPC() || execution(* Stack.*(..));
7  }
8  refines aspect Sync {
9    pointcut syncPC() : super.syncPC() && cflow(execution(* Socket.*(..)));
10 }
```

Fig. 5. Altering the set of locked methods via pointcut refinement.

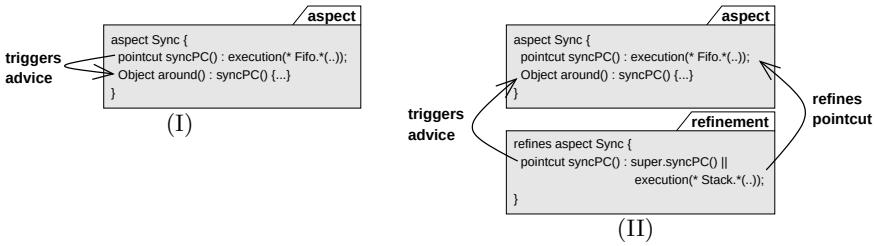


Fig. 6. The most refined pointcut triggers the connected advice.

Named advice. In order to refine advice in subsequent development stages, they must be named first-class entities. Unfortunately, advice in AspectJ-like languages are explicit but cannot be referred from other entities, e.g. other advice or methods, by name or reference. Rajan and Sullivan argue that this causes a fundamental asymmetry between OOP and AOP [17]. Although advice have a similar aim as methods, i.e. executing some functionality, they cannot be treated as methods. They are invoked implicitly when a pointcut matches.

In order to overcome this tension we propose *named advice*. Named advice are named first-class entities of aspects. They can be overridden and referred to from a child advice in order to refine its functionality. This enables the programmer to reuse and evolve advice over several development stages. Having named advice all structural elements of aspects can be refined in the same way (by using the *super* keyword).

Figure 7 depicts and synchronization aspect that contains a named advice (Lines 3-7). It consists of a name (*syncMethod*) and a binding to a pointcut (*syncPC*). We call this *advice binding* (Line 3). Additionally, for every named advice an associated method must be defined in the same aspect – called *advice method* (Line 4-7). Consequently, the advice method has an equal name and signature as the named advice.

In case of an around advice the corresponding method may use the *super* keyword to call the advised join point, e.g. invoking an advised method call (Line 5).[4] This is different from AspectJ that uses the *proceed* keyword. We de-

[4] An advised join point is accessed via *super* followed by the name of the advice.

```
1 aspect Sync {
2   pointcut syncPC() : execution(* Fifo.*(..));
3   Object around syncMethod() : syncPC();
4   Object syncMethod() {
5     lock(); Object o = super.syncMethod(); unlock();
6     return o;
7   }
8 }
```

Fig. 7. The notion of *named advice*.

cided for *super* to have a unique mechanism for referring to the refined methods, pointcuts, and advice. This also includes the advised join points.

In advice methods the full runtime API of AspectJ is available, e.g. *thisJoinPoint*. This can be achieved by passing information transparently from named advice to advice method. In the rest of the paper the meaning of named advice also includes the associated advice methods.

Refining named advice. By introducing named advice to AspectJ we can refine advice of parent aspects. Figure 8 depicts an aspect that refines our synchronization aspect shown in Figure 7 by extending its named advice, i.e. its advice method *syncMethod* is overridden and the number of threads is counted. Within the refining aspect the advice is handled as an advice method (Lines 2-5) and is extended accordingly using the *super* keyword (Line 3). Notice that always the most refined advice method is invoked by an bound pointcut.

```
1 refines aspect Sync {
2   Object syncMethod() {
3     threads++; Object o = super.syncMethod(); threads--;
4     return o;
5   }
6 }
```

Fig. 8. Refining named advice.

Refining advice methods that refer to around advice yields some interesting issues regarding the meaning of *super*. In a base aspect – where a named around advice is introduced for the first time – the *super* call inside an advice method means accessing the advised join point. In an refined advice method calling *super* means calling the parent advice method. Figure 9(I) shows a named around advice, i.e. the advice methods as well as to pieces of refining advice that use this calling pattern. Each advice method calls its parent method except those methods that have no parent; those methods call the advised join point directly. We call this calling pattern *super-call-pattern*. It is similar to the *CLOS* function *call_next_method* that iterates over all methods of the refinement chain [27].

During our investigations in named advice we explored a different alternative calling pattern. This alternative pattern keeps the *proceed* keyword to access the advised join point. Therefore, we call it *proceed-call-pattern*. Refinements of

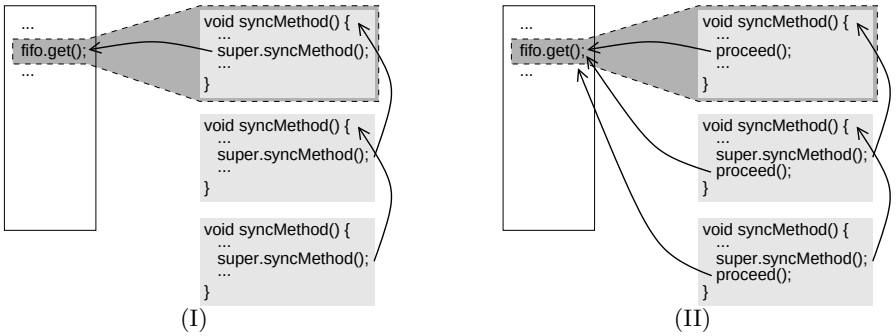


Fig. 9. *super*-call-pattern vs. *proceed*-call-pattern.

