Aspect Refinement

Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake

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

Abstract. *Stepwise refinement (SWR)* is fundamental to software engineering. As *aspect-oriented programming (AOP)* gains momentum in software development, aspects should be subject to SWR as well. In this paper, we introduce the notion of *aspect refinement* that unifies AOP and SWR. To reflect this unification to programming language level, we present an implementation technique for refining aspects based on mixin composition. Specifically, we propose a set of concrete mechanisms for refining all kinds of structural elements of aspects in a uniform way (methods, pointcuts, advice). To underpin our proposal, we contribute a formal syntax and semantics specification as well as a fully functional compiler on top of AspectJ. We apply our approach to a non-trivial case study and derive several programming guidelines.

1 Introduction

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

AOP pervades more and more phases and parts of software engineering. This paper relates AOP to *stepwise refinement (SWR)*, a fundamental approach to software development [2–5]. By adding of new program details in a stepwise manner, the programmer breaks down complex software into manageable pieces (*modules*). This *incremental process* results in conceptually *layered designs*. The increments are called *refinements*. They are implemented in distinct steps during the development and evolution of software. This methodology is supposed to promote reuse, customization, and maintenance of software artifacts [2, 4, 5]. Since most software is developed and evolved in a more or less incremental process [6–8], it is desirable that modern programming paradigms reflect this by explicit support of SWR at language level.

We argue that due to its significance, we should take the methodology of SWR into account when designing AOP languages. Therefore, we propose the notion of *aspect refinement*. Aspect refinement is the consequent application of SWR principles to AOP. It is a design methodology to incrementally develop and evolve aspects in layered architectures by means of SWR to improve aspect reuse and customization.

In order to implement aspect refinement, we introduce the notion of *mixin-based inheritance* [9] to AOP. *Mixin-based aspect inheritance* explicitly supports SWR at language level by introducing mixin capabilities to aspects. Though most aspect languages support a limited form of aspect inheritance, mixin-based aspect inheritance enables the programmer to flexibly compose aspects and their refinements. Thereby the programmer alters the refinement chain that evolves over several development steps. Mixin-based aspect inheritance provides the required flexibility to compose, reuse, and customize aspects for developing and evolving highly customizable, layered designs, e.g., product lines. Although we aim at extending *AspectJ*[1], our results are applicable to other AOP languages.

Furthermore, the notions of aspect refinement and mixin-based aspect inheritance unify aspects and classes with respect to SWR. We propose a general approach for refining all kinds of structural elements of aspects. Specifically, we present several concrete language-level mechanisms that implement this approach for the particular kinds of elements, i.e., *pointcut refinement*, *named advice*, and *advice refinement*.

We demonstrate the practical applicability of our language proposal by providing a formal syntax and semantics specification as well as a fully functional compiler. We use this compiler to apply aspect refinement to a non-trivial case study.

In this paper we make the following contributions:

- We introduce the notion of aspect refinement that unifies AOP and SWR as well as mixin-based inheritance for reflecting that unification to language level.
- We present concrete language mechanisms for refining aspects, pointcuts, and advice.
- We provide a formal specification of the syntax and the semantics.
- We contribute a fully functional compiler that supports all proposed extensions to AspectJ.
- We discuss the key results of applying aspect refinement to a non-trivial case study.

2 Aspect Refinement

This section briefly discusses some selected problems of aspects when integrated into layered designs and SWR. Afterwards, we describe the notion of aspect refinement and how it approaches these problems.

2.1 Problems of Aspects in SWR

Aspect inheritance. Inheritance is known as a concept for reusing and non-invasively refining software artifacts [10]. Therefore, most AOP languages support aspect inheritance. Although this facilitates aspect refinement to some de-

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

gree, it lacks flexibility to interchange and reuse refinements. Using aspect inheritance, a refinement (a subaspect) is fixed to a specific parent aspect. Hence, refinements cannot be reused with other aspects nor flexibly combined in different orderings for customization purposes. Mixin-based inheritance overcomes this limitation by moving the selection of the parent to composition time. This increases reusability in different contexts, allows for SWR, and improves customizability by plugging a whole bunch of tailored refinements to an aspect.

Constrained aspect extension. Using traditional aspect inheritance in AspectJ an aspect has to be declared as *abstract* to be able to be refined. This means that adding a subaspect requires the programmer to modify the parent aspect. This and similar requirements[2] cause a fundamental problem of AspectJ-like languages with regard to SWR. Implementing an aspect in a particular development step forces the programmer to decide whether the aspect would be refined in a later step. 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 concrete 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.

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. They are invoked implicitly when associated pointcuts match. This prevents other advice or methods from invoking them explicitly. Thus, advice is the only unnamed structural element of aspects. This complicates SWR using aspects and hinders reuse and customization of advice code. We propose to refine all structural elements of aspects uniformly. In order to facilitate SWR, advice have to be referable from subsequent refinements.

2.2 SWR of Aspects

Aspect refinement is the incarnation of SWR in the AOP world. The key idea is to refine aspects incrementally. Thus, aspects – as with all other software artifacts – are developed and evolve over time. In each development step aspects may be refined. Refinements reuse as much as possible functionality of previous steps. This view is consistent with the early work on hierarchical (module) system designs [2, 4]; it follows the *principle of uniformity* that proclaims that every kind of software artifact is refineable [11]. 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 new possibilities of aspect reuse and customization.

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

The idea of aspect refinement emerged from prior work on aspect-oriented and feature-oriented product lines and program synthesis [12]. In this work we now elaborate and generalize these ideas to all kinds of layered designs that follow SWR principles. The notion of aspect refinement does not depend on features, components, or product lines, but merely is a fundamental concept of SWR using AOP. Furthermore, we contribute a thorough implementation, formal underpinning, and practical application of aspect refinement particularly with regard to the broader perspective of SWR and layered designs.

2.3 A SWR Example Using Aspects

Figure 1 shows four steps in the evolution of a program developed using AOP. It implements several types of buffers and network sockets that support concurrent access.

