

Aspect Refinement – Unifying AOP and Stepwise Refinement

Sven Apel, Department of Informatics and Mathematics, University of Passau

Christian Kästner, School of Computer Science, University of Magdeburg

Thomas Leich, School of Computer Science, University of Magdeburg

Gunter Saake, School of Computer Science, University of Magdeburg

Stepwise refinement (SWR) is fundamental to software engineering. As *aspect-oriented programming (AOP)* is gaining momentum in software development, *aspects* should be considered in the light of SWR. In this paper, we elaborate the notion of *aspect refinement* that unifies AOP and SWR at the architectural level. To reflect this unification to the programming language level, we present an implementation technique for refining aspects based on mixin composition along with a set of language mechanisms for refining all kinds of structural elements of aspects in a uniform way (methods, pointcuts, advice). To underpin our proposal, we contribute a fully functional compiler on top of AspectJ, present a non-trivial, medium-sized case study, and derive a set of programming guidelines.

1 INTRODUCTION

Aspect-oriented programming (AOP) is a novel programming paradigm to implement complex software in a modular way [16]. *Aspects*, the main abstraction mechanism of AOP, modularize crosscutting concerns. Without aspects, the implementation of such concerns would result in code scattering, tangling, and replication.

AOP is gaining momentum and pervading more and more phases and parts of software engineering [12]. This paper relates AOP to *stepwise refinement (SWR)*, a fundamental approach to software development [33, 13, 26, 8]. By adding new program details in a stepwise manner, the programmer breaks down complex software into manageable pieces (*modules*). The resulting program structure is supposed to be more comprehensible, reusable, and customizable – compared to a monolithic structure [33, 26, 8]. By its incremental development methodology SWR, facilitates software evolution, where each increment in program functionality reflects an evolutionary development step. Sometimes these increments are called *refinements* [8, 25].

Since most software is developed, adapted, and evolved in a more or less incremental way [29], it is desirable that modern programming paradigms reflect this by explicit support of SWR. That is, the principles of SWR should be taken into account when designing AOP languages, e.g., all software artifacts are subject of the subsequent refinement [8]. To this day, aspects have not been not adequately studied and understood with respect to SWR, with some notable exceptions [24, 21].

In this paper, we develop the idea of *aspect refinement (AR)*, which is the application of SWR principles to AOP [4]. It is a design methodology to incrementally develop, adapt, and evolve aspects by means of SWR. Our study of AR explores

the differences of AOP and SWR and proposes a convergence of both. AR unifies aspects and classes with respect to SWR and improves the reusability and customizability of aspects, e.g., it enables the adaptation of aspects to changed requirements or to a modified base program.

In prior work we outlined the mere possibility of refining aspects and examined the consequences for software development [4, 5, 1]. In this paper we explore how to support AR at the language level by taking *AspectJ*¹ as an archetype as well as by a non-trivial case study. We introduce the notion of *mixin-based inheritance* [11] to AOP. *Mixin-based aspect inheritance* explicitly supports SWR at the language level by introducing mixin capabilities to aspects. Though most aspect languages such as AspectJ support a limited form of aspect inheritance, they do not do so flexibly enough to express refinements of aspects and their structural elements, as we will discuss. Mixin-based aspect inheritance overcomes this limitation and enables programmers to freely compose aspects and their refinements. This provides the required flexibility to reuse, customize, and evolve aspects in the sense of SWR.

Furthermore, we propose a uniform approach for refining all kinds of structural elements of aspects. Specifically, we propose mechanisms for refining pointcuts (*pointcut refinement*) and advice (*named advice*, *advice refinement*), which are tailored to AspectJ-like languages.

We demonstrate the practical applicability of our language proposal by providing a fully functional compiler on top of AspectJ. We use our compiler to apply AR to a non-trivial, medium-sized case study. Based on this study, we propose a set of programming guidelines for applying AR. In this paper we make the following contributions:

- an elaboration of the idea of AR that unifies AOP and SWR
- the mechanism of mixin-based aspect inheritance, which is accompanied by language mechanisms for refining aspects, pointcuts, and advice
- a fully functional compiler on top of AspectJ that implements our proposal
- a case study and a set of programming guidelines

2 ASPECTS AND STEPWISE REFINEMENT

AHEAD – The Big Picture

AHEAD (Algebraic Hierarchical Equations for Application Design) is an architectural model for SWR and a framework for large-scale program synthesis based on features [8]. A *feature* is an increment in program functionality and corresponds to a development step. The goal of AHEAD is to synthesize software (individual programs) by composing a series of desired features.

AHEAD unifies several approaches of SWR and scales them to programming in the large. First, AHEAD generalizes the operations a feature performs on a given base program: (a) the introduction of new functionality and (b) the modification of existing functionality. Second, AHEAD scales the idea of program refinement to ar-

¹<http://eclipse.org/aspectj/>



bitrary kinds of software artifacts (e.g., code, test cases, documentation, makefiles), which is captured by the underlying principle of uniformity: *features are implemented by a diverse selection of software artifacts and any kind of software artifact can be subject to subsequent refinement* [8].