advice methods may either use *super* to call the parent advice method or use *proceed* to directly access the advised join point (see Fig. 9(II)). This allows every advice method of a refinement chain to access the advised join point. We do not prefer this alternative pattern because it introduces more complexity due to two different call mechanisms, *proceed* and *super*. Moreover, the possibility to access ancestors of a refinement chain that are different from the direct parents violates the principle of step-wise refinement. In order to keep our language extension as simple as possible, we prefer the *super*-call-pattern.

4 Bounding the Quantification of Aspects

In order to decrease the unpredictability of aspects as well as to increase the aspect's reusability, Lopez-Herrejon and Batory propose a novel approach to aspect composition [7]. They model aspects as functions that operate on programs. Applying several aspects to a program is modeled as function composition. In this way the scope of aspects is restricted to a particular stage in a program's development. Such *bounded quantification* of aspects follows an important principle of incremental software development: Layers (in our case aspects) should only affect and interfere layers of previous development stages. It has been argued that current AOP languages do not respect this principle because it is not possible to distinguish between different development stages [7].

With aspect refinement and mixin-based aspect inheritance it is possible for the first time to implement such bounded quantification: Since we know for each aspect and for each refinement to which development stage it belongs we can determine to which program elements the are permitted to bind.

However, our model differs from the one proposed Lopez-Herrejon and Batory. They assume that aspects that implement different features are woven immediately to a program. That is a straightforward way for incrementally refining a program via aspects. In our approach we propose something additional (we also support their approach): We do not only *add* aspects incrementally, but also *refine* aspects incrementally. That means we use (mixin-based) inheritance

to extend existing aspects. Whereas the former approach weaves the aspects immediately, ours weaves the most refined version only (the final compound aspect). Their approach is similar to adding classes incrementally; ours is analogous to refining classes incrementally via inheritance.

Without aspect refinement, bounding aspects is straightforward: aspects are woven exclusively to those program parts that where implemented in previous development stages. In our extended approach things are different. The overall refinement chain, e.g. all *Sync* aspects, forms a final aspect that is applied to the composed program. In fact, the refinement chain reflects but also hides the evolution of the overall aspect. We argue that at each development stage the current aspect (the leaf of the chain) should only affect code of previous stages. That means that not only each aspect but also each of its constituting evolutionary steps (its refinements) should only affect those program parts that are already known at their implementation time. This results in a final compound aspect that contains the overall functionality, but without allowing its fragments (the particular refinements) affecting subsequent added code.

In [14] we show how such bounded quantification – implemented in FeatureC++[5] a feature-oriented and aspect-oriented language extension to C++ – improves the reusability of aspects as well as their refinements and decreases their unpredictable behavior. For implementing our bounding mechanism we restructure the pointcut expressions of the aspects and their refinements: Restructured pointcuts match only a constrained set of join points, in particular all desired join points except those that originate from subsequent development stages. However, this study is only a first step because we considered only a limited set of pointcuts. Therefore, we omit a further discussion of implementing this bounding mechanism in this paper. A more detailed analysis for restructuring rules and their impact is part of further work.

5 Syntax and Semantics

In order to define our language extension in an unambiguous and consistent way we give a specification of its syntax and semantics.

5.1 Grammar of an AspectJ Extension

Figure 10 lists our grammar rules that extend the native AspectJ LALR(1) grammar described in [28]. Those rules that are highlighted by an underline are novel; all others are references to native AspectJ.

