Technical Report No. 06 of the Otto-von-Guericke Universitaet, Magdeburg, Germany, May 2006.

An Empirical Study on Features and Aspects

Sven Apel[1] and Don Batory[2]

[1] Department of Computer Science, University of Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de
[2] Department of Computer Sciences, University of Texas at Austin
batory@cs.utexas.edu

Abstract. Recent studies have suggested the techniques of aspect-oriented and feature-oriented programming be combined to overcome their individual shortcomings. While previous work mainly argues on the basis of conceptual considerations and micro examples, in this paper, we evaluate the key ideas quantitatively by means of a non-trivial case study, a product line for overlay networks. Specifically, we pick out *aspectual mixin layers* as a representative approach that unifies AOP and FOP and show how our results apply to other approaches that integrate aspects and features. Although we have many results to report, we reveal and discuss several issues that remain open. Furthermore, we present a set of guidelines to assist programmers in how and when to use aspect-oriented and feature-oriented techniques for implementing product lines in a stepwise and generative manner.

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)* [1] aims at large-scale compositional programming and feature modularity in product lines. *Aspect-oriented programming (AOP)* [2] focuses on crosscutting modularity in complex software.

In several studies it has turned out that both paradigms, despite their advantages, bear several shortcomings [3–6]. It has been recognized that the weakness of one maps roughly to the strength of the other. Hence, both paradigms are not competitive and can profit from each other [6]. Recent studies have suggested that both paradigms be combined to exploit their synergetic potential [3, 4, 7, 6].

However, prior work has not gone past conjectures. Although there are numerous plausible reasons on the benefits of combining AOP and FOP (e.g., improvement in modularity), there is no empirical evidence to support these conjectures. An obvious question is if their symbiosis contributes more than it impairs. That is, are there real world applications that demand the synergetic effects of merging FOP and AOP?

In this paper, we contribute a thorough, practical application, evaluation, and discussion of the key ideas by means of a non-trivial case study. To be able to make precise statements, we limit our considerations to the domain of

software product lines developed in a generative and stepwise manner [1, 8]. As a concrete application, we choose a product line for peer-to-peer overlay networks (*P2P-PL*). This product line was developed to experiment with novel overlay mechanisms and algorithms. These experiments demand for a high degree of customizability, reusability, and evolvability.

In order to compare our findings with prior work, our investigations draw on a conceptual evaluation framework that analyzes and compares AOP and FOP [6]. Furthermore, we pick out *aspectual mixin layers (AMLs)* as a concrete approach that realizes an architectural integration of AOP and FOP [6]. AMLs support collaboration-based design, mixin composition, aspect weaving, and refinement of aspects to decompose and structure software along its features. AMLs are not specific to a particular interpretation or implementation of FOP and AOP. After our analysis, we discuss how to generalize our results to alternative approaches.

Our study addresses the following issues: When and how does a programmer use and combine the provided mechanisms of AMLs? Do the individual implementation techniques used in AMLs collaborate well together? Was our usage of AMLs in our case study subjective? It boils down to the question when to use traditional object-oriented mechanisms (e.g., mixins) and when aspect-oriented techniques to implement features. To demonstrate that our decisions in P2P-PL were not driven by personal preferences, we collected several statistics about the application and properties of AMLs, and in doing so revealed some deeper and fundamental open issues. We contribute a discussion of these issues that may help to improve future symbiotic approaches that combine FOP and AOP.

Recapitulating the results of our case study, we extract a set of guidelines for programmers to assist in when and how to use aspect-oriented and feature-oriented mechanisms for stepwise and generative product line development. In this paper we make the following contributions:

- an evaluation of a non-trivial case study that yields empirical evidences for the successful integration of aspects and features.
- a set of guidelines of using AOP and FOP in product lines for improving feature modularity, and
- a discussion of open issues.

2 Background

This section reviews FOP and AOP as well as the key results of their conceptual evaluation. Furthermore, we describe aspectual mixin layers that integrate FOP and AOP techniques.

2.1 Feature-Oriented Programming

FOP studies the modularity of *features* in product lines, where a feature is an increment in program functionality [1]. *Feature modules* realize features at design and implementation levels. The idea of FOP is to synthesize software (individual programs) by composing feature modules. Typically, features modules refine the

content of other features modules in an incremental fashion. Hence, the term *refinement* refers to the set of changes a feature applies to others. *Stepwise refinement* leads to conceptually layered software designs. For simplicity, we use the terms feature and feature module synonymously.

Mixin layers are one concrete approach to implement features [9, 1]. The basic idea is that features are seldomly implemented by single classes (or aspects). Often, a whole set of *collaborating* classes defines a feature. Classes play different *roles* in different *collaborations* [10]. 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 [11]. A mixin layer is a static component encapsulating 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. The mixin layers crosscut multiple classes ($C_A - C_C$). White boxes represent mixins; gray boxes denote the enclosing feature modules; filled arrows refers to mixin-based inheritance for composing mixins [12].