In AHEAD, features are modeled as *functions*. A *constant function* (a.k.a. *constant*) represents a base program. All other functions take a program as input and return a modified program as output. That is, functions represent program refinements that implement program features. For example, ‘*Add* • *X*’ adds feature *Add* to program *X*, where ‘•’ denotes function composition. A generated program is represented by a named *feature expression*, e.g., ‘*Prog* = *Add* • *Base*’.

In AHEAD, each feature is represented by a *containment hierarchy*, which is a directory that exhibits a subdirectory structure to store the feature’s artifacts. Composing features means composing containment hierarchies and, to this end, composing respective artifacts by *mixin composition* [11, 31]. Hence, for each artifact type, a different implementation of the *composition operator* has to be provided.

For example, Figure 1 depicts a Java class *Add* (Lines 1–3), which is part of feature *Base* and a refinement (Lines 4–7), which is part of feature *Buffer*. The refinement adds a new field (Line 5) and extends an existing method via overriding (Line 6); calling *super* invokes the refined method. The refinement is implemented in *Jak* [8], a Java language extension that introduces basically the keyword *refines*.

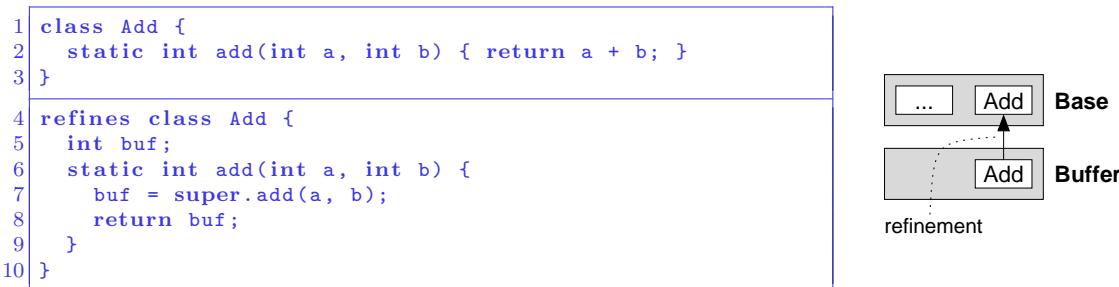


Figure 1: A class and a refinement in Jak.

Aspects – Just Another Kind of Software Artifact

In AHEAD, a feature is implemented by a collection of collaborating software artifacts of varying types. In this sense, an aspect is just another kind of software artifact. The AHEAD principle of uniformity has an interesting consequence: since aspects are artifacts as any others, it is natural to refine them in a SWR manner as well. That is, a feature may not only extend and modify *classes* via subsequent refinement but also *aspects*, which we call AR. Hence, AR is the consequential application of SWR principles to the world of AOP.

With AR, aspects can be adapted, customized, evolved, as can all other software artifacts. In each development step, aspects may be refined, i.e., extended and modified. We focus on three use cases of AR, which may overlap in parts:

1. Adapting aspects to the changes made to a base program, e.g., join points have changed or new join points occur.
2. Tailoring aspects to changing user requirements, e.g., the user needs an aspect to implement a new design decision.
3. Decomposing aspects to decouple them from a specific configuration of the base program, e.g., a base program in different configurations demands aspects in different variants.

Applying AR to deal with the above situations means decomposing and subsequently composing an aspect out of a *base aspect* and a *series of refinements*. Refinements should be freely combinable – of course, in the limits of desired program behavior. This flexibility facilitates reuse of aspect code. The user-driven composition of aspects and refinements customizes aspect-specific functionality. AR enables a similar improvement in reusability and customizability of aspect code as the analogous object-oriented mechanisms do for classes [11, 31, 8].

AR unifies classes and aspects with respect to subsequent refinement. An advantage of this view is that several ideas of class refinement can be mapped directly to aspects, as we will show. 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.

An Example of Aspect Refinement

Figure 2 illustrates the evolution of a program developed using aspects. The program contains classes for buffers and sockets as well as aspects for synchronizing concurrent access. The evolution spans four steps shown in four subfigures (*Base*, *Sync*, *Stack*, *Socket*). Each development step is explained in terms of its Java/AspectJ code and in diagram form; refinements of aspects are implemented as subaspects; aspect weaving is denoted by dashed arrows.

Base: Buffer objects store sets of data items; the class `Buffer` provides the methods `put` and `get` for accessing the stored items.

Sync: The aspect `BufferSync` synchronizes the access to the methods `put` and `get` of `Buffer` by invoking the methods `lock` and `unlock`.

Stack: The class `Stack` is introduced; in order to synchronize the access to `Stack` objects, the aspect `StackSync` refines the aspect `BufferSync` and broadens the set of intercepted method executions by `push` and `pop`; for that it overrides and extends the pointcut `syncPC` of aspect `BufferSync`.

Socket: The class `Socket` is introduced; a `Socket` object uses several `Buffer` and `Stack` objects. The aspect `SocketSync` limits the set of synchronized methods to those that are inside the control flow of `Socket`, i.e., method executions are synchronized only when they are initiated directly or indirectly by a `Socket` object. This is achieved by overriding the pointcut `syncPC` and restricting the set of captured join points via the pointcut `cflow`.

