Accepted for Publication in the Generative Programming and Component Engineering (GPCE) Conference, October 2006

When to Use Features and Aspects? A Case Study

Sven Apel
Department of Computer Science
University of Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu

Abstract

Aspect-Oriented Programming (AOP) and *Feature-Oriented Programming (FOP)* are complementary technologies that can be combined to overcome their individual limitations. *Aspectual Mixin Layers (AML)* is a representative approach that unifies AOP and FOP. We use AML in a non-trivial case study to create a product line of overlay networks. We also present a set of guidelines to assist programmers in how and when to use AOP and FOP techniques for implementing product lines in a stepwise and generative manner.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Experimentation

Keywords Feature-oriented programming, aspect-oriented programming, software product lines, collaboration-based design, stepwise development, separation of concerns, crosscutting

1. Introduction

Two advanced programming paradigms are gaining attention in the overlapping fields of program generation, product lines, and *stepwise development (SWD)*. *Feature-Oriented Programming (FOP)* [9] aims at large-scale compositional programming and feature modularity in product lines. *Aspect-Oriented Programming (AOP)* [23] focuses on crosscut modularity in complex software. Several recent studies have observed that both paradigms have limitations [33, 25, 30, 5], where the weakness of one maps roughly to the strength of the other. Hence, the two paradigms are not competitive and can profit from each other [5]. Recent work also has suggested that both paradigms be combined to exploit their synergistic potential [33, 25, 20, 5].

In this paper, we use *Aspectual Mixin Layers (AML)* as a representative approach that integrates AOP and FOP [5]. AML supports collaboration-based designs, mixin composition, aspect weaving, and refinement of aspects to decompose and structure software by features. Further, we present a non-trivial case study using AML: a product line of peer-to-peer overlay networks (*P2P-PL*) whose software products demand a high degree of customizability, reusability, and evolvability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010. . . $5.00.

Although AOP and FOP are complementary, their technologies overlap. FOP uses traditional object-oriented mechanisms (e.g., mixins) to implement features, whereas AOP (as exemplified by AspectJ) uses introductions, pointcuts, and advice. An interesting and fundamental question is: when should a programmer use feature-oriented mechanisms (i.e., object-oriented concepts like classes and mixins) and aspect-oriented mechanisms (i.e., introductions, pointcuts, and advice) to implement features of a product line? The results of our paper are a set of guidelines and supporting statistics to answer this question. We begin with a summary of FOP, AOP, and AML.

2. Background

2.1 Feature-Oriented Programming

FOP studies the modularity of *features* in product lines, where a feature is an increment in program functionality [9]. The idea of FOP is to synthesize software (individual programs) by composing features (a.k.a. *feature modules*). Typically, features refine the content of other features in an incremental fashion. Hence, the term *refinement* refers to the set of changes a feature applies to a code base. Adding features incrementally, called *stepwise refinement*, leads to conceptually layered software designs. For simplicity, we use the terms feature and feature module synonymously.

Mixin layers is one approach to implement features [37, 9]. The basic idea is that features are seldomly implemented by single classes (or aspects). Typically, a feature implements a *collaboration* [41], which is a collection of roles represented by mixins that cooperate to achieve an increment in program functionality. FOP aims at abstracting and explicitly representing such collaborations. Hence, it stands in the long line of prior work on object-oriented design and role modeling [38].

A mixin layer is a module that encapsulates fragments of several different classes (roles) so that all fragments are composed consistently. Figure 1 depicts a stack of three mixin layers ($L_1 - L_3$) in top down order. These mixin layers crosscut multiple classes ($C_A - C_C$). White boxes represent classes or mixins; gray boxes denote the enclosing feature modules; filled arrows refers to *mixin-based inheritance* [13] for composing mixins.

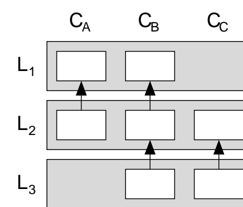

Figure 1. Stack of three mixin layers.

2.2 Aspect-Oriented Programming

AOP aims at separating and modularizing crosscutting concerns [23]. Using object-oriented mechanisms, crosscutting concerns result in tangled and scattered code [23, 18]. The idea behind AOP is to implement crosscutting concerns as *aspects* where the core (non-crosscutting) features are implemented as components. Using *pointcuts* and *advice*, an aspect weaver glues aspects and components at *join points*. Pointcuts specify sets of join points in aspects and components, advice defines code that is applied to (or executed at) these points, and introductions (a.k.a. *inter-type declarations*) inject new members into classes. With aspects, a programmer is able to refine a program coherently at multiple join points. Typically, aspects introduce new members to existing classes and extend existing methods. Figure 2 shows two aspects (A_1, A_2) that extend three classes at multiple join points (dashed arrows denote aspect weaving) in classes ($C_A - C_C$).

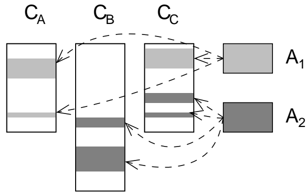

Figure 2. Two aspects extend three classes.

2.3 Comparison

The essential differences between FOP and AOP technologies are their emphasis on different types of crosscuts [5].