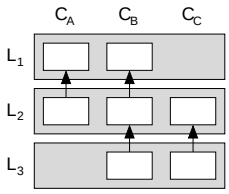

Fig. 1. Stack of three mixin layers.

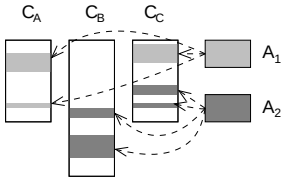

Fig. 2. Two aspects extend three classes.

2.2 Aspect-Oriented Programming

AOP aims at separating and modularizing crosscutting concerns [2]. Using object-oriented mechanisms the implementation of crosscutting concerns results in tangled and scattered code [2, 8]. The idea behind AOP is to implement crosscutting concerns as *aspects* whereas the core (non-crosscutting) features are implemented as components. Using *pointcuts* and *advice*, an aspect weaver glues aspects and components at predefined *join points* together. Pointcuts specify sets of join points of aspects and components, whereas advice defines which code is applied to (or executed at) these points. Typically, aspects introduce new members to classes and extend existing methods dynamically. By means of an aspect a programmer is able to refine a program coherently at multiple join points. Figure 2 shows two aspects (A_1, A_2) that extend three classes at multiple join points (dashed arrows denote aspect weaving).

2.3 Conceptual Evaluation Framework

We review a subset of our evaluation framework that was presented originally in [6].

Homogeneous vs. heterogeneous crosscuts. Homogeneous crosscuts refine multiple join points by adding one coherent piece of functionality. *Heterogeneous crosscuts* refine multiple join points with multiple pieces of functionality [13]. Implementing features spans several collaborating classes. Typically, a feature introduces a set of classes and methods that refines a complementary set of classes and methods in the base program. Hence, they are well qualified to implement heterogeneous crosscuts.

In contrast to features, aspects perform well in refining a set of parent entities using one coherent advice, thus, modularizing a homogeneous crosscut. By using aspects for homogeneous crosscuts, programmers avoid accidental code replication. Although, both approaches are able to implement the crosscuts of the other, they cannot do so elegantly [3, 6]. Consider a synchronization feature, which is a homogeneous crosscut. Using a mixin layer one refines a whole set of target methods with an appropriate set of refining methods that replicate the same synchronization code for all wrapped methods.

Conversely, an aspect may implement a collaboration of classes by applying a set of introductions. It has been argued that not expressing the collaboration explicitly decreases the program comprehensibility [14, 3, 15, 6]. This is because the programmer cannot recognize the original structure of the base program within subsequent refinement. A further argument is that aspects lack of scalability with respect to large-scale features: Suppose a collaboration consists of plenty of roles, e.g., a data management feature. Merging all participating roles (storage structures, file access, indexes, transaction management, etc.) in one or more aspects[3] would flatten the inherent object-oriented structure of the feature, obscure the intension of the programmer, and the resulting program would be hard to understand [14, 3].

Static vs. dynamic crosscutting. Features and aspects may extend the structure of a base program statically (*static crosscutting*), i.e. by injecting new members. Additionally, feature modules are able to encapsulate and introduce new classes. While aspects are not able to introduce independent classes, they provide the means to alter inheritance hierarchies, e.g., by introducing new interfaces to existing classes.

With dynamic crosscutting we refer to the ability of an implementation technique to apply a refinement dependently on the runtime control flow. 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.

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

2.4 Symbiosis of Aspects and Features

Comparing aspects and features, it turned out that in their current incarnation they are intended for solving problems at different levels of abstraction [3, 4, 6]. Whereas aspects in its current form act on the level of classes and objects (object-oriented architectures) in order to modularize crosscutting concerns, features act on a higher level of abstraction. A feature decomposes an object-oriented architecture to encapsulate those classes and their collaborations with other classes that contribute to its functionality. From an architectural point of view introducing aspects into an object-oriented architecture demands as next logical step for features that decompose the resulting *aspect-oriented architecture*. Figure 3 shows at left-hand side an aspect-oriented architecture and at the right-hand side features that decompose and structure this architecture. With this decomposition, a feature encapsulates fragments of classes *and* aspects that collaborate together to implement a feature. Note that the original aspect was split into two pieces. In Section 2.6, we address this issue in more depth.