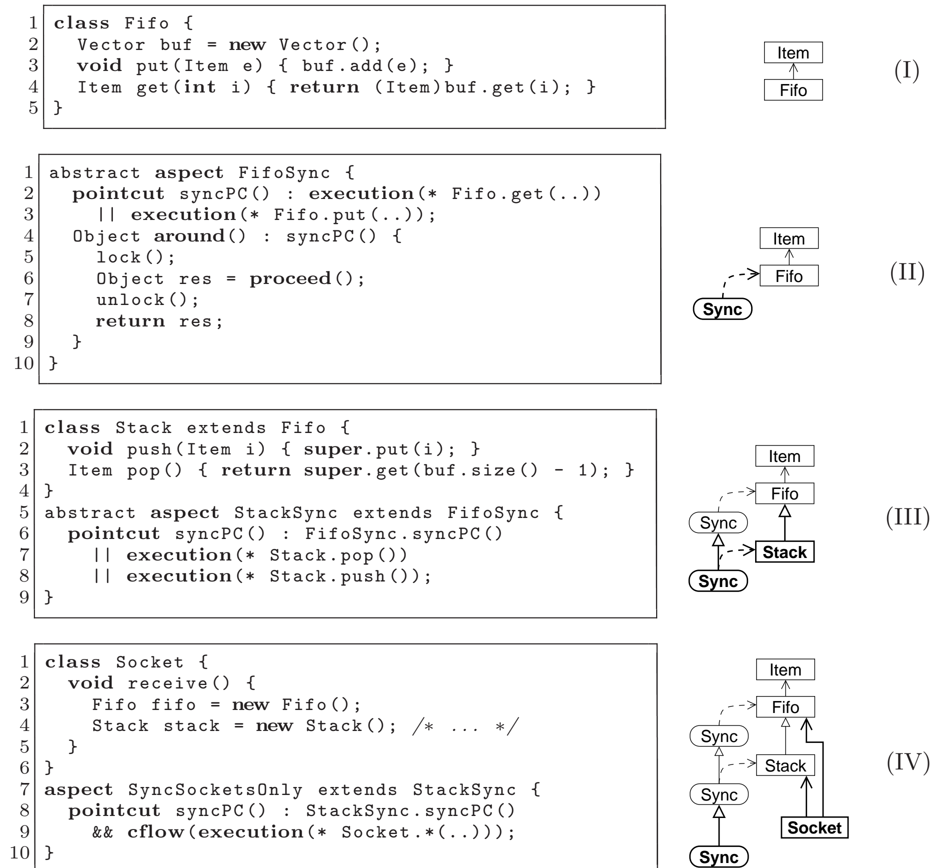

Fig. 1. A SWR example using aspects (AspectJ code).

The evolution spans 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 step are highlighted bold. Aspect weaving is displayed by dashed arrows.

I. In the initial class hierarchy a *Fifo* buffer stores a set of data items. For that, it provides a *put* and a *get* method.
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. Then, 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*). *StackSync* 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. *SyncSocketsOnly* limits the set of matched join points to those that are inside the control flow of *Socket*. This is achieved by overriding the parent pointcut to restrict the set of join points.

This example illustrates the usefulness of refining aspects in a step-wise manner over several development steps. Aspect refinement is a logical consequence of applying SWR to AOP. Goal of this layered design is to encapsulate the different program features. Consequently, we want to derive different customized program variants that share common features. For example one variant contains only a synchronized *Fifo* buffer (*FifoSync + Fifo*), another a fifo buffer that is synchronized only with respect to calls from *Socket* (*SyncSocketsOnly + FifoSync + Fifo*), or finally a variant that consists of all features (*SyncSocketsOnly + StackSync + Stack + FifoSync + Fifo*).

Our present design is not flexible enough for creating all these different variants without modifying the implementation. Using inheritance, refinements are fixed to their target aspects (super-aspect). We are not able to remove refinements or combine them in different orders. Furthermore, aspects that are supposed to be refined have to be declared as abstract; advice cannot be refined at all.

To overcome these limitations, we propose to introduce and employ alternative language level mechanisms that explicitly satisfy the requirements imposed by SWR.

3 Mixin-Based Aspect Inheritance

In order to increase the flexibility of composing refinements, we introduce mixin-based aspect inheritance. First, we review the original approach of mixins; then we show how it can be adopted for implementing refinements of aspects. Specifically, we explain the different mechanisms to refine aspects such as adding fields and methods, extending existing methods, refining pointcuts and advice.

3.1 Mixin-Based Inheritance and SWR

Refinements 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. *Mixin-based inheritance*, introduced by Bracha and Cook, is a flexible approach to implement incremental program refinements [9]. 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 is, they can be applied to different classes implementing *reusable* extensions and are therefore qualified to implement refinements. *Mixin composition* is also called *instantiation 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 four alternative refinement chains generated out of a set of mixins.

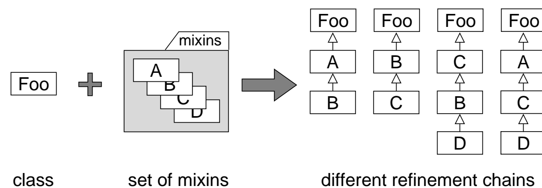

Fig. 2. Composing mixins in different ways to form alternative refinement chains.

The Jak language. Jak extends Java with mixin capabilities [5]. Jak serves as our archetype for enhancing AOP languages with constructs for implementing refinements via mixins.

Jak 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). Note that the refinement is applied in a different development step than the base class *Fifo*. For brevity, we depict in code listings classes (and aspects) together with their subsequent refinements. In contrast to traditional mixins, a Jak mixin has no explicit name (Line 5). It simply encapsulates a refinement to *Fifo* (a.k.a. *program delta/increment*). The identity of a Jak mixin is determined by its association to a development step. That is, the actual meaning of the mixin is moved out of the code to an external instance that manages all refinements [5]. A development step usually refines several classes.

Though in the original approach mixins are named entities, we consider only the Jak variant of 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*. Hence, introducing class *Fifo* is the initial development step (*Base*). In a subsequent development step the size limitation refinement is applied (*Size*).

```
class Fifo {           // step 1: basic fifo buffer
  void put(Item e) { }
  Item get(int i) { }
}
refines class Fifo {  // step 2: bounded fifo buffer
  static int max = 100;
  void put(Item e) {
    if(fill < max) super.put(e);
  }
}
```

Fig. 3. Refinements in Jak.