2.3.1 Homogeneous and Heterogeneous Crosscuts

Two different kinds of crosscuts have been identified in the AOP literature. *Homogeneous crosscuts* refine multiple join points with a single piece of advice. *Heterogeneous crosscuts*, in contrast, refine multiple join points each with different pieces of advice [17]. Recall that a feature typically implements a collaboration involving multiple classes. This requires a feature to introduce a set of new classes, introduce a set of new members to existing classes, and to refine existing methods. Method refinement in features is usually accomplished using heterogeneous crosscuts. In contrast, aspects perform well in refining a set of methods using one coherent advice, thus, modularizing a homogeneous crosscut and avoiding gross code replication.

Although both approaches are able to implement the crosscuts of the other, they cannot do so elegantly [33, 5]. Consider a synchronization feature, which is a homogeneous crosscut. To emulate a homogeneous aspect, a mixin layer would have to explicitly refine each target method, and the code of each method refinement would be identical (leading to code replication).

Conversely, an aspect may implement a collaboration of classes by applying a set of introductions and advice. It has been argued that not expressing the collaboration explicitly in terms of roles or classes decreases program comprehensibility [39, 33, 12, 5]. This is because the programmer cannot recognize the original class structure of the base program. A further argument is that aspects lack scalability with respect to large-scale features: suppose a collaboration consists of many roles, e.g., a data management feature. Merging all participating roles (storage structures, file access, indexes, transactions, etc.) in one or more aspects[1] flattens the inherent object-oriented structure of the collaboration, obscures the intent of the programmer, and results in a program that is difficult to understand [39, 33].

[1] In Section 5, we address the issue of implementing each individual role as aspect.

2.3.2 Static and Dynamic Crosscuts

Features and aspects may extend the structure of a base program statically (*static crosscuts*), i.e., by injecting new members. Moreover, both are able to introduce new super-classes and interfaces to existing classes. Additionally, features are able to encapsulate and introduce new classes. Aspects are not able to introduce independent classes – at least not as part of an encapsulated feature. The reason for that is that aspects have no architectural model.[2]

A *dynamic crosscut* implements a refinement in terms of the dynamic program semantics [31]. By using feature modules one has only the limited abilities of method overriding to intercept method executions. Aspects provide a more sophisticated set to refine a base program based upon its execution, e.g., mechanisms for tracing the dynamic control flow.

2.4 Symbiosis of Aspects and Features

Aspects and features in their current incarnation are intended for solving problems at different levels of abstraction [33, 25, 5]. Whereas aspects in AspectJ act on the level of classes and objects in order to modularize crosscutting concerns, features act on an architectural level. That is, a feature decomposes an object-oriented architecture into a composition of collaborations.

A next logical step is to decompose and structure *aspect-oriented architectures* (i.e., object-oriented architectures with aspects) into features. Figure 3 shows on the left an aspect-oriented architecture and on the right features that decompose and structure this architecture. With this decomposition, a feature encapsulates fragments of classes *and* aspects that collaborate together to implement an increment in program functionality. Note that the original aspect was split into two pieces (a base and a subsequent refinement). In Section 2.6, we address this issue in more depth.

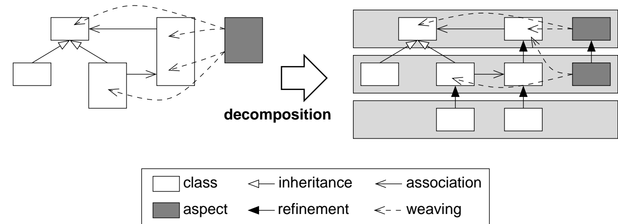

Figure 3. Feature-driven decomposition of an aspect-oriented architecture (features are depicted light-gray).

2.5 Aspectual Mixin Layers

Aspectual Mixin Layers integrate AOP and FOP. AML extends the notion of mixin layers by encapsulating both mixins and aspects (see Fig. 4). That is, an AML encapsulates both collaborating classes *and* aspects that contribute to a feature. An AML may refine a base program in two ways: (1) by using common mixin-composition or (2) by using aspect-oriented mechanisms, in particular pointcuts and advice. Probably the most important contribution of AML is that programmers may choose the appropriate technique – mixins or aspects – that fits a given problem best.

[2] While it is correct that one can just add another class to an environment, e.g., using AspectJ, this is at the tool level, and is not at a model level. The programmer has to build his own mechanisms (outside of the tool) to implement feature modularity [29].

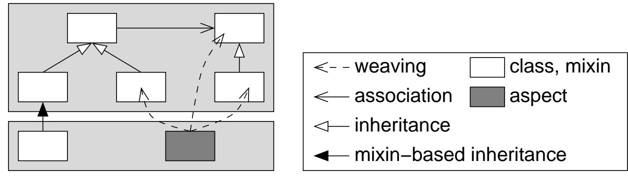

Figure 4. Aspectual Mixin Layers.

2.6 Aspect Refinement

Aspect Refinement (AR) is the incarnation of *stepwise development (SWD)* in AOP [4, 3]. Although the notion of AR does not depend on AML, it profits from the integration of aspects into features, and therewith into layered architectures. Having this, it is natural to refine aspects in subsequent features, too. This allows for reusing, refining, and evolving aspect implementations – true to the motto of SWD. Refining an aspect means adding new members and extending existing members. To support AR at the language level, the notion of *mixin-based aspect inheritance* has been proposed. It adds the notions of *pointcut refinement*, *named advice*, and *advice refinement* – all based on mixin capabilities [3]. Figure 5 depicts an aspect included in an AML that is subsequently refined in order to advise an extended set of join points. Note that the refinement of the aspect is part of an AML as well. AR allows every piece of an aspect to be reused, refined, and evolved [5, 4, 3].