This example illustrates the usefulness of refining aspects in a stepwise manner over several development steps. Aspect refinement is a logical consequence of applying SWR principles to AOP. The incremental development process makes the evolution



```

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

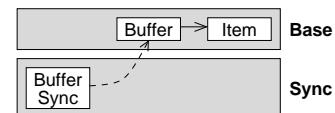
```



```

1 abstract aspect BufferSync {
2     pointcut syncPC() :
3         execution(Item Buffer.get(int)) ||
4         execution(void Buffer.put(Item));
5     Object around() : syncPC() {
6         lock(); Object res = proceed(); unlock();
7         return res;
8     }
9 }

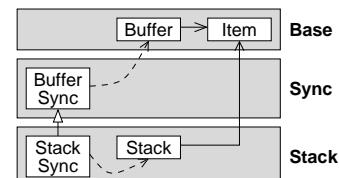
```



```

1 class Stack {
2     LinkedList list = new LinkedList();
3     void push(Item i) { list.addFirst(i); }
4     Item pop() { return (Item)list.getFirst(); }
5 }
6 abstract aspect StackSync extends BufferSync {
7     pointcut syncPC() : BufferSync.syncPC() ||
8         execution(Item Stack.pop()) ||
9         execution(void Stack.push(Item));
10 }

```



```

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

```

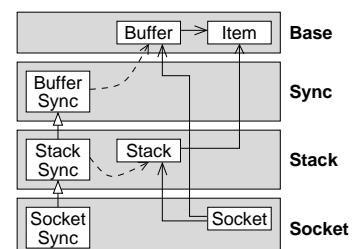


Figure 2: Four steps in the evolution of a program using aspects.

of the program explicit. Design decisions are encapsulated and can be modified in separation as well as combined and reused in different variants. A reasonable wish is to derive different customized program variants that share common features and reuse invariant code, e.g., ‘*Sync* • *Base*’ or ‘*Socket* • *Stack* • *Sync* • *Base*’.

Limited Language-Level Support for Aspect Refinement

Beside the advantages of AR, our example also demonstrates the shortcomings of AspectJ in supporting SWR:

Aspect inheritance: While aspect inheritance enables the refinement of aspects to some degree, it lacks flexibility to interchange and reuse refinements. Using aspect inheritance, a refinement (subaspect) is tied to a specific base aspect. Hence, refinements cannot be combined flexibly in different permutations for customization and adaptation purposes. For example, we are not able to derive different variants of our buffer example without changing code invasively.

Constrained aspect extension: Using traditional aspect inheritance in AspectJ, an aspect has to be declared `abstract` to be able to be refined. This means that adding a subaspect requires the programmer to modify the parent aspect. This and similar requirements² cause a fundamental problem regarding SWR: implementing an aspect in a particular development step forces the programmer to decide whether the aspect will be refined in a subsequent step. While declaring the aspect as `abstract` makes it necessary to add later at least one concrete subaspect, declaring it as concrete (without modifier) the programmer prevents the subsequent refinement of the aspect.

Advice is not first-class: Advice is one of the main mechanisms of AOP [23]. A piece of advice is invoked implicitly, i.e., it executes code when an associated pointcut matches. This prevents other advice or methods from invoking it explicitly. Since advice has no name it cannot be overridden and extended by another piece of advice, inside a subsequent refinement. This prevents reusing and customizing advice code. It has been shown that advising advice is also not an adequate solution [9].

The problems sketched above show that current AOP languages, as exemplified by AspectJ, do not support SWR appropriately. Consequently, we propose an alternative approach implementing AR along with a set of language constructs.

3 MIXIN-BASED ASPECT INHERITANCE

In order to implement AR at the language level, we introduce the notion of *mixin-based aspect inheritance*. It supports SWR at the language level by introducing mixin capabilities to aspects. Traditional aspect inheritance is not flexible enough to express refinements of aspects and their structural elements. Mixin-based aspect inheritance overcomes this limitation by allowing refinements to be freely combined and applied to a base aspect. A set of accompanying language mechanisms enables

²E.g., refining a pointcut in AspectC++ requires to declare the parent pointcut as `virtual`.

to refine all the kinds of structural elements of an aspect, in particular *pointcut refinement* and *advice refinement*, which are tailored to AspectJ-like languages.

We use the Jak language as the archetype for expressing AR at the language level. This emphasizes the uniformity of classes and aspects with respect to refinement. Figure 3 shows a synchronization aspect (Lines 1–4) and a refinement (Lines 5–14) extending the aspect. Refinements may introduce new structural elements as well as extend existing ones, as we will explain soon. They can be applied to abstract and concrete aspects as well as to other refinements. This eliminates the dilemma of anticipating subsequently applied refinements by declaring base aspects as `abstract`. Moreover, it allows a series of refinements to be applied to an aspect in different permutations.

```

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(Item Buffer.get(int)) ||
10     execution(void Buffer.put(Item));
11   Object around() : syncPC() {
12     lock(); Object res = proceed(); unlock(); return res;
13   }
14 }
```

Figure 3: Adding members and extending methods via AR.

Notably, refining aspects is conceptually different from weaving aspects. Weaving two aspects modifies the base program in two independent steps. In our example this would lead to two different instances of the aspect `Sync`. Instead, AR results in two aspect fragments that are merged via mixin composition. That is, an aspect together with all of its refinements constitutes a final aspect that is woven *once* to the base program.