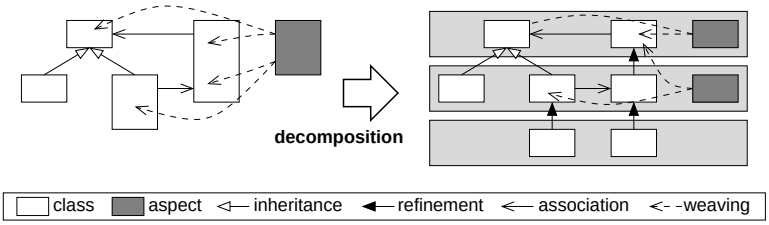

Fig. 3. Feature-driven decomposition of aspect-oriented architectures (features are depicted light-gray).

2.5 Aspectual Mixin Layers

Aspectual mixin layers (AMLs) is an approach to implement the architectural integration of AOP and FOP. AMLs extend the notion of mixin layers by encapsulating besides mixins also aspects (see Fig. 4). That is, an AML encapsulates those roles of 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 AMLs is that programmers may choose the appropriate technique – mixins or aspects – that fits a given problem best. Moreover, they can apply a collaboration of both and decide to what extent one technique is used.

2.6 Aspect Refinement

Aspect refinement (AR) is the incarnation of SWD in AOP [16]. Although the notion of AR does not depend on AMLs, it profits from the integration of aspects

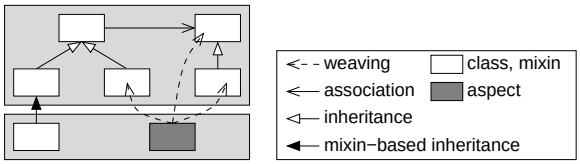

Fig. 4. Aspectual mixin layers.

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 aspects means adding new members and extending existing members of these aspects. To support AR at language level, the notion of *mixin-based aspect inheritance* has been proposed. It adopts ideas of mixins for composing aspects [16]. In order to uniformly refine all structural elements of aspects, the notions of *pointcut refinement*, *named advice*, and *advice refinement* have been introduced – all based on mixin capabilities. 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 to the aspect is part of an AML as well. AR allows every piece of an aspect to be reused, refined, and evolved [16, 6].

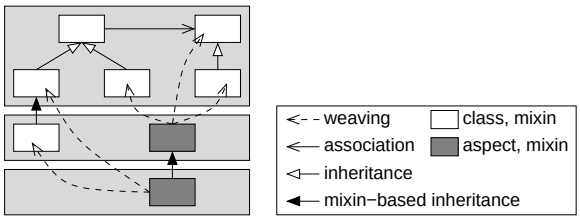

Fig. 5. Aspect refinement.

In context of AMLs, refinements to aspects are encapsulated by subsequently applied AMLs. In other words, decomposing an aspect into a base aspect and several refinements means decomposing the enclosing AML into several pieces that are themselves AMLs, and that encapsulate the corresponding refinements to the base aspect (see Fig. 6). Notably, feature decomposition does not result always in a set of fully-fledged features, but merely in modules that implement only a subset of a desired feature functionality [17, 18].

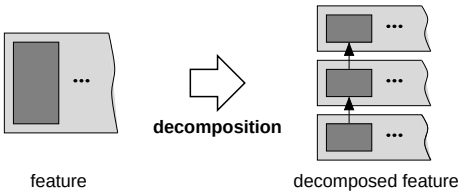

Fig. 6. Decomposing aspects by decomposing AMLs.

3 Case Study

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

As case study we use a product line for *peer-to-peer overlay networks (P2P-PL)* [19–21]. Besides the basic functionality as routing and data management, P2P-PL supports several advanced features, e.g. query evaluation optimization [20], swarm-like meta-data propagation [19], incentive mechanisms to counter free riding [22]. Numerous experiments concerning those features demanded for deriving plenty of different configurations to make statements about their specific effects, their variants, and combinations. The implementation of P2P-PL was almost complete when we began our evaluating study. At this point, we identified several design and implementation problems caused by code scattering and tangling.

P2P-PL has a very 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 [23]. In its current state, it consists of 113 features, categorized into several sub-domains, e.g. hashing, or overlay topology.

We implemented the features mainly using the *AHEAD tool suite (ATS)* [1]. The ATS supports FOP for Java. To integrate aspects into feature modules, we used several tools, first of all *ARJ*[4], an extended AspectJ compiler that supports AR [16].

We used AMLs and AR to improve the modularity of P2P-PL in order to avoid the mentioned problems of code scattering and tangling. Furthermore, we aimed at customizability and code reuse. This study gives insight into the practical applicability and the benefits of AMLs and AR applied to a real world scenario. In the following section, we describe when and how we used AMLs and AR to implement new and to refactor existing features. For sake of simplicity, we pick out three representative features. Afterwards, we summarize our overall experiences.