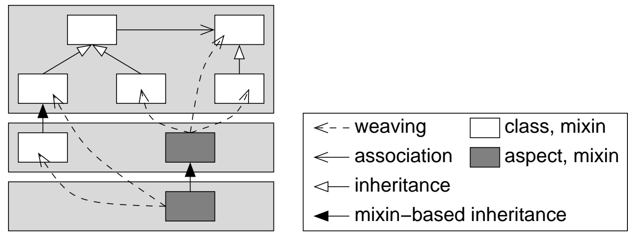

Figure 5. Aspect Refinement.

3. Case Study

3.1 A Product Line for Peer-to-Peer Overlay Networks

As a case study, we use a product line for *peer-to-peer overlay networks (P2P-PL)* [14, 2, 1]. Besides the basic functionality as routing and data management, P2P-PL supports several advanced features, e.g., query optimization based on flexible routing path selection [2], meta-data propagation for the continuous exchange of control information among peers [14], incentive mechanisms to counter peers that misbehave (*free riders*) [11]. Numerous experiments concerning those features demand plenty of different configurations to make statements about their specific effects, their variants, and combinations.

P2P-PL has a fine-grained architecture. It follows the principle of evolving a design by starting from a minimal base and applying incrementally minimal refinements to implement design decisions [34]. In its current state, it consists of 113 end-user visible features, categorized into several sub-domains, such as hashing and overlay topology.

We implemented the features of P2P-PL mainly using the *AHEAD tool suite (ATS)* [9]. We used ATS for implementing and composing traditional mixin layers, i.e., collaborations of Java classes and refinements. To implement AML, we integrated aspects into mixin layers. Aspects were implemented using *ARJ*[3], an extended AspectJ compiler that supports AR [3]. This combination of ATS and ARJ enabled us to implement and compose AMLs.

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

3.2 Aspectual Mixin Layers in P2P-PL

An *end-user visible* feature is an increment in program functionality that users feel is important in describing and distinguishing programs within a product line. 14 of the 113 end-user visible features of P2P-PL (12%) used aspects (see Tab. 1); the remaining 99 features were implemented as traditional mixin layers. In the following, we explain two representative examples.

aspect	**description**
responding	sends replies automatically
forwarding	forwards messages to adjacent peers
message handler	base aspect for message handling
pooling	stores and reuses open connections
serialization	prepares objects for serialization
illegal parameters	discovers illegal system states
toString	introduces *toString* methods
log/debug	mix of logging and debugging
dissemination	piggyback meta-data propagation
feedback	generates feedback by observing peers
query listener	waits for query response messages
command line	command line access
caching	caches peer contact data
statistics	calculates runtime statistics

Table 1. Aspectual Mixin Layers used in P2P-PL.

Feedback generator. The feedback generator feature is part of an incentive mechanism for penalizing free riders – peers that profit of the P2P network but do not contribute adequately [11]. A feedback generator feature, on top of a peer implementation, identifies free riders by keeping track whether other peers respond adequately to messages. If that is not the case, an observed peer is considered a free rider. Specifically, the generator observes the traffic of outgoing and incoming messages and keeps track of which peers have responded in time to posted messages. The generator creates positive feedback to reward cooperative peers and negative feedback to penalize free riders. Feedback information is stored in a feedback repository and is passed to other (trusted) peers to inform them about free riding. Based on the collected information, a peer judges the cooperativeness of other peers. Free riders are subsequently ignored and only cooperative peers profit from the overall P2P network [11].

The implementation of the feedback generator crosscuts the message sending and receiving features. As Figure 6 shows, the feedback generator AML contains an aspect (dark-gray) and introduces four new classes for feedback management. Additionally, it refines the peer abstraction (by mixin composition) so that each peer owns a log for outgoing queries and a repository for feedback information.

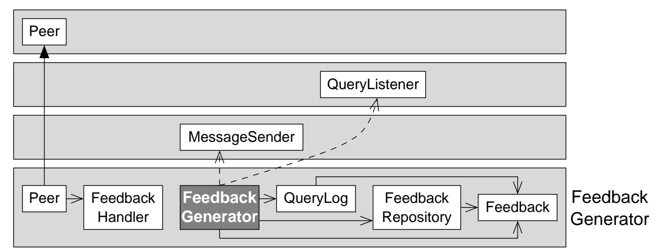

Figure 6. Feedback generator AML.

While the feedback generator feature is heterogeneous, it relies on dynamic context information. Figure 7 lists an excerpt from the above mentioned aspect. The first advice refines the message sending mechanism by registering outgoing messages in a query log (Lines 2-7). It is executed only if the method *send* was called in the dynamic control flow of the method *forward*. This is expressed using the *cflow* pointcut (Line 5) and avoids advising unintended calls to *send*, which are not triggered by the message forwarding mechanism. The second advice intercepts the execution of a query listener task for creating feedback (Lines 8-10).

```
1  aspect FeedbackGenerator { ...
2    after(MessageSender sender, Message msg, PeerId id) :
3      target(sender) && args(msg, id) &&
4      call(* MessageSender.send(Message, PeerId)) &&
5      cflow(execution(* Forwarding.forward(..))) &&
6      if(msg instanceof QueryRequestMessage)
7      { /* ... */ }
8    after(QueryListener listener) : target(listener) &&
9      execution(void QueryListener.run())
10     { /* ... */ }
11 }
```

Figure 7. Feedback generator aspect (excerpt).