Adding Members and Extending Methods.

A refinement may add new members to an aspect. As shown in Figure 3, the refinement adds a field (Line 6), a pointcut (Lines 9–10), and a piece of advice (Lines 11–13). Refinements may also extend methods to reuse existing functionality (Lines 7 and 8). A method extension usually overrides and calls the overridden method via the keyword `super`.

Pointcut Refinement

A refinement may refine the pointcuts of an aspect. Recall our example aspect that synchronizes the access to `Buffer` (Fig. 2). Two refinements (*Stack*, *Socket*) override

the pointcut `syncPC`, reuse its expression, and add new pointcut expressions that extend or constrain the set of matched join points.

In AspectJ, pointcuts have to be accessed by their fully qualified name, in our example, `BufferSync.syncPC`. Thus, the programmer is forced to hard-wire the aspect to be refined and the subaspect, which decreases reusability. Using mixin-based aspect inheritance the refined pointcut is accessed via `super` (Fig. 4). Hence, the programmer need not be aware of the actual sequence of refinements applied to a base aspect. While with standard inheritance the refinement order is fixed, with mixin-based inheritance the order is variable.

```

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

Figure 4: Altering the set of locked methods via pointcut refinement.

Semantics of pointcut refinement. The semantics of pointcut refinement is as follows: the most refined (specialized) pointcut in a series of pointcut refinements specifies when advice is executed. Which pieces of advice are executed can be defined all along the refinement chain, i.e., in every refinement of an aspect advice may be connected to a pointcut.

Figure 5 illustrates what happens when refining a pointcut (solid arrow): the most refined pointcut alters the set of matched join points (dotted arrows) and triggers the advice (dashed arrow). After refinement, the advice advises an extended set of join points (dot-dashed arrows).

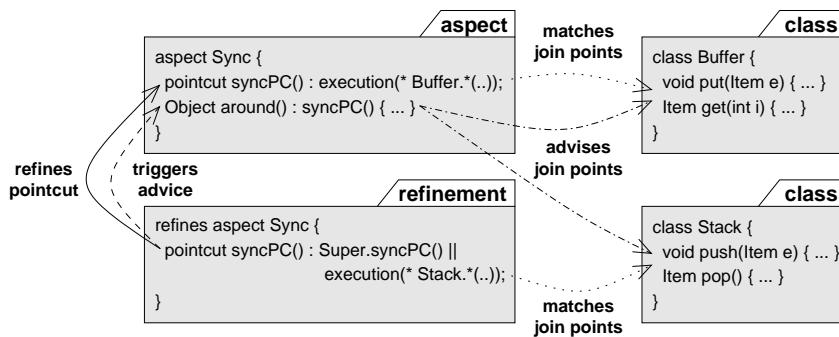


Figure 5: The most refined pointcut triggers connected advice.



Advice Refinement

Named advice. Before explaining advice refinement it is necessary to introduce *named advice*. Named advice is a named version of advice. It can be overridden and referred to from advice inside subsequent refinements. Figure 6 depicts an aspect for synchronization that contains named advice (Lines 3–5). Named advice is defined by an optional result type (`Object`), an advice type (`around`), a name (`syncMethod`), an argument list (`empty`), an exception list (`empty`), a binding to a pointcut (`syncPC`), and an advice body. One can think of named advice as a pair of an unnamed advice and a separate method (*advice method*), in which the advice calls the method. The difference to this analogy is that named advice has full access to the dynamic context (`proceed` and join point API). Though named advice can be implemented differently (cf. Sec. 5), this view is helpful for understanding the semantics of advice refinement.

```

1 aspect Sync {
2   pointcut syncPC() : execution(* Buffer.*(..));
3   Object around syncMethod() : syncPC() {
4     lock(); Object res = proceed(); unlock(); return res;
5   }
6 }
```

Figure 6: An aspect with named advice.

Refining named advice. In contrast to traditional advice, named advice can be refined in subsequent development steps. The key idea is to treat named advice in subsequent refinements similarly to a method. This is possible since named advice has a name, a result type, and an argument list. According to our analogy, an advice refinement simply refines the advice method by method overriding, i.e., by defining a method with the same name and signature as the named advice to be refined.

Figure 7 depicts an aspect that refines our synchronization aspect by extending its named advice. The refinement contains an advice method `syncMethod` (Lines 3–5) that overrides the parent named advice by counting the number of threads. The refining method must have the same name and the same signature as the parent advice. The keyword `super` is used to refer to the parent advice (Line 4).