3.2 Aspectual Mixin Layers in P2P-PL

We implemented 14 of the 113 features of P2P-PL as AMLs (12%); the remaining 99 features were implemented as traditional mixin layers. These numbers are

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

not arbitrary but influenced by the features to implement (see Sec. 3.4). Table 1 summarizes information about the AMLs. Before we quantitatively examine the properties of the AMLs, we explain two concrete 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. A feedback generator is part of an incentive mechanism for penalizing free riders – peers that profit of the P2P network but do not contribute adequately [22]. The generator observes the message traffic to keep track which messages have been responded. Depending on if an answer came in time, it creates positive or negative feedback. That is stored and used by other mechanisms to judge about the cooperativeness of peers.

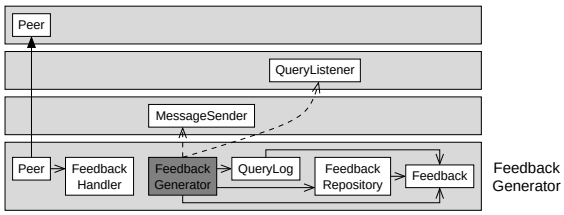

Fig. 7. Feedback generator AML.

The implementation of the generator crosscuts the message sending and receiving features. Although it is heterogeneous, it relies on dynamic context information (e.g., it uses *cflow*). As Figure 7 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 objects.

Figure 8 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 2-6). In order to not affect multiple *send* methods, it uses pointcuts that match the dynamic control flow (Lines 3-6). The second advice intercepts the execution of a query listener task for creating feedback (Lines 7-8).

```
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   after(QueryListener listener) : target(listener) &&
8     execution(void QueryListener.run()) { ... }
9 }
```

Fig. 8. Feedback generator aspect (excerpt).

Figure 9 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 and for other message types.

```
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 }
```

Fig. 9. Feedback management refinement of the peer class.

In summary, within the feedback generator, AML four classes implement the basic feedback management; an aspect intercepts the message transfer; and a mixin refines the peer abstraction by capabilities for feedback management. On one hand, 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,

we found that implementing this feature as one standalone aspect would not reflect the structure of the P2P framework including the feedback management. All would be merged within one aspect and not explicitly represented for program comprehension and for subsequent refinement.

Instead, our AML encapsulates all contributing elements coherently as a collaboration that reflects the intuitive structure of the P2P framework we had in mind during its design.

Connection pooling. Connection pooling is a mechanism for reusing open connections 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 10 shows that the AML consists of an aspect and a pool class. The aspect intercepts all calls that create and close connections.[5] The pool stores open connections.

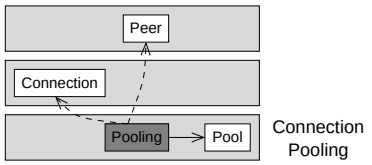

Fig. 10. Connection pooling AML.

Figure 11 lists the pooling aspect; it owns 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[6] *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 each of both advice advise a whole set of join points that are related, e.g., all client-side calls to *close* (Lines 7-9). We extend this set in the next paragraph.

Implementing this feature only by using 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.

[5] Note that this is not ideally visualized because the calls are intercepted at client / caller side.

[6] Named advice assigns a name to advice for enabling subsequent refinement [16].

```
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 }
```

Fig. 11. Connection pooling aspect (excerpt).

3.3 Aspect Refinement in P2P-PL

In summary, we applied the notion of AR to 8 of our 14 AMLs. That is, we decomposed each of the 8 aspects into several pieces (each aspect into a base aspect and several refinements). We encapsulated each refinement to an aspect in a separate AML. The resulting AMLs were not counted to the overall number of AMLs (14) because we did not consider them as fully-fledged features but as subsets that contribute to a larger feature (cf. Sec. 2.6). Again, we explain two concrete examples in detail; later on we summarize our quantitative results.

Serialization. The serialization feature is very simple. It consists only of one aspect. We chose this example because it is an homogeneous crosscut and it illustrates the benefits of AR. The aspect introduces a new interface to a set of classes that objects are interchanged via stream and network connections. These classes, e.g. data items, keys, contacts, etc., do not declare this interface because they are supposed to be reusable in other contexts, which do not rely on serialization.[7]

Figure 12 depicts the serialization aspect of an arbitrary configuration. It simply enumerates a list of *declare parent* statements for introducing the interface *Serializable* to a set of target classes.[8]

The list of declared parents depends on the current configuration of P2P-PL. Hence, the serialization aspect depends highly on the feature selection. For example, if we decide to remove the feedback generator, we would get an error because the class *Feedback* would not be part of P2P-PL, but the serialization aspect refers to it.

Such *multiple feature dependencies* are an appropriate use case for AR; we apply AR in order to break open the aspect into smaller pieces – refinements –

[7] Some derived overlay networks do not operate on top of a physical networks, but virtually inside a computer.