Figure 8 lists the refinement to the peer class implemented as mixin. It adds a feedback repository (Line 2) and a query log (Line 3). Moreover, it refines the constructor by registering a feedback handler in the peer's message handling mechanism (Lines 4-7). For simplicity, we omit presenting the remaining code for feedback management.

```
1 refines class Peer {
2   FeedbackRepository fr = new FeedbackRepository();
3   QueryLog ql = new QueryLog();
4   refines Peer() {
5     FeedbackHandler fh = new FeedbackHandler(this);
6     this.getMessageHandler().subscribe(fh);
7   }
8 }
```

Figure 8. Feedback management refinement of the class *Peer*.

In summary, the feedback generator AML encapsulates four classes that implement the basic feedback management, an aspect that intercepts the message transfer, and a mixin that refines the peer abstraction. Omitting AOP mechanisms would result in code tangling and scattering since the retrieval of dynamic context information crosscuts other features, e.g., clients of the message forwarding mechanism. On the other hand, implementing this feature as one standalone aspect would not reflect the structure of the P2P-PL framework that includes feedback management. All would be merged in one or more aspect(s) that would both decrease program comprehension. Our AML encapsulates all contributing elements coherently as a collaboration that reflects the intuitive structure of the P2P-PL framework we had in mind during its design.

Connection pooling. The connection pooling feature is a mechanism to save time and resources for frequently establishing and shutting down connections. To integrate connection pooling into P2P-PL, we implemented a corresponding AML. Figure 9 shows this AML consisting of an aspect and a pool class. The aspect intercepts all calls that create and close connections.[4] The pool stores open connections.

Figure 10 lists the pooling aspect; 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 to shutting down and opening connections. Named advice[5] *putPool* (Lines 7-9) intercepts the shutdown process of connections and instead stores them in the pool. Named advice *getPool* (Lines 10-13) recovers open connections (if available) and passes them to clients that request a new connection. This crosscut is heterogeneous because it advises creating connections (Lines 7-9) differently than closing connections and (Lines 10-13); but it is also homogeneous because advice advises a set of join points that are related, e.g., all client-side calls to *close* (Lines 7-9).

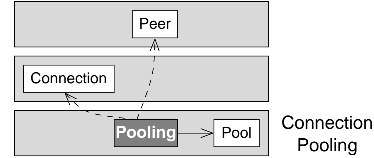

Figure 9. Connection pooling AML.

```
1  aspect Pooling {
2    static Pool pool = new Pool();
3    pointcut close(Connection con) :
4      call(void Connection.close()) && target(con);
5    pointcut open(SocketAddr sa) :
6      call(Connection Peer.connect(..)) && args(sa);
7    Object around putPool(Connection con) : close(con) {
8      pool.put(con); return null;
9    }
10   Connection around getPool(SocketAddr sa) :open(sa) {
11     if(pool.empty(sa)) return proceed(sa);
12     return (Connection)pool.get(sa);
13   }
14 }
```

Figure 10. Connection pooling aspect (excerpt).

Implementing this feature using only mixins would result in redundant code. This is because for each method that is associated with opening and closing connections we would have to implement a distinct method extension. Furthermore, we implemented the pool not as a nested class within the aspect to emphasize that it is regular part of the P2P-PL. We consider it as part of the collaboration of artifacts that implement the feature. Subsequent refinements may extend and modify it.

3.3 Aspect Refinement in P2P-PL

Features can interact. Feature interaction in FOP can sometimes take the following form: when two features F and G are present in a system, a third feature or refinement FG – sometimes called a *derivative* [35, 26] – is required to alter the behavior of F or G (or both). Derivatives are features that are not end-user visible; they are needed *only* when a particular combination of interacting end-user features are present in a target system, e.g., only if *both* F and G are present in a target system will feature/derivative FG be needed. Derivatives arose in 7 of the 14 AMLs that used aspects. That is, we decomposed each of the 7 AMLs with aspects into a base AML and multiple derivatives, where each derivative is itself implemented by an AML. We explain two representative examples below.

Serialization. The *Serialization* feature consists only of one aspect. Figure 11 depicts this aspect for a fully-configured P2P system. It enumerates a list of *declare parent* statements that add the

[4] Note that this is not ideally visualized because the calls are intercepted only on the client / caller side.

[5] Named advice assigns a name to advice for enabling subsequent refinement [3].

interface *Serializable* to a set of target classes.[6] The key thing to note here is that the *list of declared parents depends on the set of features that are in a P2P system*. This means that if the feedback generator feature is not present in a target P2P system, the *declare parents: Feedback* statement in Figure 11 would need to be removed from the *Serialization* aspect, otherwise a warning would be reported (because there would be no *Feedback* class). [7] The same holds for the *PeerId*, *Contact*, *Key*, and *DataItem* features. Thus, the definition of the *Serialization* aspect depends on other features that are present in a target system. We can model the synthesis of a customized *Serialization* aspect by refining a base aspect and encapsulating these refinements in derivatives. That is, we apply AR to break apart the *Serialization* aspect into smaller pieces – a base aspect + derivatives – to synthesize a system-specific *Serialization* aspect.

```
1 aspect Serialization {
2   declare parents : Message implements Serializable;
3   declare parents : PeerId implements Serializable;
4   declare parents : Contact implements Serializable;
5   declare parents : Key implements Serializable;
6   declare parents : DataItem implements Serializable;
7   declare parents : Feedback implements Serializable;
8   ...
9 }
```

Figure 11. Serialization aspect (excerpt).