```

1 refines aspect Sync {
2   int count = 0;
3   Object syncMethod() {
4     count++; Object res = super.syncMethod(); count--; return res;
5   }
6 }
```

Figure 7: Refining named advice.

Semantics of advice refinement. The semantics of named advice is equivalent to a virtual method, which passes the control flow to the most specialized descendant method of the inheritance chain. Mapped to advice refinement this means that, when the associated pointcut matches, the most specialized advice method is invoked. Figure 8 shows an advice method that refines a named advice (solid arrow). It is executed (dashed arrows) when the pointcut `syncPC` matches (dotted line). Programmers use `super` to navigate the refinement chain upwards. The root of the refinement chain binds the pointcut to the piece of advice.

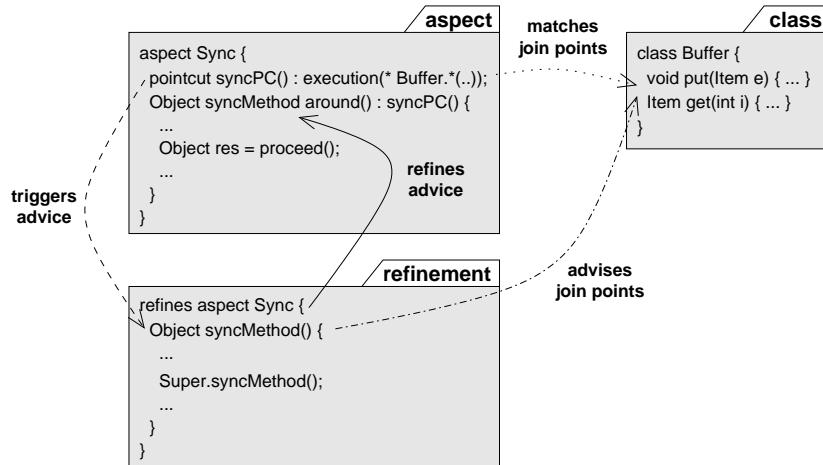


Figure 8: Semantics of advice refinement.

Accessing contextual information. An issue that we did not address yet is 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 passed explicitly via the advice interface. Should refinements have unlimited access to context information and `proceed`?

We argue that an advice refinement should only be permitted to access those pieces of context information that are passed via the advice interface, and thus are part of the advice method signature. This would preclude invoking `proceed` or accessing `thisJoinPoint` from within a refinement of an advice method. While this restriction is not necessary for our proposal, we believe it preserves simplicity and robustness of the aspect language. Furthermore, we do not allow named advice to be invoked directly by other advice and methods – such a mechanism is out of scope of this paper and addressed elsewhere (cf. Sec. 5).

Discussion

AR and its implementation via mixin-based aspect inheritance offer the following benefits: they allow a base aspect to be composed with a series of refinements, thus enabling to customize and reuse aspect code. Pointcut refinement decouples

refinements from their immediate base aspects, thus enhancing the composability of aspects and refinements. Advice refinement promotes reuse in the same way as method extension between classes. Named advice can be reused in different variants of an aspect, thus supporting the customization of advice code.

At the beginning of the paper we identified three beneficial use cases of AR, which we now want to revisit:

1. A programmer applies a refinement to adapt an aspect to the changes made to a base program. For example, Figure 9 shows an aspect that counts the updates of `Buffer` objects (Lines 1–5) and a refinement that adapts the aspect to count also executions of `clear` (Lines 6–8) that updates the `Buffer` object state as well; this is achieved by pointcut refinement (Line 7).

```

1 aspect UpdateCounter {
2   int count = 0;
3   pointcut updatePC() : execution(void Buffer.put(Item));
4   after updateCounter() returning: updatePC() { count++; }
5 }
6 refines aspect UpdateCounter {
7   pointcut updatePC() : super.updatePC() || execution(void Buffer.clear());
8 }

```

Figure 9: Counting the updates of `Buffer` objects.

2. A programmer customizes an aspect to react to a changed user requirement. Suppose a new design decision that requires our `UpdateCounter` aspect to inform a listener when an update operation was performed. Figure 10 shows a refinement of `UpdateCounter` (Fig. 9) that implements this design decision via advice refinement.

```

1 refines aspect UpdateCounter {
2   UpdateListener listener = null;
3   void setListener(UpdateListener 1) { listener = 1; }
4   void updateCounter() { super.updateCounter(); listener.notify(); }
5 }

```

Figure 10: Notify a listener when `Buffer` objects are updated.

3. A programmer decomposes an aspect to decouple it from a specific configuration of the base program. For example, Figure 11 shows an aspect that introduces a new interface `Serializable` to a set of target classes (`Buffer`, `Stack`). Figure 12 shows the result of decomposing this aspect into a base and two refinements, where each refinement introduces the interface to one target class. Before composing and compiling the final program, a programmer or a tool selects only those refinements that target classes that are actually present in the program configuration, e.g., when `Stack` is present then also the according refinement is present (Lines 5–7).

The use cases discussed have one thing in common: aspect code can be reused in different variants of a program; aspects can be customized to the specific needs of a programmer or to fit the structure of the base program.

```

1 aspect Serialization { /* ... */
2   declare parents : (Buffer || Stack) implements Serializable;
3 }
```

Figure 11: Introducing the interface `Serializable` to `Buffer` and `Stack`.