[8] AspectJ is not able to encapsulate this homogeneous crosscut completely because the code piece "*implements Serializable*" is redundant. Other AOP languages such as *AspectC++* [24] and *LogicAJ* [25] provide more sophisticated means.

```
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 }
```

Fig. 12. Serialization aspect (excerpt).

to resolve interactions between the serialization feature and others. This allows us to select only those pieces that refine selected features.

Figure 13 lists the refactored serialization aspect and its factored out refinements (merged in one listing). How fine-grained such refactoring has to be depends on the desired flexibility for composing different variants. In P2P-PL, we split the compound serialization aspect in 16 pieces.

```
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 } ...
```

Fig. 13. Refactored serialization aspect (excerpt).

Connection pooling. Based on our experiences with overlay networks, we recognized several useful and genuine refinements to the connection pooling aspect. For the sake of simplicity, we limit our focus to the essential parts of three refinements and we abstract over implementation details.

Figure 14 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 [16]).

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

```
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 }
```

Fig. 14. Encapsulating design decisions using AR.

can simply extend them (via advice refinement) with synchronization code. As one can see, when refining named advice they are treated similarly to conventional methods (see [16]).

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 example refinements show how the effect of different design decisions can be encapsulated in order to configure different pooling variants, depending on the application context. These examples show the usefulness of our proposed mechanisms, pointcut refinement, named advice, and advice refinement. Note that the connection pooling aspect and its refinements differ from the serialization aspect. While here we used AR for encapsulating different design decisions, with the serialization aspect, we used AR for resolving feature dependencies.

3.4 Quantitative Analysis

The examples have shown that we found several situations where AMLs and AR have proved useful (12% of features were implemented by AMLs). However, this does not prove that we applied AMLs and AR in an appropriate way. Therefore, we analyze when and how we applied them. It is interesting to know to what extent we employed AOP and FOP. Since the conceptual evaluation framework suggests for what kind of feature what mechanism is most beneficial (Sec. 2.3), we are able to compare our experimental results with these suggestions. In the Sections 3.4-3.4, we present the facts extracted from our analysis and in Section 3.5, we interpret and discuss them in more depth.

Statistics on Used AOP and FOP Mechanisms. We collected the following statistics: (1) number of used implementation mechanisms, (2) LOC associated

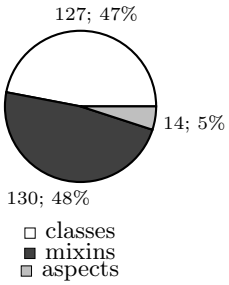

Fig. 15. Classes, mixins, and aspects (number).

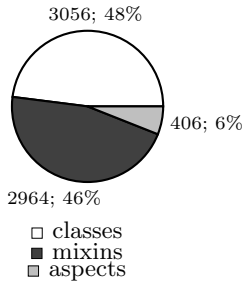

Fig. 16. Classes, mixins, and aspects (LOC).

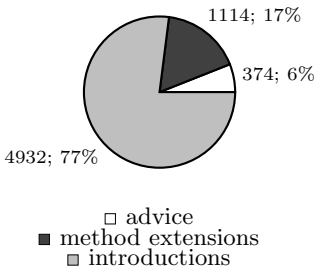

Fig. 17. Static and dynamic crosscutting (LOC).

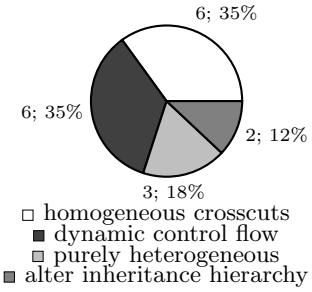

Fig. 18. Applications of aspects (number).

with these mechanisms, and (3) LOC associated with introductions (static crosscutting) and extending and advising methods (dynamic crosscutting).

Number of classes, mixins, and aspects. P2P-PL consists of 127 classes. These classes were introduced incrementally in several development steps. Furthermore, we implemented 130 subsequent class refinements as mixins, and we used 14 aspects for modularizing crosscutting features, as we will explain later. The main point is that we used mainly 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. 15).

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. 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. 16).

LOC associated with extensions and introductions. 1488 LOC of all implemented mixins and aspects are associated to extending existing methods (dynamic crosscutting). Thereof, 374 LOC are associated with AspectJ advice and 1114 with method extensions via mixins. The remaining 4938 LOC are associated with introductions of new functionality (static crosscutting). These statistics demonstrate that introducing new structures in P2P-PL was the dominant activity (77%), rather than extending and advising existing methods (Fig. 17).

Statistics on Aspectual Mixin Layers. As Table 1 shows, we used 14 AMLs within P2P-PL. A question that arises is, was the use of AMLs subjective or even arbitrary? Did we employ AMLs as opposed to traditional mixin layers in the right situations? In order to explain that our implementation decisions were not driven by personal preferences, we collected statistics about the properties and application of AMLs.