Rule 1 introduces the *refines* keyword as prefix to aspect declarations. Thus, it refers to the native AspectJ rule for declaring aspects ($\langle aspect\_declaration \rangle$). Rule 2 and 3 extend the AspectJ's pointcut mechanism with the ability to access the parent pointcuts via the *super* keyword. Rule 2 is an extended version of its AspectJ counterpart that combines common unary pointcuts with *super* calls

[5] wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

$$\underline{\langle aspect\_ref\_declaration\rangle} ::= \text{`}\underline{\texttt{refines}}\text{`}\ \langle aspect\_declaration\rangle \quad (1)$$

$$\begin{aligned}\langle unary\_pointcut\_expr\rangle ::= {} & \langle basic\_pointcut\_expr\rangle \\ | \ & \text{`!`}\ \langle unary\_pointcut\_expr\rangle \\ | \ & \underline{\langle super\_pointcut\_opt\rangle}\end{aligned} \quad (2)$$

$$\underline{\langle super\_pointcut\_opt\rangle} ::= \text{`}\underline{\texttt{super}}\text{.`}\underline{\textsc{Identifier}}\ \text{`(`}\langle formal\_param\_list\_opt\rangle\text{`)`} \quad (3)$$

$$\begin{aligned}\langle advice\_declaration\rangle ::= {} & \langle modifiers\_opt\rangle\langle advice\_spec\rangle\langle throws\_opt\rangle\ \text{`:`} \\ & \langle pointcut\_expr\rangle\langle method\_body\rangle \\ | \ & \langle modifiers\_opt\rangle\underline{\langle named\_advice\_spec\rangle}\langle throws\_opt\rangle\ \text{`:`} \\ & \langle pointcut\_expr\rangle\text{`;`}\end{aligned} \quad (4)$$

$$\begin{aligned}\underline{\langle named\_advice\_spec\rangle} ::= {} & \text{`}\texttt{before}\text{`}\ \underline{\textsc{Identifier}}\text{`(`}\langle formal\_param\_list\_opt\rangle\text{`)`} \\ & \ldots\end{aligned} \quad (5)$$

Fig. 10. An LALR(1) grammar for mixin-based aspect inheritance.

($\langle super\_pointcut\_opt\rangle$). Rule 3 defines the syntax of such *super* calls. Rule 4 and 5 introduce named advice. In order to be downward compatible to native AspectJ, Rule 4 accepts besides the novel named advice also the common unnamed advice. Rule 5 defines the syntax of named advice. It is assumed that for each named advice an associated advice method is defined. Furthermore, the named advice and the advice method must have the same signature.

5.2 Formal Semantics

We use a set of rewriting / reduction rules in order to define the formal semantics of our AspectJ language extension. As a basis, we use a simplified subset of the AspectJ grammar (adopted from [29]) because the original grammar including our extension would be too complex. In order to define the semantics we introduce a set of reduction rules that extend the semantics of AspectJ [29]. These reduction rules are formulated using abstract syntax and reduce our added constructs to native constructs described in the AspectJ's operational semantics.

Simplified abstract syntax. We introduce a simplified and trimmed-down set of abstract grammar rules in order to provide a basis for a formal definition of the semantics of our language extension. Figure 11 shows our rules that build up on the grammar specification given in [29, 30]. The new rules concern aspects, pointcuts, and named advice. We use the following conventions for metavariables used in the rules: Variables written in capital letters represent the main language constructs. They are the non-terminal symbols of our grammar and range over particular kinds of declarations. Metavariables in lower case letters are placeholders for program variable names. Keywords (terminal symbols) are highlighted in bold letters.

The metavariables used in the grammar are reserved to their particular types, e.g. we use $c_1, \ldots, c_n$ for referring to class names or S to range over statements. $\vec{X}$ means an ordered set and $\bar{X}$ an unordered set of elements. Furthermore, we use x and y for referring to program variables as well as $\vec{x}$ and $\vec{y}$ for a sequence of variables, e.g. for expressing an argument list.

Simple classes (C) have a name (c_1), a parent (c_2), and consist of a set of methods ($\bar{M}$)[6]. Inheritance is expressed using <:. The expression $c_1 <: c_2$ means that class c_1 inherits from class c_2. Methods (M) have a name, a results type (c_r), a set of arguments ($\vec{c}\ \vec{x}$), and execute a set of statements ($\vec{S}$). $\vec{c}\ \vec{x}$ means a sequence of arguments with its particular types: $c_1\ x_1, \ldots, c_n\ x_n$. Statements ($S$) are not further explained. We assume a standard set, e.g. *new*, *return*, etc. (see [29] for a comprehensive list). Aspects (A) contain methods ($\bar{M}$), pointcuts ($\bar{P}$), and advice ($\bar{Z}$). Besides aspects also aspect refinements can be declared (R). Refining aspects cannot inherit from other aspects. The parent is determined by the order of the refinement chain. Pointcuts (P) have a name (p), a set of arguments ($\vec{c}\ \vec{x}$), and a set of pointcut expressions (Φ)[7]. In contrast to AspectJ we add the possibility to access the parent pointcuts via *super*. For the sake of simplicity, we consider only disjunctions.[8] Advice (Z) have either names (z) and do not own bodies or they are unnamed and have a body. Both types of advice expect a set of arguments ($\vec{c}\ \vec{x}$) and have a result type (c_r). For each named advice an advice method with the same name and signature must be defined. Unnamed advice are included to be compatible to native AspectJ. The *advice* keyword abstracts over all three possible advice types: *before*, *after*, and *around* (see [29]).