```

1 aspect Serialization { /* ... */ }
2 refines aspect Serialization {
3   declare parents : Buffer implements Serializable;
4 }
5 refines aspect Serialization {
6   declare parents : Stack implements Serializable;
7 }
```

Figure 12: Decomposed `Serialization` aspect.

It is worth noting, that without the AHEAD model for SWR, it would be difficult to realize AR. The layered AHEAD structure assigns to each aspect an enclosing feature (layer), which is associated with a development step. This information helps to organize and compose refinements and their base aspects (see [15]).

With regard to AHEAD, mixin-based aspect inheritance is a composition operator that is invoked when aspects (and their refinements) of different development steps are composed. Hence, this aspect composition operator corresponds to the class composition operator, which composes classes using mixin-based inheritance.

Tool Support

ARJ. *ARJ* is a language extension of AspectJ that supports AR. It has been implemented as a modular extension to the *abc* compiler framework [7]. It extends the *abc* parser enabling it to recognize our new syntactical elements and it adds several frontend and backend passes for implementing a syntax tree transformation. ARJ is implemented to work in concert with the AHEAD Tool Suite and Jak to integrate AR into program features: ARJ expects a feature expression in the form of an AHEAD equation file. Feature containment hierarchies contain the associated aspects, classes, and refinement files (class and aspect refinements). Further details about ARJ are explained elsewhere [15]. The current version of ARJ supports all language constructs proposed here. The compiler as well as several documents and examples can be downloaded from the ARJ Web site³.

FeatureC++. *FeatureC++*⁴ is a Jak-like language extension of C++ with aspect support [3]. The FeatureC++ compiler supports a limited form of AR. Aspects can be refined (using ‘`refines`’) by adding members, extending methods, and refining pointcuts. There is no support for named advice or advice refinement.

³http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/

⁴http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/

aspect (# pieces)	description
serialization (11)	prepares objects for serialization
responding (4)	sends replies automatically
toString (12)	introduces <i>toString</i> methods
log/debug (13)	mix of logging and debugging
pooling (3)	stores and reuses open connections
dissemination (3)	piggyback meta-data propagation
feedback (2)	generates feedback by observing peers

Table 1: Refined aspects in P2P-PL.

4 CASE STUDY

We applied AR to a non-trivial case study, a *product line for peer-to-peer overlay networks (P2P-PL)*. P2P-PL was developed as part of a comprehensive study that explored the relationship of program features and aspects [1]. Here, we discuss the results relevant for AR.

P2P-PL was implemented to experiment with advanced overlay network features such as query evaluation optimization and meta-data propagation. Hence, there was a desire for a highly customizable design that allows features to be reused in different configurations. One 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 6,426 LOC. In summary, it contains 14 aspects (406 LOC; 6%). We applied AR to 7 aspects, i.e., we decomposed each of the 7 aspects into several pieces (one base aspect and several refinements). Table 1 gives an overview of the refactored aspects and the number of refinements.

Example. Connection pooling is a mechanism to save time and resources for frequently establishing and shutting down connections. Figure 13 lists the aspect `Pooling`; it uses a `Pool` for storing references to connections (Line 2). The pointcuts `close` (Lines 3–4) and `open` (Lines 5–6) match the join points that are associated with shutting down and opening connections. Named advice `putPool` (Lines 7–9) intercepts the shutdown process of connections and instead stores the associated `ClientConnection` objects in a `Pool` object. Named advice `getPool` (Lines 10–13) recovers open connections (if available) and passes them to clients that request a new connection. This aspect makes use of the built-in pointcut `this` to limit the advised calls to those that originate from `MessageSender` objects.

We refined the aspect `Pooling` twice, as shown in Figure 14. The first refinement (Lines 1–4) refines the pointcut `open` to limit the matched joint points to those that occur in the control flow of `Peer`. This excludes join points associated with helper and experimentation classes that use `ClientConnection` objects as well. Pointcut refinement decouples the aspect refinement from a fixed base aspect and thus increases the flexibility to combine this refinement with other refinements.

The second refinement is more sophisticated (Lines 5–12). It refines both pieces of advice (`putPool`, `getPool`) to guarantee thread safety. Since the pooling activities are implemented via named advice, a refinement can add synchronization code.

```

1 aspect Pooling {
2     static Pool pool = new Pool();
3     pointcut close(ClientConnection con) :
4         call(void ClientConnection.close()) && target(con) && this(MessageSender);
5     pointcut open(ClientSocket socket) :
6         call(ClientConnection.new(ClientSocket)) && args(socket) && this(MessageSender);
7     void around putPool(ClientConnection con) : close(con) {
8         pool.put(con);
9     }
10    ClientConnection around getPool(ClientSocket socket) : open(socket) {
11        if(!pool.contains(socket)) return proceed(socket);
12        else return (ClientConnection)pool.get(socket);
13    }
14 }

```

Figure 13: Connection pooling aspect (excerpt).