Figure 12 lists the decomposed *Serialization* aspect, i.e., a basic *Serialization* aspect and several refinements (merged in one listing). Each refinement introduces the *Serializable* interface to only one target class. This enables programmers to choose only those pieces (derivatives) that are required for a particular configuration of P2P-PL. For example, the refinement that targets the class *Feedback* (Lines 10-12) is only included in a program if the feedback generator feature is added as well. How fine-grained such decomposition should be depends on the flexibility of composing end-user visible features. In P2P-PL, we split the compound *Serialization* feature into 12 pieces (one base and 11 refinements).[8]

```
1 aspect Serialization {
2   declare parents : Message implements Serializable;
3 }
4 refines aspect Serialization {
5   declare parents : PeerId implements Serializable;
6 }
7 refines aspect Serialization {
8   declare parents : Contact implements Serializable;
9 }
10 refines aspect Serialization {
11   declare parents : Feedback implements Serializable;
12 } ...
```

Figure 12. Refactored serialization aspect (excerpt).

[6] This particular aspect could also be implemented by enumerating all target classes in a logical expression, e.g., *declare parents : (Message || PeerId || ...) implements Serializable*.

[7] Not all aspect compilers will issue warnings; some may issue errors when non-existent classes are referenced. Our use of derivatives avoids compiler warnings/errors at the expense of imposing more structure on synthesized P2P-PL programs.

[8] We could have implemented the Serialization feature using mixins, as they too provide the ability to add 'implements interface' clauses to existing classes. This design would have exactly the same AML structure with the same set of derivatives.

Connection pooling. Figure 10 (shown earlier) depicted the *Pooling* aspect for a fully-configured P2P system. By removing features from a fully-configured system, the definition of the *Pooling* aspect changes. Using AR we decomposed the *Pooling* aspect into a base and multiple derivatives.

```
1 refines aspect Pooling {
2   pointcut open(SocketAddr sock) : super.open(sock) ||
3     execution(* TCP.getConnection(..));
4 }
5 refines aspect Pooling {
6   boolean putPool(Connection con) {
7     synchronized(pool) { return super.putPool(con); }
8   }
9   Connection getPool(SocketAddress adr) {
10    synchronized(pool) { return super.getPool(adr); }
11  }
12 }
13 refines aspect Pooling {
14   boolean putPool(Connection con) {
15     boolean res = true;
16     if(TCP.calcAverageThroughput(con) > MIN_TP)
17       res = super.putPool(con);
18     return res;
19   }
20 }
```

Figure 13. Encapsulating design decisions using AR.

Figure 13 depicts the three refinements (merged in one listing). The first (Lines 1-4) refines the pointcut *open* to match also connection requests not addressed to *Peer*, in our example addressed to a different network component *TCP*. The notion of *pointcut refinement* decouples the aspect from a fixed parent aspect and therefore increases the flexibility to combine this refinement with other refinements (see [3]).

The second refinement is more sophisticated (Lines 5-12). It refines both advice (*putPool*, *getPool*) with synchronization code to guarantee thread safety. Since the pooling activities are implemented via named advice, this refinement adds synchronization code. Refining named advice is similar to refining conventional methods (see [3]).

The third refinement (Lines 13-20) selects only those connections for pooling that satisfy specific network properties, i.e., the data throughput. It extends the *putPool* advice by code for analyzing the network traffic.

These three refinements are derivatives that are added depending on the presence of end-user visible features. The first refinement is added if the *TCP* component is present, the second refinement is added if a multi-threaded peer feature is used, and the third is added if a reliable communication feature is used.

3.4 Statistics on FOP and AOP Mechanism Usage

In this section, we present statistics on how and when FOP and AOP mechanisms were used in implementing our P2P product line. These statistics provide insight into design guidelines on mechanism usage, which we discuss in detail in Section 4.

3.4.1 Statistics on Used AOP and FOP Mechanisms

We collected the following statistics: (1) the number of implementation mechanisms used, (2) the *lines of code (LOC)* associated with these mechanisms, and (3) the LOC associated with introductions (static crosscuts) and refining methods (dynamic crosscuts).

Number of classes, mixins, and aspects. The base P2P application contains only 2 classes. A fully-configured P2P system consists of 127 classes. Thus, refining the base application into a fully-configured system required the incremental introduction of 125 classes. In addition to class introductions, there were 120 class

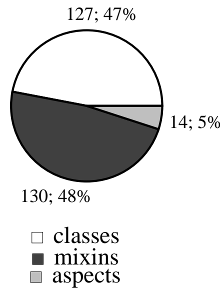

Figure 14. Classes, mixins, and aspects (number).

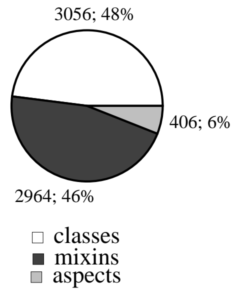

Figure 15. Classes, mixins, and aspects (LOC).