$Classes:$	$C ::= \textbf{class}\ c_1 <: c_2\ \{\bar{M}\}$
$Methods:$	$M ::= c_r\ m(\vec{c}\ \vec{x})\ \{\vec{S}\}$
$Statements:$	$S ::= \textbf{new} \mid \textbf{return} \mid \ldots$
$Aspects:$	$A ::= \textbf{aspect}\ a_1 <: a_2\ \{\bar{M}, \bar{P}, \bar{Z}\}$
$Refinements$	$R ::= \textbf{refines aspect}\ a\ \{\bar{M}, \bar{P}, \bar{Z}\}$
$Pointcuts:$	$P ::= \textbf{pointcut}\ p(\vec{c}\ \vec{x})\ \textbf{:}\ \Phi\ \vee\ \textbf{super::}q(\vec{c}\ \vec{y})$
$PointcutExpr.$	$\Phi ::= \textbf{false} \mid \neg\phi \mid \phi \vee \phi\prime \mid \textbf{call}(c :: m)$
$Advice:$	$Z ::= c_r\ \textbf{advice}\ z(\vec{c}\ \vec{x})\ \textbf{:}\ p \mid c_r\ \textbf{advice}(\vec{c}\ \vec{x})\ \textbf{:}\ p\ \{\vec{S}\}$

Fig. 11. Abstract syntax of the extended AspectJ grammar

Reduction rules. In order to describe the semantics precisely, we extend the operational semantics of AspectJ by a set of reduction rules, i.e. we add three new rules to the existing set that describe the reduction to native rules. In order to determine the behavior of an AspectJ program that includes our extensions one has to apply the reduction rules depicted in Figure 12. These can be reduced to native rules explained in [29].

[6] For brevity, we omit other elements as fields or nested classes

[7] For the time being, we consider only *call* pointcuts

[8] Conjunctions can be created using *DeMorgan* and the negation.

Rule R-1 reduces an aspect refinement to a native aspect a_1 that inherits from a parent aspect a_2. It is presumed that such parent aspect exists. In other words, an parent aspect with n refinements is translated to an aspect with n child aspects. The order of the inheritance chain is inferred from the order of the refinement chain.

Rule R-2 reduces a pointcut p, i.e. its pointcut expression ϕ and an additional *super* call to a parent pointcut q. As mentioned before we consider only disjunctions. The resulting pointcut accesses the parent pointcut using its fully qualified name $a_2{::}q$, with a_2 as parent aspect that contains the pointcut q. We omit the pointcut arguments because they remain unchanged. A precondition for such reduction is that a parent aspect a_2 contains an fitting pointcut q.

Rule R-3 defines the reduction of named advice to native advice. For each named advice it is presumed that an advice method exists with an equal name and the same signature. The named advice z is simply transformed into a native advice that calls the associated advice method m. It is presumed that the programmer has defined a method m with equal name and signature $\vec{c}\,\vec{x}$. In the resulting advice body this method is called and the results are returned. This notation follows the grammar specification of *Featherweight Java* [30] where method bodies are defined as return value of an expression. We omit a detailed specification of the passed parameters because these depend on the use in the advice body, e.g. if one does not use the *thisJoinPoint* object we do no need to pass it to the advice body.

$$\frac{\textbf{aspect}\ a_1\ \{\ldots\}}{\textbf{refines aspect}\ a_1\ \{\ldots\}\ \rightarrow\ \textbf{aspect}\ a_2 <: a_1\ \{\ldots\}} \tag{R-1}$$

$$\frac{\begin{array}{c}\textbf{aspect}\ a_2\ \{\ldots q\ldots\}\\ \textbf{aspect}\ a_1 <: a_2\ \{\ldots p\ldots\}\end{array}}{\textbf{pointcut}\ a_1{::}p : \phi\ \vee\ \textbf{super}{::}q\ \rightarrow\ \textbf{pointcut}\ a_1{::}p : \phi\ \vee\ a_2{::}q} \tag{R-2}$$

$$\frac{c_r\ m(\vec{c}\ \vec{x})\ \{\ldots\} \qquad z = m}{c_r\ \textbf{advice}\ z(\vec{c}\ \vec{x}) : p(\vec{c}\ \vec{x})\ \rightarrow\ c_r\ \textbf{advice}(\vec{c}\ \vec{x}) : p(\vec{c}\ \vec{x})\ \{\textbf{return}\ m(\vec{x})\}} \tag{R-3}$$

Fig. 12. Reduction rules.

6 Extending AspectJ

In order to prove the applicability of our approach we present a program transformation approach that translates code written in our extended AspectJ to native AspectJ code.

6.1 Mixin Transformation to native AspectJ

As our formal semantics description dictates, mixin-based aspect inheritance is transformed to common aspect inheritance. Each mixin is transformed to a common aspect that inherits from its parent based on the composition order. The order is inferred from the input of the aspect weaver. Furthermore, our semantics specification states that all *super* keywords found in pointcut expressions are translated to the corresponding types of the parent aspects. Named advice is translated into advice that calls the associated method that executes the advice code. Furthermore, all necessary parameters are passed to the method as well as the AspectJ runtime objects and further context information.