3.2 Aspects and Mixin Composition

Figure 4 shows a synchronization aspect (Lines 1-4) and a refinement (Lines 5-15). Both are composed via mixin-based aspect inheritance: an aspect together with all of its refinements constitute the final aspect that is woven *once* to the base program.

Refinements can be applied to abstract and concrete aspects as well as to aspect refinements. This eliminates the dilemma to anticipate subsequent added refinements by declaring aspects to refine as abstract. As with traditional inheritance, the ordering of applying refinement is not arbitrary and affects the program semantics (see Sec. 3.3).

Adding Members and Extending Methods Aspects may extend parent aspects by adding new members. As shown in Figure 4, the refinement adds a field (Line 6), a pointcut (Lines 9-10), and an advice (Lines 11-14). Aspects may also extend methods to reuse existing functionality. An extension usually overrides and calls the parent method (Lines 7,8).

```
aspect Sync {
  void lock() { /* locking access */ }
  void unlock() { /* unlocking access */ }
}
refines aspect Sync {
  int threads;
  void lock() { threads++; super.lock(); }
  void unlock() { threads--; super.unlock(); }
  pointcut syncPC() : execution(* Fifo.get(..))
                   || execution(* Fifo.put(..));
  Object around() : syncPC() {
    lock(); Object res = proceed(); unlock();
    return res;
  }
}
```

Fig. 4. Adding members and extending methods.

Pointcut Refinement A refinement may extend the pointcuts of the parent aspect.[3] Recall our example aspect that synchronizes calls to *Fifo* (cf. Fig. 1). For this aspect we defined two refinements, an aspect that extends the set of join points by all method calls to *Stack* (III), and an aspect that constrains this set to calls that originate from *Socket* (IV). Both aspects were derived using traditional aspect inheritance. They override the parent pointcut (*syncPC*), reuse the parent, and add new pointcut expressions that extend or constrain the set of matched join points. Hence, the refinements reuse the parent aspect's functionality for synchronization.

Figure 5 shows how a pointcut triggers a corresponding advice (dashed arrow). Figure 6 illustrates how refining a pointcut (solid arrow) alters the triggering mechanism: the most refined pointcut triggers the advice, albeit the advice was defined in a parent aspect.

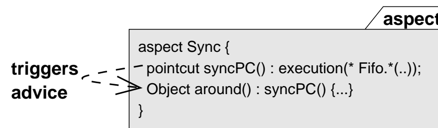

Fig. 5. Pointcut-advice-binding.

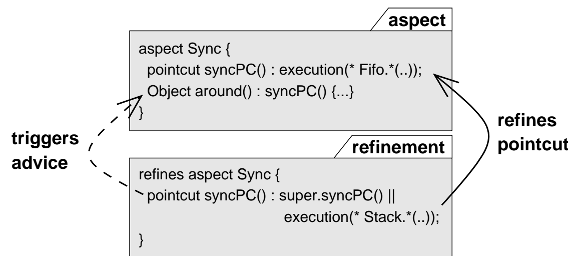

Fig. 6. The most refined pointcut triggers the connected advice.

In 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. Figure 7 depicts the synchronization aspect for *Fifo* and our refinements regarding *Stack* and *Socket*, but now implemented using mixin-based aspect inheritance. This example clarifies an important advantage of mixin-based aspect inheritance related to pointcut reuse. Using *super* the programmer refers to the parent's pointcut

[3] Hanenberg et al. propose several design patterns that utilize those techniques to improve reusability and extensibility of aspects [13].

without being aware of what actual sequence of refinements is applied to the base aspect. With traditional inheritance each refinement would change the final type of the aspect and thereby fix the refinement order. With aspect refinement the order is variable and allows aspects to be reused and customized by composing refinements in different permutations.

```
1  aspect Sync {           // synchronize Fifo
2    pointcut syncPC() : execution(* Fifo.get(..))
3                     || execution(* Fifo.put(..));
4    Object around() : syncPC() { /* synchronization */ }
5  }
6  refines aspect Sync { // synchronize Stack
7    pointcut syncPC() : super.syncPC()
8                     || execution(* Stack.*(..));
9  }
10 refines aspect Sync { // only within cflow of Socket
11   pointcut syncPC() : super.syncPC()
12                    && cflow(execution(* Socket.*(..)));
13 }
```

Fig. 7. Altering the set of locked methods via pointcut refinement.

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

Named advice. In order to refine advice in subsequent development steps, they must be named entities. Hence, we propose the notion of *named advice*[4]. Named advice are named elements of aspects. They can be overridden and referred to from a child advice to refine their functionality. This enables the programmer to reuse and evolve advice over several development steps.

Figure 8 depicts a synchronization aspect that contains a named advice (Lines 3-6). The advice consists of a name (*syncMethod*) and a binding to a pointcut (*syncPC*). One can think of a named advice as a pair of an unnamed advice and a separate method (*advice method*). The advice method contains the whole advice functionality. The unnamed advice simply calls this method and passes all arguments. The difference is that named advice has full access to the dynamic context (*proceed* and join point API). Though named advice can be implemented differently, this view is helpful for understanding the semantics of advice refinement.

Refining named advice. By introducing named advice programmers are able to refine advice of parent aspects. The key idea is to treat named advice in subsequent refinements similarly to methods. As mentioned, named advice can be understood roughly as pair of unnamed advice and corresponding advice

[4] An early version of AspectJ had a related language construct. It was removed in favor of unnamed advice.

```
aspect Sync {
  pointcut syncPC() : execution(* Fifo.*(..));
  Object around syncMethod() : syncPC() {
    lock(); Object res = proceed(); unlock();
    return res;
  }
}
```

Fig. 8. The notion of *named advice.*

method. Thus, an advice refinement simply refines the advice method. This is reasonable because the advice method contains the entire advice functionality.

Figure 9 depicts an aspect that refines our synchronization aspect, shown in Figure 8, by extending its named advice. The refinement introduces an advice method *syncMethod* (Lines 2-5) that overrides the parent named advice by counting the number of threads. Since we exploit method overriding, the refining method must have the same name and the same signature than the parent advice. The *super* keyword is used to refer to the parent advice (Line 3). This promotes reuse in the same way as method extension within classes.

```
refines aspect Sync {
  Object syncMethod() {
    count++; Object res = super.syncMethod(); count--;
    return res;
  }
}
```

Fig. 9. Refining named advice.