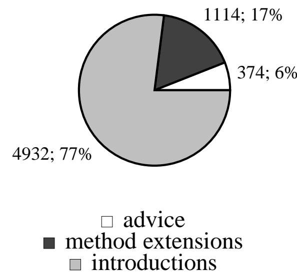

Figure 16. Static and dynamic crosscutting (LOC).

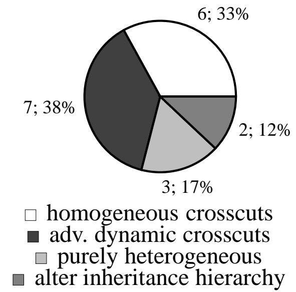

Figure 17. Applications of aspects (number).

refinements implemented as mixins, and 14 aspects were used to modularize crosscuts. The main point is that we primarily used classes and mixins for implementing features rather than aspects, which were used only to a minor degree – about 5% of the overall number of mechanisms for constructing features (Fig. 14).

LOC associated with classes, mixins, and aspects. The overall code base of P2P-PL consists of 6426 LOC. Thereof, 3056 LOC are associated with classes, 2964 LOC with mixins, and 406 LOC with aspects and refinements to aspects. These statistics are in line with the above given numbers on the ratio of implementation mechanism usage. Aspect code sums up to only 6% and mixin code to 46% of the overall code base (Fig. 15).

LOC associated with refinements and introductions. 1488 LOC of all mixins and aspects target the refinement of existing methods (dynamic crosscuts). Thereof, 374 LOC are associated with AspectJ advice and 1114 with method refinements via mixins. The remaining 4938 LOC are associated with introductions of new functionality (static crosscuts). This suggests that the dominant activity of features is to introduce new structures in P2P-PL (77%), rather than refining existing methods (Fig. 16).

3.4.2 Statistics on AMLs with Aspects

Number and properties of aspects. Of the 14 aspects that were used, 6 modularized homogeneous crosscuts (that refined a set of targets coherently with a single piece of code), 7 aspects implemented advanced dynamic crosscuts (that access dynamic context information, e.g., *cflow*), 2 aspects altered inheritance relationships (that introduce interfaces), and 3 aspects implemented purely heterogeneous crosscuts (Fig. 17).[9] 11 of 14 aspects exploit the advanced capabilities of AOP (cf. Sec. 2). Using only mixins would result in redundant, scattered, and tangled workarounds, as explained before. Only 3 aspects implement collaborations that could also be implemented by a set of mixins. Section 4 explains why in these particular cases using aspects was appropriate.

Number of feature-related classes and mixins. To explore if aspects are used stand-alone or with other classes and mixins in concert, we observed that an AML with one aspect also has 1 to 2 additional classes and mixins – up to 6. This shows that our AMLs encapsulate collaborations of aspects, classes, and mixins.

3.4.3 Statistics on Aspect Refinement

As explained in Section 3.3, AR is useful for decomposing and refining aspects as derivatives. Table 2 summaries the considered aspects and the number of pieces (base + derivatives) into which they were decomposed. On average, there were 7 derivatives per base aspect and 1/2 of all aspects were candidates for decomposition via AR. While this increased the total number of AMLs considerably, it enabled us to synthesize the application-specific aspects needed for a particular P2P system.

decomposed aspect	number of derivatives
serialization	11
responding	4
toString	12
log/debug	13
pooling	3
dissemination	3
feedback	2

Table 2. The aspects that were decomposed by AR.

4. Lessons Learned: A Guideline for Programmers

4.1 Mixins and Aspects – When to Use What?

A central question for programmers is when to use mixins and when to use aspects? What we have learned from our case study is that a wide range of problems can be solved by using object-oriented mechanisms and mixins (FOP). Specifically, we used mixins for expressing and refining collaborations of classes. Collaborations typically are heterogeneous crosscuts with respect to a base program. Each added feature reflects a subset of the structure of the base program and adds new and refines existing structural elements. A significant body of prior work advocates this view [41, 33, 25, 9, 37, 38, 39, 12].

Using aspects standalone for implementing collaboration-based designs, as proposed in [36, 19], would not reflect the object-oriented structure of the program that the programmer had in mind during the design. For example, the peer abstraction of P2P-PL plays different roles in different collaborations, e.g., with the network driver and with the data management. Encapsulating these different roles and their collaborations in single aspects would hinder a programmer's ability to recognize and understand the inherent object-oriented structure and the meaning of these features. In particular, if a collaboration embraces many roles and they are merged into one (or more) standalone aspect(s), the resulting code would be hard to read and to understand.

Nevertheless, aspects are a useful modularization mechanism. In our study we learned that they help in those situations where traditional object-oriented techniques and mixins fail. We found that (1) aspects reduce replicated code when implementing homogeneous crosscuts, (2) they help to modularly express advanced dynamic crosscuts (e.g., *cflow*), and (3) they support the subsequent altering of inheritance relationships. Aspects perform better in these respects than traditional object-oriented approaches be-

[9] Note that some aspects were counted for more than one category, e.g., homogeneous *and* dynamic.

cause they provide sophisticated language-level constructs that capture the programmers intension more precisely and intuitively.