Figure 13 shows an example aspect (Lines 1-7) and a refinement (Lines 8-17) connected via mixin-based aspect inheritance. The base aspect contains two methods (Line 2,3), a pointcut (Line 4), and a named advice with advice method (Lines 5-6). The child aspect adds a field (Line 9) refines the parent methods (Lines 10,11), the parent pointcut (Line 12), and the parent advice (Line 13-16). In all three cases we use the *super* keyword to access the parent entities.

```
1  aspect Sync {
2    void lock() { /* lock access */ }
3    void unlock() { /* unlock access */ }
4    pointcut syncPC() : execution(* Fifo.*(..));
5    Object around syncMethod() : syncPC();
6    Object syncMethod() { /* synchronization code */ }
7  }
8  refines aspect Sync {
9    int threads;
10   void lock() { threads++; super.lock(); }
11   void unlock() { threads--; super.unlock(); }
12   pointcut syncPC() : super.syncPC() || execution(* Stack.*(..));
13   Object syncMethod() {
14     /* advice code */
15     return super.syncMethod();
16   }
17 }
```

Fig. 13. An aspect with refinement using mixin-based inheritance.

For simplicity we assume that aspects and their different refinements are located in different file system directories. Each directory reflects a particular development stage and contains the corresponding aspects (and classes). Assuming that the base aspect is located in the directory *Base* and the refinement in the directory *Ext*, Figure 14 shows both aspects transformed into native AspectJ aspects. The transformation consists of the following steps:

1. The synchronization aspect and its refinement are transformed into two native aspects that contain the name of the source directory in their names in order to distinguish them (Line 1,10). Except the leaf of the refinement chain all aspects are declared as *abstract* – in our example only the base aspect.
2. The *lock* and *unlock* methods and their refinements remain unchanged (Lines 2,3 and 12,13).

```
1  abstract aspect Sync_Base {
2    void lock() { /* lock access */ }
3    void unlock() { /* unlock access */ }
4    pointcut syncPC() : execution(* Fifo.*(..));
5    Object around() : syncPC() {
6      return syncMethod(thisJoinPoint);
7    }
8    Object syncMethod(JoinPoint thisJoinPoint) { /* synchronization code */ }
9  }
10 aspect Sync_Ext extends Sync_Base {
11   int threads;
12   void lock() { threads++; super.lock(); }
13   void unlock() { threads--; super.unlock(); }
14   pointcut syncPC() : Sync_Base.syncPC() || execution(* Stack.*(..));
15   Object syncMethod(JoinPoint thisJoinPoint) {
16     /* advice code */
17     return super.syncMethod(thisJoinPoint);
18   }
19 }
```

Fig. 14. Translating mixin-based inheritance to common inheritance.

3. The *syncPC* pointcut of the base aspect is moved one-to-one to its native counterpart (Line 4). The *syncPC* pointcut of the refining aspect is tranformed to a AspectJ compatible version (Line 14): the *super* keyword is replaced by the name of the parent aspect, in our case *Sync_Base*
4. The named advice of the base aspect and its connected method are transformed to a native advice (Lines 5-7) and a simple method (Line 8). The advice calls this method and passes necessary runtime objects to its scope (Line 6). The signature of the method and the passed arguments depend on the signature of the pointcut, e.g. which variables are needed inside the advice, and the needed runtime objects, e.g. *thisJoinPoint*. In our example only join point information is passed.
5. The refined advice method *syncMethod* (Lines 15-18) gets an extended signature that allows to pass the runtime objects as well as pointcut-specific arguments, e.g. target objects, to the advice code.

6.2 An Alternative Transformation – JamPack Composition

A drawback of the above explained transformation of a refinement chain into an inheritance hierarchy are the performance penalties and the overhead regarding the memory consumption. An aspect with n refinements results in $n + 1$ native aspects that form the resulting inheritance chain. Even if the refinements implement only a very small delta to the existing functionality, for each delta a new aspect is introduced. In the worst case, refining a method or advice results in n calls upward along the inheritance chain.

To eliminate such penalties, Batory et al. propose an alternative composition mechanism for mixin classes, called *JamPack* composition [5]. Mapping it to aspects means that an aspect and all of its refinements are merged into *one* final aspect at source code level. We call such aspect *flattened*; Jampack composition is different from mixin composition where a inheritance chain of aspects

is generated. However, in both case only one final aspect instance is woven to the target program.

JamPack composition can be implemented by merging pointcuts using logical operators and by concatenating and wrapping the code of methods and advice. This composition mechanism would yield better performance and memory consumption characteristics compared to our proposal. However, the JamPack composition makes it hard to debug the generated code because the origin of code fragment is not traceable. We argue that for the development phase our proposed composition (mixin-based inheritance into inheritance) is more intuitive and helps the programmer to understand the effects of refinements. For producing a release version the JamPack composition approach would be appropriate, especially for resource-constrained systems.

6.3 Implementation

Currently, we have implemented our ideas on aspect refinement and mixin-based inheritance (except named advice) in FeatureC++ [12, 13], an feature-oriented and aspect-oriented language extension to C++. In this paper we focused on AspectJ because it is better known in the community and it is a more mature aspect language. We already extended the *abc* – an extensible AspectJ compiler [31] – with mixin-based aspect inheritance, i.e. the possibility to implement refinements to aspects. Named advice as well as pointcut refinement are the next steps to be implemented.