Figure 10 depicts a more complex example that uses multiple arguments. A logging aspect intercepts all executions of *Item.toString* (Lines 2-3). A reference to the *Item* object that is called is passed to a named advice (Lines 4-7) that prints out some logging text (Line 6). Additionally the named advice has a second argument, a reference to the resulting *String* object (Line 4). When refining this named advice subsequently (Lines 11-14), we have to introduce an advice method with the same name *and* the same signature. In our example the signature is composed of the two advice arguments.[5]

Named advice behave similarly to virtual methods, which pass the control flow to the most specialized descendant method of the inheritance chain. Mapped to our approach this means that when the associated pointcut matches the most specialized advice method is invoked. Figure 11 shows how in our running example the most refined version of advice *syncMethod* is executed (dashed arrows) when the pointcut *syncPC* matches. Programmers can use *super* to navigate

[5] This is because of advice declaring the argument list at two positions in their declarations, behind their name and behind the *returning* or *throwing* statement.

```
 1 aspect Logging {
 2   pointcut ItemToString(Item i) :
 3     execution(* Item.toString()) && this(i);
 4   after LogToString(Item i) returning(String s) :
 5     ItemToString(i) {
 6     System.out.println("item:" + i + "=" + s);
 7   }
 8 }
 9 refines aspect Logging {
10   FileBuffer buf = new FileBuffer("foo");
11   void LogToString(Item i, String s) {
12     super.LogToString(i, s);
13     buf.write("item:" + i + "=" + s);
14   }
15 }
```

Fig. 10. Refining named advice with arguments.

the refinement chain upstairs (solid arrows). The root of the refinement chain defines to which pointcut the advice is bound.

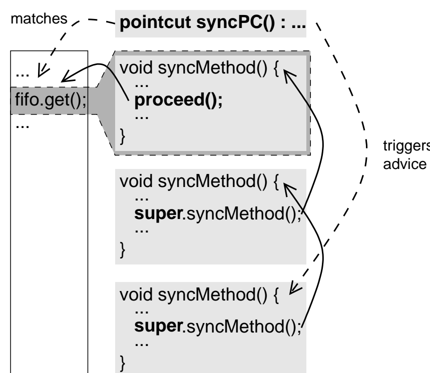

Fig. 11. Refinement chain navigation.

Refining advice methods yields some interesting issues regarding the use and access of *proceed* and of contextual information. Up to now, we made no statement which information of the exposed context of a join point should be visible to descendant advice methods. This issue arises because programmers may access the context using *proceed* or runtime variables as *thisJoinPoint*. Thus, one may use information that is not explicitly passed via the advice interface. The questions that arises is, should refinements have access to context information and *proceed*?

We argue that advice refinements should only be permitted to access those pieces of context information that are passed via the advice interface, and thus part of the advice method signature. Figure 10 shows that the refined aspect accesses such explicitly passed context information (Lines 10,11). To preserve

simplicity and safety the usage of the reflective support for accessing context information (e.g., *thisJoinPoint*) is forbidden in advice refinements. Furthermore, we do not allow named advice to be invoked directly by other advice and methods.

3.3 Discussion

In order to reflect the idea of SWR to programming language level, we proposed to implement aspect refinement using mixin-based inheritance and mixin composition. According to our approach, aspects and their structural elements are refined via member introduction (adding fields, methods, pointcuts, and advice), method refinement, pointcut refinement, and advice refinement. All these refinement mechanisms are equal in their inherent structure; refinements override parent elements with the same name and signature; they may use *super* to refer to the parent.

We left out the possibility to refine aspects via pointcuts and advice. Although this is supported by our approach, a comparison of both approaches is out of scope. The strengths and weaknesses of mixin composition compared to pointcuts and advice are discussed elsewhere [12, 14]. Note also that this issue is not specific to aspect refinement, but to refinement in general.

One may argue that these few new mechanisms complicate AOP languages. We believe that the simple elegance of uniform extensibility explicitly reflects the conceptual nature of refinement and thereby facilitates a better and more flexible design and implementation. However, this flexibility bears also some risks. Although our approach does not impose a fixed order of refinements, the final order affects the program semantics. Programmers may not be aware of potential conflicts of certain orderings. We suggest to employ established mechanisms for design rule checking to discover and prevent invalid orderings (e.g., [15]).

Nevertheless, aspect refinement improves the customization and the reuse of aspects by enabling programmers to select the desired functionality. By this means, aspects can be tailored to specific programs and requirements: pointcuts can be refined to match desired join points; advice can be refined to customize the added functionality; desired members can be combined into one tailored aspect.

4 Syntax and Semantics

In order to define our language proposal in a precise way, we provide a formal specification of its syntax and semantics on the basis of AspectJ.

4.1 Abstract Syntax

Figure 12 depicts the formal specification of the abstract syntax of our language proposal. The depicted rules partially extend grammar specification given in [16, 17]. The new rules cover aspects, pointcuts, and named advice.

$$
\begin{array}{lll}
P \in \mathit{programs} & ::= & \bar{D} \\
D \in \mathit{declarations} & ::= & C \mid A \mid R \\
C \in \mathit{classes} & ::= & \textbf{class}\ c_1 <: c_2\ \{\bar{F}, \bar{M}\} \\
F \in \mathit{fields} & ::= & c\ f \\
M \in \mathit{methods} & ::= & c_r\ m(\vec{c}\ \vec{x})\ \{\vec{S}\} \\
S \in \mathit{statements} & ::= & c.m(\vec{S}) \mid \textbf{new}\ c(\vec{S}) \mid x \mid \ \dots \\
A \in \mathit{aspects} & ::= & \textbf{aspect}\ a_1 <: a_2\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\} \\
R \in \mathit{refinements} & ::= & \textbf{refines aspect}\ a\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\} \\
\Pi \in \mathit{pointcuts} & ::= & \textbf{pointcut}\ \pi(\vec{c}\ \vec{x})\ :\ \Phi \\
\Phi \in \mathit{pointcut\ expressions} & ::= & \textbf{false} \mid \neg\Phi \mid \Phi \vee \Phi\prime \mid \Psi \\
\Psi \in \mathit{atomic\ pointcut\ expressions} & ::= & \textbf{super} :: \pi(\vec{x}) \mid \textbf{call}(c_r\ c :: m(\vec{c}\ \vec{x})) \mid \\
 & & \textbf{execution}(c_r\ c :: m(\vec{c}\ \vec{x})) \mid \ \dots \\
\Omega \in \mathit{advice} & ::= & c_r\ \textbf{advice}\ \omega(\vec{c}\ \vec{x})\ \textbf{:}\ \pi(\vec{x})\ \{\vec{S}\} \mid \\
 & & c_r\ \textbf{advice}(\vec{c}\ \vec{x})\ :\ \pi(\vec{x})\ \{\vec{S}\}
\end{array}
$$

Fig. 12. Abstract syntax of the extended AspectJ grammar.