Number and properties of aspects. To get deeper insight in why and for what we used aspects within AMLs, we analyzed their structure and purpose: we implemented 6 aspects that modularize homogeneous crosscuts (that refine a set of targets coherently with the same code piece), 6 aspects that employ dynamic crosscutting (that access dynamic context information, e.g., *cflow*), 2 aspects that alter inheritance relationships (that introduce interfaces), and 3 aspects that implement purely heterogeneous crosscuts (Fig. 18).[9] As explained in Section 2, most of these uses (82%) exploit the advanced capabilities of aspects. Simply applying mixins would result in redundant, scattered, and tangled workarounds, as explained before. Only three aspects implement collaborations that could also be implemented by a set of mixins. Section 4 explains why in these particular cases aspects were appropriate anyways.

Number of feature-related classes. To explore if aspects are used stand-alone or with other classes and mixins in concert, we determined the number of classes and aspects per AML. On average, an AML introduces one aspect and 2 to 3 additional classes – up to 11 new classes per AML. This shows that our AMLs encapsulate collaborations of aspects, classes, and mixins.

Statistics on Aspect Refinement. As we have demonstrated, AR is useful for decomposing and refining aspects. Although AR is not limited to AMLs, the layer structure imposed by AMLs facilitates handling aspects and their refinements. We have applied the notion of AR to 8 of 14 AMLs within P2P-PL. Result for each decomposed AML is a set of aspect refinements encapsulated in several separate AMLs. Table 2 gives information about the decomposed AMLs and explains briefly for what reasons we used AR.

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

aspect	# pieces	description
serialization	16	serialization support for several classes
responding	5	introduces a separate refinement for each type of message
toString	14	introduces *toString* methods to several classes
log/debug	18	logging and debugging depending on other features
pooling	4	connection pooling, e.g. synchronization, network parameters
dissemination	12	meta-data propagation, e.g. diss. strategy, time stamps
feedback	6	generates feedback, e.g. feedback types, storing
caching	7	caches contact data, e.g. caching strategy,

Table 2. Overall use of aspect refinement.

Feature dependencies. Within P2P-PL, we used AR for resolving feature dependencies (cf. Sec. 3.3). Besides the mentioned serialization aspect, we identified 3 further aspects that have only one single purpose but affect plenty of other features (rows 3-5 in Table 2). To decouple them from a fixed set of base features, we decomposed them into pieces, encapsulated in distinct AMLs.

Multiple design decisions. Another use case of AR is to encapsulate the effects of several design decisions that are associated with an aspect (cf. Sec. 3.3). Thereby, we break off the otherwise hard-wired functionality to be able to trace design decisions in code and to improve configurability. Within P2P-PL, we decomposed 4 of such aspects (last four aspects in Table 2).

Average decomposition degree. On average, we decomposed the considered aspects in a base aspect and 9 refinements. Table 2 shows in how many pieces the individual aspects were decomposed. Surely, we applied the notion of AR only to those aspects that seemed promising. Nevertheless, over 1/2 of all aspects shaped up as good candidates for decomposition via AR.

3.5 Summary

Our case study shows that AMLs and AR are applicable to a real software project. Aspects integrated into traditional features helped to modularize crosscutting features. Moreover, it demonstrates that aspects and mixins work together in concert.

Specifically, we could improve the feature modularity in 12% of all features by using AOP mechanisms. In this way, we avoided code scattering that would otherwise affect lots of other feature modules. However, our study showed that aspects were associated only to 6% of the code base. This is simply because features whose implementation demands for AOP mechanisms occur not as frequently as features that come in form of collaborations that are super-imposed.

Additionally, we refactored 8 AMLs using AR. On average, each aspect within the considered AMLs was decomposed into 10 pieces encapsulated by 10 separate

AMLs. While this increased the number of AML considerably, it allowed us (1) to resolve dependencies of 4 AMLs to other features, and (2) to encapsulate and separate several design decisions that were otherwise hard-wired, in case of 4 aspects.

Reviewing these results, we perceive traditional collaborations as skeletons of product lines. This does not imply a specific implementation mechanism, but we were able to implement 94% of the P2P-PL code base using collaborations of classes and mixins, which we consider standard object-oriented techniques. Aspects are not as frequently used as collaborations. This observation is in line with the original purpose of aspects to implement specific crosscutting concerns. Furthermore, the study demontrated that aspects and mixins collaborate well together in P2P-PL.

AR is a logical next step when integrating aspects into layered designs as in SWD. The study demonstrated the usefulness of AR and revealed guidelines to discover, design, and refine aspects in a stepwise manner.

4 Perspective

In this section we put the results of our case study into perspective.