Our case study provides statistics on how often AOP and FOP mechanisms are used. AOP mechanisms were used in 12% of all end-user visible features, because they allowed us to avoid code replication, scattering, and tangling. However, aspects occupied only 6% of the code base. This is because standard object-oriented mechanisms are sufficient to implement most features (i.e., 94% of the P2P-PL code base).

4.2 Borderline Cases

While we understand the above considerations as guideline for programmers that helps in most situations to decide between aspects and mixins, we also discovered few situations where a decision is not obvious.

We realized that some homogeneous crosscuts alternatively could be modularized by introducing an abstract base class that encapsulates this common behavior. While this works, for example, for all messages or message handlers, it does not work for classes that are completely unrelated, as in the case of a logging feature. It is up to the programmer to decide if the target classes are syntactically and semantically close enough to be grouped via an abstract base class.

Although, our study has shown that a traditional collaboration-based design ala FOP works well for the most features, we found at least one heterogeneous feature where it is not clear if it would not be more intuitive to implement it via an aspect. This feature introduces *toString* methods to a set of classes (cf. Tab. 1). Naturally, each of these methods is differently implemented. Thus, the feature is a heterogeneous crosscut. However, in this particular case it seems more intuitive to group all *toString* methods in one aspect. We believe this is caused by the partly homogeneous nature of this crosscut, i.e., introducing a set of methods for the same purpose to different classes.

5. Related Work

We limit the discussion of related work to the evaluation, combination, and comparison of AOP and FOP. Work related to AMLs and AR is discussed elsewhere [5, 4, 3].

5.1 Evaluation of AOP

Recent studies have evaluated AOP by its application to real world software projects. We review a representative subset.

Colyer and Clement refactored an application server using aspects [16]. Specifically, they factored 3 homogeneous and 1 heterogeneous crosscuts. While the number of aspects is marginal, the size of the case study is impressively high (millions of LOC). Although they draw positive conclusions, they admit (but do not explore) a strong relationship to FOP. Our study has demonstrated the useful integration of both worlds.

Zhang and Jacobsen refactored several CORBA ORBs [43]. Using code metrics, they demonstrated that program complexity could be reduced. They proposed an incremental process of refactoring which they call *horizontal decomposition*. Liu et al. have pointed to the close relationship to FOP layering [26]. Our study has confirmed former arguments that for implementing features, as aspects are too small units of modularization [33, 25, 5].

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [15]. They factored 4 concerns and evolved them in three steps; inherent properties of concerns were not explained in detail. Our study has shown that AR can help to evolve aspects over several development steps.

Lohmann et al. examined the applicability of AOP to embedded infrastructure software [27]. They have shown that AOP mechanisms, carefully used, do not impose a significant overhead. For their study they factored 3 concerns of a commercial embedded operating system; 2 concerns were homogeneous and 1 heterogeneous. Furthermore, they showed that aspects are useful for encapsulating design decisions, which is also confirmed by our study.

5.2 Evaluation of FOP

A significant body of research supports the success of FOP to implement large-scale applications, e.g., for the domain of network software [8], databases [10, 24, 8], avionics [6], and command-and-control simulators [7], to mention a few. The AHEAD tool suite is the largest example with about 80-200 KLOC [9, 40]. However, none of these studies make quantitative statements about the properties of the implemented features, nor do they evaluate the used implementation mechanisms with respect to the structures of the concerns. The features they considered were traditional collaborations with heterogeneous crosscuts, which is in line with our findings in P2P-PL.

Lopez-Herrejon et al. explored the ability of AOP to implement product lines in a FOP and SWD fashion [28]. They demonstrate how collaborations are translated automatically to aspects. They do not address in what situations which implementation technique is most appropriate nor how the generated aspects affect program comprehensibility.

Xin et al. evaluated *Jiazzi* and AspectJ for feature-wise decomposition [42]. They reimplemented a AspectJ-based CORBA event service [21] by replacing aspects with *Jiazzi units*, which are a form of feature modules. They concluded that Jiazzi provides better support for structuring software and manipulating features, while AspectJ is more suitable for manipulating existing Java code in unanticipated ways. However, they do not examine the structure of the implemented features. Their success to implement all features of their case study using Jiazzi feature modules hints that most of them (if not all) come in form of object-oriented collaborations.

We are not aware of further published studies that take both, AOP and FOP, into account.

5.3 Combining AOP and FOP

Several studies suggest the synergistic potential of aspects, roles, and collaborations, e.g., *Caesar* [32, 33], *Aspectual Collaborations* [25], and *Object Teams* [20]. Since these approaches were highly influenced by one another, we compare our approach to their general concepts. We choose caesar as a representative because it is the most mature.

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 crosscuts.

All of the mentioned approaches abstract collaborations of classes explicitly at language level and enrich these abstractions by different AOP mechanisms. Since all principally support collaboration-based designs, they all are capable of implementing those features of P2P-PL that do not contain aspects (99 features that sum up to 94% of the code base). It is not clear how our structuring and refinement of aspects can be mapped to these approaches; this is a subject of future work.

5.4 Collaborations and Super-Imposition

Steimann argues that expressing collaborations using object-oriented techniques facilitates a better program understanding than using aspects [39]. He builds his arguments on a long line of work on object-oriented and conceptual modeling, as surveyed in [38].

However, he does not distinguish between homogeneous and heterogeneous crosscuts nor between static and dynamic crosscuts.