We use the following conventions for meta-variables used in the rules: Variables written in capital letters represent the main language constructs. They are the non-terminal symbols of our grammar and represent particular kinds of declarations. Metavariables in lower case letters are placeholders for program variable names. Keywords are highlighted in bold letters.

The metavariables used in the grammar are reserved to their particular types, e.g., we use $c_1, \dots, c_n$ for referring to class names and C to refer to a class declaration. $\vec{X}$ means a sequence of elements and $\bar{X}$ a 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.

As Figure 12 shows, a program P consists of a set of declarations. Declarations can be classes, aspects, and refinements. Class declarations (C) consist of a name (c_1), a reference to a parent class (c_2), and a set of fields ($\bar{F}$) and methods ($\bar{M}$). Inheritance is expressed by $<:$. A field declaration (F) contains a type (c) and a name (f). A method declaration (M) consists of a name (m), a results type (c_r), and a set of arguments ($\vec{c}\ \vec{x}$). $\vec{c}\ \vec{x}$ means a sequence of arguments with its particular types: $c_1\ x_1, \dots, c_n\ x_n$. A method executes a sequence of statements ($\vec{S}$). The possible kinds of statements (S) are not further explained. We assume a standard set (see [16]).

Aspects (A) declare a name (a_1), a parent (a_2), and contain a set of fields ($\bar{F}$), methods ($\bar{M}$), pointcuts ($\bar{\Pi}$), and advice ($\bar{\Omega}$). Refinements (R) are similar to aspects but they do not declare a name and cannot be derived from other external aspects. The affiliation to a development step is not modeled within the grammar because it is defined and managed outside the actual code. When defining the semantics we use an auxiliary function to determine the current parent (cf. Sec. 4.2).

Pointcuts (Π) declare a name (π), a set of arguments ($\vec{c}\ \vec{x}$), and a pointcut expression (Φ). Such pointcut expressions are build of atomic pointcut expressions (Ψ) and can be negated and logically combined (for simplification, we consider only disjunctions). In contrast to AspectJ, parent pointcuts may be accessed via

super. The set of atomic pointcut expressions is determined by the host AOP language, in our case by AspectJ.

When declaring advice we distinguish between named and unnamed advice. Hence, advice declarations (Ω) are either named (ω) or unnamed. Both types of advice expect a set of arguments ($\vec{c}\ \vec{x}$), have a result type (c_r), are bound to a specific pointcut (π)[6], and execute a sequence of statements ($\vec{S}$). Unnamed advice are included to be compatible to standard AspectJ. The *advice* keyword abstracts over all possible advice types, e.g., *before*, *after*, and *around* [16].

$$\frac{\begin{array}{c} a_1 \ = \ concat(a2, id()) \\ \textbf{aspect}\ a_2\ \{_,_,_,_\} \ \in \ P \end{array}}{P \ \vdash \ \textbf{refines aspect}\ a_2\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\} \ \rightarrow \ \textbf{aspect}\ a_1 <: a_2\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\}} \quad \text{(R-REF)}$$

$$\frac{\begin{array}{c} \Phi_i \ \rightarrow \ \Phi\prime_i \\ \bar{\Pi}\prime_1 \ \ni \ \Phi\prime_i = \ [\textbf{super} \ \mapsto \ a_2]\Phi_i \\ \textbf{pointcut}\ \pi_i(\vec{c}\ \vec{x}) \ : \ \Phi_i \ \in \ \bar{\Pi}_1 \\ \textbf{super} :: \pi_i(\vec{c}\ \vec{x}) \ \in \ \Phi_i \ \Rightarrow \ \textbf{pointcut}\ \pi_i(\vec{c}\ \vec{x}) \ : \ _ \ \in \ \bar{\Pi}_2 \\ \textbf{aspect}\ a_2\ \{\bar{F}_2, \bar{M}_2, \bar{\Pi}_2, \bar{\Omega}_2\} \ \in \ P \end{array}}{P \ \vdash \ \textbf{aspect}\ a_1 \ <: a_2\ \{\bar{F}_1, \bar{M}_1, \bar{\Pi}_1, \bar{\Omega}_1\} \ \rightarrow \ \textbf{aspect}\ a_1 <: a_2\ \{\bar{F}_1, \bar{M}_1, \bar{\Pi}\prime_1, \bar{\Omega}_1\}} \quad \text{(R-SUP)}$$

$$\frac{\begin{array}{c} \Omega_i \ \rightarrow \ \Omega\prime_i, \ M_i \\ \Omega_i \ = \ c_r\ \textbf{advice}\ \omega_i(\vec{c}\ \vec{x}) \ : \ \pi_i(\vec{x})\ \{\vec{S}_i\} \ \in \ \bar{\Omega} \\ M_i \ = \ c_r\ m_i(\vec{c}\ \vec{x})\ \{\vec{S}_i\} \ \in \ \bar{M} \\ \Omega\prime_i \ = \ c_r\ \textbf{advice}(\vec{c}\ \vec{x}) \ : \ \pi_i(\vec{x})\ \{\textbf{return}\ a_1.m_i(\vec{x})\} \ \in \bar{\Omega}\prime \end{array}}{P \ \vdash \ \textbf{aspect}\ a_1 \ <: a_2\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\} \ \rightarrow \ \textbf{aspect}\ a_1 <: a_2\ \{\bar{F}, \bar{M}, \bar{\Pi}, \bar{\Omega}\prime\}} \quad \text{(R-ADV)}$$

Fig. 13. Reduction rules.