```

1 refines aspect Pooling {
2     pointcut open(ClientSocket sock) : super.open(sock) &&
3         cflow(execution(void Peer.main(..)));
4 }
5 refines aspect Pooling {
6     Object putPool(ClientConnection con) {
7         synchronized(pool) { return super.putPool(con); }
8     }
9     ClientConnection getPool(ClientSocket sock) {
10        synchronized(pool) { return super.getPool(sock); }
11    }
12 }

```

Figure 14: Encapsulating design decisions using AR.

Other refinements. In summary, we applied the notion of AR to 7 of 14 aspects of P2P-PL (cf. Tab. 1). On average, there were 7 refinements per base aspect and 1/2 of all aspects were decomposed via AR. While the predominant role of AR was to add new structural elements, i.e., advice, pointcuts, methods, and fields, we refined 3 named advice and 1 pointcut.

The application of AR increased the total number of features in P2P-PL considerably. However, the fine-grained decomposition of aspects (which results in 48 refinements applied to 7 aspects) did not only structure the design and implementation of P2P-PL, but it also increased the configuration space, i.e., the tailored variants of P2P systems that can be derived by the configuration process. For example, the aspect `Serialization` shown in Figure 12 has as many variants as different sets of target classes are possible in P2P-PL (theoretically 2^{10}). The aspect `Pooling` comes in fewer variants (4) because it has only 2 optional refinements, which can be combined freely (2^2).

Beside an improvement in customizability we were able to reuse aspect code amongst different variants of P2P-PL. In our study, all derivable variants of aspects share common functionality, thus reusing aspect code. For example, each of the 4 `Pooling` variants reuse code of the base aspect and of possibly another refinement. On average, each variant of each of the 7 decomposed aspects reuses code of 1.5 aspects and refinements. This is because, for most aspects, all variants can be freely

combined, i.e., refinements are optional and can be applied standalone, in combination with *some* other refinements, or in combination with *all* other refinements.

Finally, we did not find many use cases for advice refinement (3 \times) and pointcut refinement (1 \times). We believe that this small number originates from the refactoring approach we chose, i.e., we decomposed each considered aspect retroactively into a base aspect and several refinements. Developing software from scratch with aspects and refinements in mind enables one to plan aspect reuse more systematically.

Discussion and Programming Guidelines

In our study we identified two main use cases of AR (cf. Sec. 2). First, we decoupled 4 aspects from a particular set of classes to be extended/ ADVISED, which maps to use case 3 (decoupling an aspect from a specific program configuration). For that, we decomposed aspects into several pieces to enable the programmer to combine these pieces in different combinations. For example, we decomposed the aspect **Serialization** to be able to choose only those pieces for adding serialization functionality that are actually needed in a particular P2P-PL configuration. This allows us to use the aspect in varying contexts, i.e., in different configurations of P2P-PL that contains different sets of classes to be serialized.

Second, we encapsulated the effects of different design decisions of 3 aspects into refinements, which maps to use case 2 (tailoring an aspect to a changed requirement). Decomposing aspects along design decisions facilitates customization, i.e., 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 differing requirements. For example, we decomposed the aspect **Pooling** into 3 pieces that encapsulate design decisions such as synchronization. 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 customization and tailoring of aspects to different P2P-PL configurations and to different requirements, e.g., performance.

Finally, we did not find applications for use case 1 (adapting aspects to changes in a base program). This is not surprising since this situation occurs typically when a program evolves, which was not covered by our case study.

5 RELATED WORK

Aspect refinement and AHEAD. The idea of AR emerged from our prior work on aspect-oriented and feature-oriented product lines and *AHEAD* [5]. *Aspectual mixin layers (AMLs)* integrate aspects and features in the sense of the AHEAD model [5, 1]. Aspect refinement based on mixin-based aspect inheritance enhances AMLs toward a unified integration of features and aspects with regard to SWR.

While in prior work we outlined just the possibility of refining aspects and examined the consequences for the architectural model of AHEAD [4, 5], in this paper we focused on the language level support for AR, its implementation in the form of

a fully functional compiler, and a non-trivial case study. We have shown how the idea of AR unifies the refinement of classes and aspects at the language level.

Implementation of aspect refinement. Beside mixin-based aspect inheritance, a further possibility arises to refine aspects: an aspect could be refined itself via advice and inter-type declarations of another aspect. In this case aspects themselves are part of a base program and the programmer has the choice to refine them via Jak-like refinements or via aspect weaving. However, it has been observed that advising advice can lead to logical errors and infinite loops [9].

AR is closely related to *superimposition* [10]. However, AR superimposes aspects, not just superimposes object-oriented structures using aspects [30].

Higher-order functions, pointcuts, and advice. Aspects are refinements and can be modeled as functions [21, 20, 6]. It is interesting, that in this view, AR is related to *higher-order functions*. A higher-order function expects a function as input and returns another function as output. Since aspects can be modeled as functions a refinement of an aspect can be understood as a function that applies to a function, which is a higher-order function, e.g., $R[A](P)$, where P is a program, A is an aspect, and R is a refinement of A . It remains open how high-order functions fit with current algebraic models [21, 20, 6].

Tucker and Krishnamurthi integrate advice and pointcuts into languages with higher-order functions and model them as first-class entities [32]. This way, it becomes possible to implement higher-order pointcuts and advice. Pointcuts can be passed to other pointcuts as arguments. Thus, they can be modified, combined, and extended. In this respect, our approach of aspect and pointcut refinement is similar. We can combine, modify, and extend pointcuts by applying subsequent refinements.

Due to the opportunity of refining named advice, we can also modify and extend advice using subsequent advice. This corresponds to higher-order advice that expects a piece of advice as input and returns a modified piece of advice. Named advice can be passed to other advice – usually to advice that refines other advice. Thus, refining advice is similar to passing a piece of advice to higher-order advice.