Bosh demonstrates how super-imposing collaborations outperforms other component integration techniques such as wrapping and aggregation [12]. Although he does not explicitly take AOP into account, he favors collaborations for implementing features.

Our study has shown that for most features in P2P-PL the arguments of Steimann and Bosh are valid. Nevertheless, in certain situations traditional object-oriented techniques fail and AOP mechanism perform better.

5.5 Roles and Aspects

Pulvermüller et al. propose to implement collaborations as single aspects that inject the participating roles into the base program by using introductions declarations and advice [36]. In our study we made the observation that explicitly representing collaborations by traditional object-oriented techniques and mixins facilitates program comprehensibility. Moreover, favoring their approach would lead at the end to a base program with empty classes that are extended by several aspects that inject structure and behavior. This would destroy the object-oriented structure of the program and would hinder the programmer to understand the structure and behavior of the overall program as well as its individual features.

Some authors suggest to use aspects for implementing individual roles [19, 22]. In our context this would mean to replace each mixin within a feature by one or more aspects. We and others [39, 33] argue that replacing traditional object-oriented techniques that suffice (e.g., inheritance) is questionable. Instead, we favor to use aspects only when traditional techniques fail.

6. Conclusion

AOP and FOP are complementary technologies: we and others before us have noticed that the weakness of one maps to the strength of the other. AML integrates both technologies. As a case study, we used AML to implement a non-trivial product line of overlay networks. Our study showed that combining AOP and FOP improved the modularity of features in our product line. That is, using only aspects or only mixins would not have achieved an elegant design or implementation; only their combination achieved these goals. We observed that the dominant activity of features is introductions – adding new classes and new members to existing classes. Refinement of existing methods involved a small fraction of feature activities in our case study. Further, while aspects were infrequently used, they enhanced the crosscut modularity of features and reduced redundant code. Although we cannot generalize the results of a single case study to future studies, we believe our work supports the hypothesis that object-oriented collaborations (expressed by classes and mixins) define the predominant way in which concerns (features) are implemented, where aspects are useful in expressing homogeneous and advanced dynamic crosscuts.

Acknowledgments

We thank Roberto Lopez-Herrejon, Peri Tarr, Sahil Thaker, Salvador Trujillo, and the anonymous reviewers for their helpful comments on earlier drafts of this paper. Sven Apel is sponsored in part by the German Research Foundation (DFG), project number SA 465/31-1, as well as by the German Academic Exchange Service (DAAD), PKZ D/05/44809. The presented study was conducted when Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. Don Batory's research is sponsored by NSF's Science of Design Project #CCF-0438786.

References

[1] S. Apel and K. Böhm. Self-Organization in Overlay Networks. In *Proceedings of CAISE Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA)*, 2005.

[2] S. Apel and E. Buchmann. Biology-Inspired Optimizations of Peer-to-Peer Overlay Networks. *Practice in Information Processing and Communications (Praxis der Informationsverarbeitung und Kommunikation)*, 28(4), 2005.

[3] S. Apel et al. Aspect Refinement. Technical Report 10, Department of Computer Science, University of Magdeburg, Germany, 2006.

[4] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounded Quantification in Incremental Designs. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, 2005.

[5] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2006.

[6] D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Proceedings of Symposium on Software Reusability (SSR)*, 1995.

[7] D. Batory et al. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), 2002.

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

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

[10] D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems (JIIS)*, 9(2), 1997.

[11] K. Böhm and E. Buchmann. Free-Riding-Aware Forwarding in Content-Addressable Networks. *VLDB Journal*, 2006.

[12] J. Bosch. Super-Imposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5), 1999.

[13] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, 1990.

[14] E. Buchmann, S. Apel, and G. Saake. Piggyback Meta-Data Propagation in Distributed Hash Tables. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST)*, 2005.

[15] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

[16] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[17] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004.

[18] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[19] S. Hanenberg and R. Unland. Roles and Aspects: Similarities, Differences, and Synergetic Potential. In *Proceedings of International Conference on Object-Oriented Information Systems (OOIS)*, 2002.

[20] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World (NetObjectDays)*, 2002.

[21] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*, 2002.

[22] E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[23] G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[24] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of East-European Conference on Advances in Databases and Information Systems (ADBIS)*, 2005.

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

[26] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2006.

[27] D. Lohmann et al. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of ACM SIGOPS EuroSys Conference*, 2006.

[28] R. Lopez-Herrejon and D. Batory. From Crosscutting Concerns to Product Lines: A Function Composition Approach. Technical Report TR-06-24, University of Texas at Austin, 2006.

[29] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proccedings of European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[30] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of International Symposium on Partial Evaluation and Program Manipulation (PEPM)*, 2006.

[31] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[32] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

[33] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[34] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2), 1979.

[35] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[36] E. Pulvermüller, A. Speck, and A. Rashid. Implementing Collaboration-Based Design Using Aspect-Oriented Programming. In *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-USA)*, 2000.

[37] 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 (TOSEM)*, 11(2), 2002.

[38] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modeling. *Data and Knowledge Engineering (DKE)*, 35(1), 2000.

[39] F. Steimann. Domain Models are Aspect Free. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML)*, 2005.

[40] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE)*, 2006.

[41] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1996.

[42] B. Xin et al. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, University of Utah, 2004.

[43] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.