4.2 Semantics

We introduce a set of reduction rules to specify the semantics of our language proposal. These rules extend a base semantics specification for AOP languages [16]. To determine the behavior of a program that makes use of our extensions, one has to include additionally our reduction rules depicted in Figure 13. Furthermore, we presume that the rules are applied in the order explained below.

Modeling the fact that aspects belong to development steps, we use two auxiliary functions. Function *id* returns for each aspect the identifier of the associated development step and function *concat* concatenates two names (using string concatenation).

Rule R-REF reduces an aspect refinement to an 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 subaspects. The order of the inheritance chain is inferred from the order of the refinement

[6] For simplicity, we limit advice to be bound only to named pointcuts.

chain. The name of the new aspect is composed of the name of the parent aspect and the identifier of the associated development step (using *id* and *concat*).

Rule R-SUP reduces all pointcuts (Π_i) whose expressions (Φ_i) contain *super* to expressions ($\Phi\prime_i$) that fully qualify the parent pointcut by it complete name ($a_2 :: \pi_i$). Premises are that the corresponding pointcut expressions (Φ_i) contain *super* and that a parent aspect (a_2) exists which contains a corresponding pointcut with the same name (π_i) and the same signature ($\vec{c}\ \vec{x}$).

Rule R-ADV defines the reduction of named advice to AspectJ-compliant unnamed advice. For each named advice (Ω_i) a pair of an unnamed advice ($\Omega\prime_i$) and an advice method is generated (M_i). Both get the same signature ($\vec{c}\ \vec{x}$). $\Omega\prime_i$ simply calls the associated advice method m_i and passes the necessary arguments. What is not shown is that if the reflective join point API is used within the advice the corresponding runtime objects are passed to the called advice method, e.g., *thisJoinPoint*. Advice refinement is not further explained since it is implemented by method overriding.

5 Implementation and Case Study

To demonstrate the practical applicability of our approach, (1) we implemented a fully functional compiler that extends AspectJ, called *arj*, and (2) we applied it to a non-trivial case study.

5.1 ARJ Implementation

Our implementation is based on a program transformation approach. It transforms *abstract syntax trees (ASTs)* containing our language constructs to abstract syntax trees that are AspectJ compliant. Our compiler is implemented as an extension to the *abc* compiler framework [18]. This extension adds several frontend and backend passes for implementing the syntax tree transformation. To establish the mapping between aspects and their associated development steps, we maintain for each input program (base program + aspects + refinements) a directory structure. Development steps are represented by directories that contain the associated aspect and refinement files. A concrete configuration is specified by enumerating the directory names that represent the desired refinements. Passing different orders results in differently composed programs.

The current status of arj supports all proposed language constructs. It is implemented as modular extension to the abc (*abc.arj*). The compiler as well as several documents and examples can be downloaded at the arj web site[7].

5.2 Case Study

Using our compiler we applied aspect refinement to a non-trivial case study, a *product line for peer-to-peer overlay networks (P2P-PL)*. Since evaluating aspect

[7] http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/

aspect (# pieces)	**description**
responding (5)	sends replies automatically
serialization (16)	prepares objects for serialization
toString (14)	introduces *toString* methods
log/debug (18)	mix of logging and debugging
pooling (4)	stores and reuses open connections
dissemination (12)	piggyback meta-data propagation
feedback (6)	generates feedback by observing peers
caching (7)	caches peer contact data

Table 1. Refactored aspects in P2P-PL.

refinement was part of a more comprehensive study that undertook a thorough empirical evaluation of program features/components and aspects with respect to SWR, we repeat here only the key results. The original study is published elsewhere [14].

P2P-PL was implemented to experiment with advanced overlay network features such as query evaluation optimization and decentralized meta-data propagation. Thus, there was a need for a highly customizable architecture that allows for reusing features in different configurations. Our goal was to improve the structuredness of the P2P-PL design as well as the reusability and customizability of the contained aspects.

The code base of P2P-PL is about 6426 LOC. In summary, it contains contains 14 aspects (406 LOC – 6%). We applied the notion of aspect refinement to 8 aspects. That is, we refactored each aspect into several pieces (one base aspect and several refinements). Table 1 gives an overview of the refactored aspects and the application of aspect refinement.

Certainly, we applied the notion of aspect refinement only to those aspects that seemed promising. However, over 1/2 of all aspects (8 of 14) shaped up as good candidates for decomposition via aspect refinement.

On average, we decomposed the considered aspects into 10 pieces. This fine-grained decomposition did not only structure the design and implementation of P2P-PL, but also increased the configuration space, i.e., the tailored variants that can be derived by the configuration process. For example, the serialization aspect has as many variants as different sets of target classes are possible in P2P-PL (theoretically 2^{15}). In contrast, the caching aspect comes in significantly fewer variants (8). This is because the caching aspect has only 3 variable features (storage management, caching strategy, supported types of contact data); each of these features comes in 2 variations and none of them can be removed (e.g., we cannot remove the caching strategy but chose between 2 variants); 4 features are mandatory and shared by all variants.

In our study, all derivable variants of aspects share common features, thereby reusing aspect code. In case of the caching aspect each of the 8 variants reuses code of 4 common features. The individual variants of the serialization aspect

share only 1 common base aspect. This indicates a trade-off between fine-grained customizability and reuse of aspect code.

Programming guidelines. In our study we identified two main use cases of aspect refinement: A first use case is to decouple aspects from a particular set of classes to be extended/advised (first four aspects in Tab. 1). For that, we decomposed aspects into several pieces to enable the programmer to combine these pieces in different combinations. For example, one aspect in P2P-PL adds capabilities for serialization to a set of classes. We decomposed this aspect to be able to chose only those pieces for adding serialization functionality that are actually needed in a particular P2P-PL configuration. This improves reuse and robustness of this aspect in varying contexts, i.e., in different configurations of P2P-PL that contains different sets of classes to be extended/advised.

A second use case is to encapsulate the effects of different design decisions into refinements (last four aspects in Tab. 1). Decomposing aspects along design decisions allows for customization, i.e., by selecting different subsets of the overall set of refinements that are desired for a particular situation. Thereby, aspects can be tailored to different base programs and to varying requirements. For example, we decomposed an aspect for contact caching into 6 pieces that encapsulate design decisions such as the caching strategy, contact types, and storage structure. By doing that, we were able to add and remove functionality and to select alternative implementations. This facilitates reuse of invariant aspect code and enables to customize and tailor aspects to different P2P-PL configurations and to different requirements, e.g., performance.