7 Related Work

Higher-order pointcuts and advice. Our notion of aspect refinement is related to higher-order pointcuts and advice proposed by Tucker and Krishnamurthi [32]. They integrate advice and pointcuts into languages with higher-order functions and model them as first-class entities. Pointcuts can be passed to other pointcuts as arguments, and therewith modified, combined, and extended. In this point our approach of aspect and pointcut refinement is similar. We can combine, modify, and extend pointcuts by adding subsequent refinements.

Due to the opportunity to refine named advice we can also modify and extend advice using subsequent advice. This corresponds to higher-order advice that expect advice as input and return a modified advice. Our named advice can be passed to other advice – usually to the child advice that refines the parent (input) advice. Thus, refining a named advice is like passing an advice to a higher-order advice.

Aspect refinement and AHEAD. We originally proposed the notion of aspect refinement [14] in the context of feature-oriented program families and *AHEAD* – an architectural model for large-scale program composition [5]. The AHEAD model proposes to perceive software as a collection of features that satisfies the requirements of stakeholders. Features do not only consist of source

code but of all artifacts that contribute to the feature, e.g. documentation, test cases, design documents, makefiles, etc. Each feature is represented by a *containment hierarchy*, a directory that maintains a subdirectory structure to organize its artifacts. Composing features means composing containment hierarchies and to its end composing corresponding artifacts, e.g. composing two program fragments. Hence, for each artifact type a distinct *composition operator* (denoted by +) is provided.

A exceptional quality of the AHEAD model is that features and their compositions are described by algebraic equations, e.g. *Prog = Buffer + Sync + Log*. This declarative style allows for compositional reasoning and algebraic equation optimization [5].

In context of the AHEAD model, mixin-based aspect inheritance is simply a composition operator that is invoked when aspects (and their refinements) of different development stages are composed. Hence, it corresponds to the class composition operator that composes classes using common mixin-based inheritance. Since our aspects and their refinements can be composed externally without modifying code[9], they best fit the AHEAD approach of algebraic equation-based composition.

In order to integrate aspects and features in the sense of the AHEAD model, we proposed *aspectual mixin layers (AMLs)* [13, 12]. The synergetic effects of aspects, features, and incremental design methodology are an improvement over common incremental designs based on classes and common AOP, e.g. the crosscutting modularity is improved [13]. Aspect refinement based on mixin-based aspect inheritance is a key technology to implement and improve AMLs and integrate features and aspects.

Aspect refinement and Caesar. Caesar supports componentization of aspects by encapsulating virtual classes as well as pointcuts and advice in collaborations. These are hidden behind *aspect collaboration interfaces* that decouple an aspect's implementation from its binding to a target program. Bindings themselves are collaborations that adapt the aspect's implementation to a application context. This on-demand remodularization (*aspectual polymorphism*) 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. This is implemented using mixin composition. Besides this, collaborations can be refined using pointcuts. Due to its embedding in classes (*family classes*), collaborations of virtual classes can be used polymorphically [33].

Although aspect refinement and mixin-based aspect inheritance are not directly related to Caesar, integrated in a component technique as AMLs it become arguably interesting to compare them to Caesar. Both, Caesar and AMLs are based on collaborations which represent the basic building blocks and both integrate AOP concepts. A main advantage of AMLs is that they have AHEAD as an architectural model; Caesar makes no statement about such a model. Hence, AMLs can revert to several advantages of AHEAD: beside classes and

[9] For example, mixins implemented in C++ have to be instantiated inside a program.

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.

A main difference of AMLs and Caesar is that with AMLs aspects are integrated into collaborations whereas with Caesar the collaborations themselves are understood as aspects (or aspect components that contain pointcuts and advice). With Caesar it is not possible to refine these pointcuts and advice. Thus, higher-order aspects as proposed in this paper are not supported. 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. However, we argue that the ideas of refining pointcuts and named advice as well as a bounded quantification can be integrated into Caesar in order to evolve aspects over time in a reliable and consistent way.

We believe the difference of AMLs and Caesar originates from their different focuses. The philosophy of AMLs is to incrementally build product lines by layering collaborations of aspects and classes. Aspects affect mainly code inside a stack of collaborations in order to implement (1) homogeneous crosscutting concerns that extend a parent feature at different points with the same new functionality, (2) features that highly depend on the runtime control flow (dynamic crosscutting), and (3) features with a structure that is different from the structure of the parent feature. In all three cases aspects and their composition and modularization mechanisms perform better than traditional OOP mechanisms, i.e. mixin-based inheritance. AMLs do not focus on adapting (a stack of) collaborations to a particular application context and do not support polymorphic collaborations, that are design goals of Caesar.