4.1 Lessons Learned: A Guideline for Programmers

Mixins and aspects – when to use what? One central question for programmers is when to use mixins and when to use aspects? What we have learned of 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 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 [10, 3, 4, 1, 9, 11, 14, 15].

Using aspects standalone for implementing collaboration-based designs, as proposed in [26, 27], would not reflect the natural structure of the program (that the programmer had in mind during the design) within subsequent refinements (implemented by single aspects). 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 the programmer to recognize and understand the inherent object-oriented structure and the meaning of these features. Especially, if a collaboration embraces plenty of roles and we merge them all into one (or more) standalone aspect(s), the resulting code would be hard to read and to understand. The structure of P2P-PL would be hidden and intermixed in the flattened aspect code. This information loss would reduce program comprehensibility and maintainability.

Nevertheless, aspects are a very useful modularization mechanism. In our study we have 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 implement otherwise inelegant workaround for expressing advanced dynamic crosscutting, and (3) they support the subsequent altering of inheritance relationships. Aspects perform better in these respects than traditional object-oriented approaches because they provide several advanced language-level constructs that capture the programmers intension more precisely and intuitively.

Using mixins standalone for implementing homogeneous crosscuts, would result in a lot of replicated code since for each target point a distinct refining method has to be introduced, each with replicated code. Moreover, mixins do not support advanced mechanisms for dynamic crosscutting, such as *cflow*.

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 this 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 super-class or an interface.

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.

4.2 Open Issues

Granularity and scalability. On average, in P2P-PL each feature is implemented by 56 LOC. Thus, our features are very fine-grained. Although, we are not aware of principle metrics that tell programmers what feature granularity is appropriate, this fine-grained approach might not scale to larger software projects. One way to address this issue would be to implement coarse-grained features. While this overcomes the problem of limited scalability, it decreases the potential scenarios a feature can be reused with [28]. Remarkably, not aware of this fact when implementing P2P-PL, we chose intuitively an approach in between. We organized the set of 113 feature into a tree structure of subsystems. A top level

we related each feature to one of 4 subsystems that themselves have 12 subsystems: *P2P networks*, *data storage*, *distributed hash tables*, *content addressable networks*. All these subsystems have counterparts in the domain model of P2P systems. Those subsystems can be understood as large-scale compound features. Such hierarchical approach might be a trade-off between fine-grained customizability and scalability.

Code tangling. Our feedback generator used several times the message subsystem for accessing information about incoming and outgoing messages. We implemented this collaboration via direct method calls from the feedback generator to the message subsystem. Moreover, the feedback generator uses a logging subsystem to log its current state. This could also be implemented via method calls. Interesting is that most programmers would probably agree that collaborating with the message subsystem is not undesirable code tangling, but invoking a log instance is considered as code tangling. However, in this particular case it might be easy to decide but in other situations it might be unclear. So what is the general rule for considering a uses-relationship as tangling or as meaningful collaboration? Although we do not have a satisfactory answer, we refer to the *law of demeter of concerns (LoDC)* [29]. Informally, it says that a concern should only know about concerns that contribute to its functionality. Mapped to our problem it is evident that the message subsystem is necessary for the implementation of the feedback generator, whereas the logging feature does not contribute anything. In other words, programmers may use the LoDC for deciding when to use aspects and when collaborations of mixins and classes.

4.3 Generalization to Other Approaches

Our study used AMLs to implement features that demand special crosscutting functionality. Now we want to illustrate if and how our experiences can be applied to related approaches that combine AOP and FOP. Prominent representatives are *caesar* [30, 3], *aspectual collaborations* [4], and *object teams* [7].

All of the considered approaches abstract collaborations of classes explicitly at language level and enrich these abstractions by different AOP mechanisms like AMLs. 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). What differs from AMLs is their usage of AOP mechanisms. While AMLs integrate aspects into collaborations of software artifacts, others treat collaborations as aspect instances themselves. For example, caesar's aspect components encapsulate sets of nested classes (roles) and pointcuts and advice.

Historically, all of the considered approaches focus on on-demand remodularization and a posteriori integration of structurally differing components. While object teams and aspectual collaborations do not explicitly support pointcuts and advice for supporting homogeneous crosscuts, caesar provides a rich set, similar to AspectJ. Thus, caesar is also capable of implementing the 14 AMLs of P2P-PL analogously to our study.

However, none of the considered approaches support AR, but there is no reason why this notion could not be integrated. In summary, we believe that this study tells us more about AOP and FOP in general than about specific implementation approaches.

5 Related Work

We limit the discussion of related work to the evaluation and combination of AOP and FOP. Work related to AMLs and AR is discussed elsewhere [6, 16].