6 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 [19]. 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. Thereby, they can be modified, combined, and extended. In this point our approach of aspect and pointcut refinement is similar. We can combine, modify, and extend pointcuts by applying 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 advice is similar to passing advice to higher-order advice.

Aspect refinement and AHEAD. The idea of aspect refinement emerged from prior work on aspect-oriented and feature-oriented product lines and *AHEAD* [12]. AHEAD is an architectural model for large-scale program synthesis [5]. It models 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 by *super-imposition* [20]. Hence, for each artifact type a distinct implementation of the *polymorphic composition operator* has to be provided.

In context of the AHEAD model, mixin-based aspect inheritance is a composition operator that is invoked when aspects (and their refinements) of different development steps are composed. Since our aspects and their refinements are associated to development steps and this association is maintained externally, they fit the AHEAD approach of algebraic equation-based composition.

Aspectual mixin layers (AMLs) integrate aspects and features in the sense of the AHEAD model [12]. The synergetic effects of aspects, features, and SWR exhibit an improvement over traditional layered designs based on classes and traditional AOP, e.g., the crosscutting modularity is improved [12, 14]. Aspect refinement based on mixin-based aspect inheritance enhances AMLs towards a unified integration of features and aspects with regard to SWR.

Aspect refinement and collaborations. Aspect refinement is related to collaboration-based designs and their symbiosis with AOP mechanisms, e.g., *Caesar* [21], *Aspectual Collaborations* [22], and *Object Teams* [23], to name a few. Since these approaches were highly influenced by one another, we relate our approach to Caesar, which unifies the most essential ideas that are relevant for our work.

Caesar supports componentization of aspects by encapsulating virtual classes as well as pointcuts and advice in collaborations, so called *aspect components*. Aspect components can be composed via their collaboration interfaces and mixin composition in a stepwise manner. Besides this, they can be refined using pointcuts in order to implement crosscutting integration.

With the mentioned approaches it is not possible to refine embedded pointcuts and advice. They do not uniformly support SWR at language level; but there is no reason why aspect refinement could not be integrated. Furthermore, their collaborations (aspect components) are first-class and their composition is done within source code. There is no separation of the source code artifacts and their association to development steps. We have shown that even this separation facilitates the composition, reuse, and customization of aspects.

Aspect quantification and composition. Traditionally, aspects are quantified globally. That means they may potentially affect all program elements. Unfortunately, this attitude ignores the principle of SWR that refinements are permitted to affect only those refinements that were applied in previous development steps [2, 4]. Several studies have shown that this circumstance is directly responsible for inadvertent aspect interactions and an unpredictable behavior [24–27]

In order to address this issue, recent studies proposed to model aspects as functions that operate on programs [24, 28]. Applying several aspects to a program is modeled as function composition. In this way the scope of aspects is

restricted to a particular step in a program's development. Such *bounded quantification* of aspects satisfies the principles of SWR.

The idea of bounding aspect quantification can seamlessly be integrated in our approach: Since our compiler knows for each aspect and for each refinement to which development step it belongs, it can determine which program elements the aspects are permitted to affect. That is, the compiler uses meta-data to control the weaving process. One implementation approach is to restructure the pointcut expressions of the aspects and their refinements so that they affect only those program elements that were introduced in previous development steps [29].

What is important is that our notion of aspect refinement (embodied in our compiler) allows for the first time to *implement and experiment with* bounded aspect quantification.

Our approach is different from other work in this field [30, 31]. We exploit SWR principles that allow us to associate aspects with development steps. Hence, a bounded quantification mechanism uses the implicit knowledge of the evolutionary grown design. This avoids a lot of explicit specifications and formulated constraints to be provided by the programmer.

Unifying advice and methods. Rajan and Sullivan propose *classpects* that combine capabilities of aspects and classes to unify the design of layered module systems [32]. A classpect 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. Since advice are implemented via methods they could be refined. However, the authors do not make a statement about this. Furthermore, our approach supports mixin composition and pointcut refinement.

Generic aspects. Several recent approaches enhance aspects with genericity, e.g., *Sally* [33], *Generic Advice* [34], *LogicAJ* [35], *Framed Aspects* [36]. This improves reusability of aspects in different application contexts. Aspect refinement and mixin-based aspect inheritance provides an alternative way to customize aspects, i.e., by composing the required refinements. However, ideas on generic aspects can combined with our compositional approach.

AspectJ design patterns. Hanenberg and Unland discuss the benefits of inheritance in the context of AOP [37, 13]. They argue that aspect inheritance improves aspect reuse and propose 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.

Implementing refinement. Our approach for implementing aspect refinement is based on mixins and AHEAD. We chose mixins because of their success in numerous domains, e.g., [38–41]. Also the appropriate integration of mixin concepts

into prominent programming languages as Java [38, 5] or C++ [42, 43] was a motivating reason. The paper has shown that mixin capabilities indeed increase the flexibility to compose, reuse, and customize aspects in layered designs. However, alternative mechanisms may achieve similar results.

For example, *traits* aim at structuring object-oriented programs [44]. Traits are units of code reuse that group multiple methods, but not state-holding members. Several traits can be combined, using *glues*, to a customized final class. Traits offer customizability at a more fine-grained level than mixins. They could be used to implement refinements of aspects that consist of pointcuts, advice and methods.

Feature-optionality problem. In feature-oriented programming, the problem of optional features arises when features depend on (or interact with) other features that are optional [45, 46]. In order to be reliable with regard to putting in and removing optional features, Prehofer proposes to split features into slices, i.e., into a base feature and several so called *lifters* [45]. Lifters encapsulate those pieces of code that depend on other features. When composing a program out of features a programmer or a tool selects for each feature the base feature and those lifters that refer to features that actually participate in the current configuration. Liu et al. lay for this methodology an algebraic foundation [46].

Our methodology to split aspects into pieces to resolve dependencies between aspects and classes of a base program is similar to their approach: Our refinements correspond to lifters, but in the context of AOP.