Unification of advice and method. Using the annotation-based programming style of AspectJ, aspects are implemented as classes and advice is implemented as method and declared as such via annotation. In this programming style advice is already named and can be refined by method overriding. However, it is not obvious how this relates to other mechanisms for refinement, e.g., pointcut refinement.

Rajan and Sullivan propose *classpects* that combine capabilities of aspects and classes [28]. A classpect associates a method that is executed for advising a particular join point to each piece of advice. Moreover, classpects unify aspects and classes with respect to instantiation. Since advice is implemented via methods, it could be refined. However, the authors of classpects do not make a statement about the relationship to SWR nor about its consequences.

Aspects and genericity. Several recent approaches enhance aspects with genericity, e.g., *Sally* [14], *Generic Advice* [19], *LogicAJ* [17], *Framed Aspects* [22]. This improves reusability of aspects in different application contexts. Aspect refinement and mixin-based aspect inheritance provide an alternative way to customize aspects, i.e., by composing a base aspect and a series of refinements. However, ideas on generic aspects can be combined with our compositional approach, just as *generic feature modules* combine features with generics [2].

AspectJ design patterns. Hanenberg et al. discuss the benefits of inheritance in the context of AOP [14]. 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. AR can enhance these patterns by simplifying the composition of aspects. The pattern *template advice* can be improved by using named advice because it becomes possible to refine advice directly.

Feature-optionality problem. In FOP, features may interact with other features that are optional [27, 18]. 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* [27]. Lifters encapsulate those pieces of code that depend on (and interact with) other features. When composing a program from 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 an algebraic foundation for this methodology [18].

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

6 CONCLUSION

AR is the incarnation of SWR in AOP. It follows directly from the integration of aspects and stepwise development methodology of AHEAD. AR unifies classes and aspects with respect to subsequent refinement. We have illustrated three use cases where AR improves reuse and customization of aspect code. To introduce the principles of SWR at the programming language level, we proposed mixin-based aspect inheritance and a set of accompanying language constructs that facilitate SWR: pointcut refinement, named advice, and advice refinement.

To underpin our proposal, we developed a fully functional compiler on top of AspectJ. The compiler implements all proposed 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 medium-sized software project. Our study indicated that the proposed language mechanisms facilitate aspect 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.

Finally, AR as embodied in our ARJ compiler can be understood as an AHEAD composition operator for aspects, thus queuing in a long line of research on program synthesis and SWR [8].

Acknowledgements. We thank Don Batory, William Cook, Christian Lengauer, and Roberto Lopez-Herrejon for their insightful comments on earlier drafts of this paper. This work was sponsored in parts by the German Research Foundation (DFG), project number SA 465/32-1.

REFERENCES

- [1] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 59–68. ACM Press, 2006.
- [2] S. Apel, M. Kuhlemann, and T. Leich. Generic Feature Modules: Two-Staged Program Customization. In *Proceedings of the International Conference on Software and Data Technologies*, pages 127–132. INSTICC Press, 2006.
- [3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 125–140. Springer, 2005.
- [4] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounded Quantification in Incremental Designs. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 796–804. IEEE Computer Society, 2005.
- [5] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering*, pages 122–131. ACM Press, 2006.
- [6] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proceedings of the ECOOP Workshop on Aspects, Dependencies, and Interactions*, pages 1–9. Computing Department, Lancaster University, 2006.
- [7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An Extensible AspectJ Compiler. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

- [8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [9] E. Bodden, F. Forster, and F. Steimann. Avoiding Infinite Recursion with Stratified Aspects. In *Proceedings of the International Net.ObjectDays Conference*, pages 49–64. Gesellschaft für Informatik, 2006.
- [10] L. Bouge and N. Francez. A Compositional Approach to Superimposition. In *Proceedings of the International Symposium on Principles of Programming Languages*, pages 240–249. ACM Press, 1988.
- [11] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 303–311. ACM Press, 1990.
- [12] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.
- [13] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [14] S. Hanenberg, A. Schmidmeier, and R. Unland. AspectJ Idioms for Aspect-Oriented Software Construction. In *European Conference on Pattern Languages of Programs*, pages 617–644. Universitätsverlag Konstanz, 2003.
- [15] C. Kästner, S. Apel, and G. Saake. Implementing Bounded Aspect Quantification in AspectJ. In *Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 111–122. School of Computer Science, University of Magdeburg, 2006.
- [16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
- [17] G. Kniesel and T. Rho. A Definition, Overview and Taxonomy of Generic Aspect Languages. *L'Objet*, 11(3):9–39, 2006.
- [18] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121. ACM Press, 2006.
- [19] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, volume 3286 of *LNCS*, pages 55–74. Springer, 2004.

- [20] R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.
- [21] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 68–77. ACM Press, 2006.
- [22] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *Proceedings of the International Conference on Software Reuse*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.
- [23] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 2–28. Springer, 2003.
- [24] T. Mens, K. Mens, and T. Tourwé. Aspect-Oriented Software Evolution. *ERCIM News*, 1(58):36–37, 2004.
- [25] C. Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.
- [26] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):264–277, 1979.
- [27] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.
- [28] H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *Proceedings of the International Conference on Software Engineering*, pages 59–68. ACM Press, 2005.
- [29] V. Rajlich. Changing the Paradigm of Software Engineering. *Communications of the ACM*, 49(8):67–70, 2006.
- [30] M. Sihman and S. Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5):529–541, 2003.
- [31] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.
- [32] D. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 158–167. ACM Press, 2003.
- [33] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.