Caesar proposes also a layer structure to incrementally develop aspect components. But it aims at – besides others – on-demand remodularization of these layered aspects in order to prepare them for weaving to external code. Caesar's collaborations (aspect components) are first-class and their composition is done within source code. This prevents compositional reasoning and an optimization based on algebraic equations.

Unifying advice and methods. Rajan and Sullivan propose *classpects* that combine capabilities of aspects and classes to unify the design of layered module systems [17]. A classpect simply associates for each advice a method that is executed for advising a particular join point. Moreover, classpects unify aspects and classes with respect to instantiation which is not addressed by our approach.

A second approach for unifying methods and advice is proposed by Lopez-Herrejon et al. [7]. Although they do not aim at a unification at language level they propose an interesting mechanism, called *pure advice*. Pure advice is a named advice that replaces the advice body with a call to a method that is introduced to a target class. It simply introduces a name to advice and further separates the advice declaration of its definition. Mapping this methodology

to our approach an inheriting child aspect may extend a parent advice by (1) overriding the named parent advice or (2) by overriding the associated method.

To our opinion pure advice are less elegant compared to the classpects advice because there are two associated names: the advice name and the called method name. This make the code unnecessary complex and leads to the question which of both to override. Therefore, our approach of named advice is mostly influenced by classpects.

Generic aspects. Several recent approaches enhance aspects with genericity, e.g. *Sally* [34], *Generic Advice* [35], *LogicAJ* [36], *Framed Aspects* [37]. This improves reusability of aspects in different application contexts. Aspect refinement and mixin-based aspect inheritance provides a alternative way to customize aspects, i.e. by composing the required refinements. However, ideas on generic aspects can combined with our approach.

AspectJ design patterns. Hanenberg and Unland discuss the benefits of inheritance in the context of AOP [38, 26]. They argue that aspect inheritance improves aspect reuse. To underpin their claims they propose different design patterns that exploit structural elements specific to AspectJ. Their patterns *pointcut method*, *composite pointcut*, and *chained advice* suggest to refine pointcuts in subsequent development steps to improve customizability, reusability and extensibility. Due to its flexibility mixin-based inheritance can enhance these patterns by simplifying the composition of aspects. The pattern *template advice* can be simplified using named advice because it becomes possible to directly refine advice. Mixin-based aspect inheritance is the logical next step towards compositional inheritance for aspects [15].

AOP and modular reasoning. A couple of work tries to tackle the problem of aspects and their unpredictable behavior from another side. *Aspect-ware interfaces* [10] and *open modules* [11] support modular reasoning by encapsulating explicitly the interactions between two concerns. This reduces unpredictable aspect/feature interactions but also reduces the flexibility to implement unanticipated features. Our approach is different because we aim at incremental designs that allows us to assign aspects to development stages. Hence, a bounded quantification can exploit the implicit knowledge of the evolutionary design. This avoids a lot of explicit specifications and formulated constraints to be provided by the programmer.

8 Conclusion

The introduction of mixin-based inheritance to AOP yields several benefits. Firstly, it unifies classes and aspects with regard to incremental software development. The composition of classes and aspects takes place at compile-time.

Mixin-based aspect inheritance increases the ability to flexibly alter the refinement chain of aspects and therefore offers a way to customize aspects. Hence, it is an improvement of simple aspect inheritance. Mixin-based aspect inheritance

is an elegant way to implement aspect refinement in incremental designs. Moreover, our AspectJ-based implementation is an AHEAD composition operator for aspects. Aspect refinements can be seen as higher-order aspects and may also improve Caesar's abilities to refine and evolve aspects in a controlled mannor.

The introduction of mixin-based inheritance leads to the unification of methods and advice. This means that all structural elements of aspects can be subject of subsequent refinements, including pointcuts and named advice. All parent entities can be accessed uniformly via the *super* keyword. By allowing concrete aspects to be refined the programmer does not need to anticipate subsequent refinements. We believe that this work can help to understand and experiment with aspects to implement incremental designs.

A further benefit of composing aspects based on their affiliation to a development stage and a corresponding layer is that the quantification of aspects can be bound in order to decrease the unpredictability of aspects. This enhances the capabilities of aspect to be used in incremental designs.

However, the notion of aspect refinement and its implementation using mixin-based aspect inheritance has some limitations and drawbacks. Aspect refinement and mixin-based inheritance seems to be useful only in incremental designs. That means the programmer and the aspect weaver have to know the affiliation of artifacts to development stages. We believe that incremental software development is fundamental and will gain momentum in the future. Furthermore, we want to emphasize that our mechanisms are most effective in large software systems that demand for compositional reasoning. Besides this general limitation also some open issue remain:

- How pointcuts and named advice fit polymorphism and virtuality?
- Currently, we left out static crosscutting as intertype declarations.
- We do not addressed access modifiers, e.g. *public* or *private*, for named advice, advice methods, and pointcuts.
- An important point is to explore useful design patterns of mixin-based aspect inheritance.
- A further interesting topic is to explore the deeper relationship of higher-order aspects and the algebraic description of aspect refinement.