7 Conclusion

The paper addressed the unification of AOP and SWR. SWR is fundamental to software development. Aspect refinement is its incarnation for AOP. We argued that the principles of SWR have to be reflected at programming language level. Consequentially, we proposed mixin-based aspect inheritance and a set of accompanying language constructs that facilitate SWR. Aspect refinement and the enabling language mechanisms unify classes and aspects with respect to SWR.

Our proposed language mechanisms aim at improving the reusability and customizability of aspects. Mixin composition enables aspects to be tailored to different application contexts: pointcut refinement allows for adapting an aspect to different base programs, by modifying the target join points. Advice refinement makes it possible to reuse and evolve existing advice code in subsequent development steps.

Our uniform approach to refinement allows for treating all structural elements of aspects equally with regard to subsequent refinement, including pointcuts and named advice. All parent entities are accessed uniformly via *super*. By allowing concrete aspects to be refined, the programmer does not need to anticipate subsequent refinements.

To underpin our proposal, we provided a formal syntax and semantics description and we implemented a fully functional compiler on top of AspectJ.

The compiler implements all proposed mechanisms and language constructs. We used the compiler to apply our approach to a non-trivial case study. The study demonstrated that technically our approach is realizable and applicable to a non-trivial software project. It has been shown that our proposed language mechanisms promote aspect composition, reuse, and customization in a SWR manner. Furthermore, the study revealed guidelines when aspect refinement is useful: (1) for decoupling aspects from fixed configurations of the base program, and (2) for structuring aspect-oriented designs along design decisions. By composing refinements that encapsulate design decisions or interactions with the base program, one customizes aspects to a specific context. Our language mechanisms facilitate this composition.

In further work we intend to address the following issues:

- How fit pointcuts and named advice polymorphism and virtuality?
- We did not consider intertype declarations, precedence rules, and access modifiers.
- An important point is to experimentally explore further useful design patterns of mixin-based aspect inheritance.
- A further interesting topic is to investigate the deeper relationship to higher-order aspects and an algebraic formalization of aspect refinement.

Acknowledgments

We thank Don Batory, William Cook, and Roberto Lopez-Herrejon for useful comments and fruitful discussions on earlier drafts of this paper. This work was done while Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. It is sponsored in parts by the German Research Foundation (DFG), project number SA 465/31-1 and by the German Academic Exchange Service (DAAD), PKZ D/05/44809.

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**(4) (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**(2) (1979)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering **30**(6) (2004)
6. Greenfield, J., et al.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley Publishing, Inc. (2004)
7. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley (2002)
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

9. 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)
10. Taivalsaari, A.: On the Notion of Inheritance. ACM Computing Surveys **28**(3) (1996)
11. 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)
12. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: Proceedings of International Conference on Software Engineering. (2006)
13. Hanenberg, S., Schmidmeier, A.: Idioms for Building Software Frameworks in AspectJ. In: AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. (2003)
14. Apel, S., Batory, D.: When to Use Features and Aspects – A Case Study. In: Proceedings of the International Conference on Generative Programming and Component Engineering. (2006)
15. Batory, D., et al.: Design Wizards and Visual Programming Environments for GenVoca Generators. IEEE Transactions on Software Engineering **26**(5) (2000)
16. Jagadeesan, R., Jeffrey, A., Riely, J.: A Calculus of Untyped Aspect-Oriented Programs. In: Proceedings of European Conference on Object-Oriented Programming. (2003)
17. 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**(3) (2001)
18. Avgustinov, P., et al.: abc: An Extensible AspectJ Compiler. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2005)
19. Tucker, D., Krishnamurthi, S.: Pointcuts and Advice in Higher-Order Languages. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
20. Bosch, J.: Superimposition: A Component Adaptation Technique. Information and Software Technology **41**(5) (1999)
21. Aracic, I., et al.: An Overview of CaesarJ. Transactions on Aspect-Oriented Software Development **3880 (LNCS)**(1) (2006)
22. Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal **46**(5) (2003)
23. Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Proceedings of NetObjectDays. (2002)
24. 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)
25. 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)
26. Douence, R., Fradet, P., Südholt, M.: Composition, Reuse and Interaction Analysis of Stateful Aspects. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2004)
27. Douence, R., Fradet, P., Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. In: Proceedings of Generative Programming and Component Engineering. (2002)

28. Apel, S., Liu, J.: On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In: Proceedings of ECOOP Workshop on Aspects, Dependencies, and Interactions. (2006)
29. Kästner, C., Apel, S., Saake, G.: Implementing Bounded Aspect Quantification in AspectJ. In: Proceedings of ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution. (2006)
30. Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proceedings of European Conference on Object-Oriented Programming. (2005)
31. Ongkingco, N., et al.: Adding Open Modules to AspectJ. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2006)
32. Rajan, H., Sullivan, K.J.: Classpects: Unifying Aspect- and Object-Oriented Language Design. In: Proceedings of International Conference on Software Engineering. (2005)
33. Hanenberg, S., Unland, R.: Parametric Introductions. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2003)
34. 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)
35. 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)
36. Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. In: Proceedings of International Conference on Software Reuse. (2004)
37. Hanenberg, S., Unland, R.: Using and Reusing Aspects in AspectJ. In: OOPSLA Workshop on Advanced Separation of Concerns in OO Systems. (2001)
38. Cardone, R., et al.: Using Mixins to Build Flexible Widgets. In: Proceedings of International Conference on Aspect-Oriented Software Development. (2002)
39. Batory, D., et al.: Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. ACM Transactions on Software Engineering and Methodology **11**(2) (2002)
40. Batory, D., et al.: Creating Reference Architectures: An Example from Avionics. In: Proceedings of Symposium on Software Reusability. (1995)
41. VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: Proceedings of International Conference on Object-Oriented Programming Systems, Languages and Applications. (1996)
42. Apel, S., et al.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proceedings of International Conference on Generative Programming and Component Engineering. (2005)
43. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodology **11**(2) (2002)
44. Schärli, N., et al.: Traits: Composable Units of Behavior. In: Proceedings of European Conference on Object-Oriented Programming. (2003)
45. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proceedings of European Conference on Object-Oriented Programming. (1997)
46. Liu, J., Batory, D., Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In: Proceedings of International Conference on Software Engineering. (2006)