In future work we intend to integrate an extended AspectJ into the AHEAD Tool Suite. In doing that a programmer may implement and refine program families using classes and aspects, both integrated in features. Such features would be an instance of our proposal of AMLs – with Java and AspectJ as base languages. In this context we also plan to implement the discussed bounded quantification that bounds aspects based on their affiliation to development stages. Furthermore, we want to extend our preliminary analysis of pointcut restructuring.

Acknowledgments. We thank Don Batory and Roberto Lopez-Herrejonn for their comments on drafts of this paper. This research is sponsored in parts by the German Research Foundation (DFG), project number SA 465/31-1.

References

1. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of European Conference on Object-Oriented Programming. (1997)
2. Wirth, N.: Program Development by Stepwise Refinement. Communications of the ACM **14** (1971)
3. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
4. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering **SE-5** (1979)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering **30** (2004)
6. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proceedings of International Symposium on Foundations of Software Engineering. (2004)
7. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A Disciplined Approach to Aspect Composition. In: Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. (2006)
8. McEachen, N., Alexander, R.T.: Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2005)
9. Gybels, K., Brichau, J.: Arranging Language Features for more robust Pattern-based Crosscuts. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
10. Kiczales, G., Mezini, M.: Aspect-Oriented Programming and Modular Reasoning. In: Proceedings of International Conference on Software Engineering. (2005)
11. Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proceedings of European Conference on Object-Oriented Programming. (2005)
12. Apel, S., et al.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proceedings of Generative Programming and Component Engineering. (2005)
13. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: Proceedings of International Conference on Software Engineering. (2006)
14. Apel, S., Leich, T., Saake, G.: Aspect Refinement and Bounding Quantification in Incremental Designs. In: Proceedings of Asia-Pacific Software Engineering Conference. (2005)
15. Bracha, G., Cook, W.: Mixin-Based Inheritance. In: Proceedings of European Conference on Object-Oriented Programming and International Conference on Object-Oriented Programming Systems, Languages and Applications. (1990)
16. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
17. Rajan, H., Sullivan, K.J.: Classpects: Unifying Aspect- and Object-Oriented Language Design. In: Proceedings of International Conference on Software Engineering. (2005)
18. Taivalsaari, A.: On the Notion of Inheritance. ACM Computing Surveys **28** (1996)
19. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Proceedings of European Conference on Object-Oriented Programming. (2005)
20. Lieberherr, K.: Controlling the Complexity of Software Designs. In: Proceedings of International Conference on Software Engineering. (2004)

21. Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multi-Dimensional Separation of Concerns. In: Proceedings of International Symposium on Foundations of Software Engineering. (2003)
22. Apel, S., Böhm, K.: Towards the Development of Ubiquitous Middleware Product Lines. In: ASE Workshop on Software Engineering and Middleware. (2005)
23. Cardone, R., et al.: Using Mixins to Build Flexible Widgets. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2002)
24. Batory, D., Thomas, J.: P2: A Lightweight DBMS Generator. Journal of Intelligent Information Systems **9** (1997)
25. Batory, D., et al.: Creating Reference Architectures: An Example from Avionics. In: Proceedings of Symposium on Software Reusability. (1995)
26. Hanenberg, S., Schmidmeier, A.: Idioms for Building Software Frameworks in AspectJ. In: AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. (2003)
27. DeMichiel, G., Gabriel, R.P.: The Common Lisp Object System: An Overview. In: Proceedings of European Conference on Object-Oriented Programming. (1987)
28. Hendren, L., et al.: The abc Scanner and Parser, Including an LALR(1) Grammar for AspectJ. Programming Tools Group, Oxford University. (2004) `http://abc.comlab.ox.ac.uk/documents/scanparse/`.
29. Jagadeesan, R., Jeffrey, A., Riely, J.: A Calculus of Untyped Aspect-Oriented Programs. In: Proceedings of European Conference on Object-Oriented Programming. (2003)
30. Igarashi, A., Pierce, B.C., Wadler, P.: Featherwieght Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23** (2001)
31. Avgustinov, P., et al.: abc: An Extensible AspectJ Compiler. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2005)
32. Tucker, D., Krishnamurthi, S.: Pointcuts and Advice in Higher-Order Languages. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
33. Ernst, E.: Family Polymorphism. In: Proceedings of European Conference on Object-Oriented Programming. (2001)
34. Hanenberg, S., Unland, R.: Parametric Introductions. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
35. Lohmann, D., Blaschke, G., Spinczyk, O.: Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In: Proceedings of Generative Programming and Component Engineering. (2004)
36. Kniesel, G., Rho, T., Hanenberg, S.: Evolvable Pattern Implementations Need Generic Aspects. In: Proceedings of ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution. (2004)
37. Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. In: Proceedings of International Conference on Software Reuse. (2004)
38. Hanenberg, S., Unland, R.: Using and Reusing Aspects in AspectJ. In: OOPSLA Workshop on Advanced Separation of Concerns in OO Systems. (2001)