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

Colyer and Clement refactored an application server using aspects. Specifically, they factored 3 homogeneous and 1 heterogeneous crosscutting concerns. 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 [31]. Using code metrics, they demonstrated that program complexity could be reduced. They propose an incremental process of refactoring which they call *horizontal decomposition*. Liu et al. have pointed to the close relationship to FOP layering [17]. Our study has confirmed former arguments that for implementing features, aspects are too small units of modularization for implementing a broad variety of features [3, 4, 6].

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [32]. 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 [33]. 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 have shown that aspects are useful for encapsulating design decisions, which is also confirmed by our study.

Evaluation of FOP. A significant body of research supports the success of FOP to implement large-scale applications, e.g. for the domain of databases [34–36], network software [36], avionics [37], and command-and-control simulators [38], to mention a few. The AHEAD tool suite is the largest example with about 80-200 KLOC [1]. However, none of these studies make quantitative statements about the properties of the implemented features, neither they evaluate the used implementation mechanisms with respect to the structures of the concerns. We

have the impression the features they consider were mainly traditional collaborations that were heterogeneous crosscuts, which is in line with our findings in P2P-PL.

Lopez-Herrejon et al. explore the ability of AOP to implement product lines in a FOP and SWD fashion [39]. 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.

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

Combining AOP and FOP. Several studies suggest to exploit the synergetic potential of aspects, roles, and collaborations, e.g. *caesar* [30, 3], *adaptive plug-and-play components* [40], *pluggable composite adapters* [41], *aspectual collaborations* [4], and *object teams* [7]. 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 unifies the most essential ideas and it has grown to the most matured approach.

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

With the mentioned approaches it is not possible to refine embedded pointcuts and advice. They do not uniformly support SWD at language level. Furthermore, their collaborations (aspect components) are first-class and their composition is done within source code. There is no separation of the source code artifacts and their association to development steps. Our experience with P2P-PL was that even this separation facilitates the composition, reuse, and customization.

Collaborations and super-imposition. Steimann argues that expressing collaborations using object-oriented techniques facilitates a better program understanding than using aspects [14]. He builds his arguments on a long line of work on object-oriented and conceptual modeling [11]. However, he does not distinguish between homogeneous and heterogeneous crosscuts nor between static and dynamic crosscutting.

Bosh demonstrates how super-imposing collaborations outperforms other component integration techniques such as wrapping and aggregation [15]. 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.

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 and advice [26]. 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 [27, 42]. In our context this would mean to replace each mixin within a feature by one or more aspects. We and others [14, 3] 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

Recent studies have analyzed the strengths and shortcomings of AOP and FOP. In order to combine their advantages several studies proposed to integration of AOP and FOP concepts in different ways. But the arguments put forward by these studies are based mainly on conceptual considerations and simple examples.

In this paper we applied these ideas to a case study and evaluated the key concepts in a quantitative manner. For our study, we picked out one concrete approach for the symbiotic use of AOP and FOP and applied it to a real world application, a product line for overlay networks. Our study showed that combining AOP and FOP improved the modularity, reusability, and customizability of P2P-PL. Moreover, it turned out that although aspects were not the dominant modularization mechanism (6%), they enhanced the crosscutting modularity of feature modules and reduced thereby redundant code. Our study supports the hypothesis that collaborations of classes and mixins form the skeleton of a product line and aspects are used for certain crosscutting features. In this line, the study demonstrated the consensus of conceptual arguments and empirical findings. Furthermore, we pointed to several open issues whose clarification may sensiblize and improve further work. Finally, we gave a set of guidelines that assist the programmer in chosing and a using the right implementation mechanisms for the right problems.

Acknowledgments

We thank Sahil Thaker, Salvador Trujillo, and Roberto Lopez-Herrejon for useful comments and fruitful discussions on earlier drafts of this paper. This research is sponsored in parts 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.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering (TSE) **30**(6) (2004)
2. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP). (1997)
3. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). (2004)
4. Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal **46**(5) (2003)
5. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A Disciplined Approach to Aspect Composition. In: Proceedings of International Symposium on Partial Evaluation and Program Manipulation (PEPM). (2006)
6. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: Proceedings of International Conference on Software Engineering (ICSE). (2006)
7. Herrmann, S.: 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)
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
9. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(2) (2002)
10. VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1996)
11. Steimann, F.: On the Representation of Roles in Object-Oriented and Conceptual Modeling. Data and Knowledge Engineering **35**(1) (2000)
12. Bracha, G., Cook, W.: 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)
13. Colyer, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University (2004)
14. Steimann, F.: Domain Models are Aspect Free. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML). (2005)
15. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology **41**(5) (1999)
16. Apel, S., Leich, T., Saake, G.: Mixin-Based Aspect Inheritance. Technical Report 10, Department of Computer Science, University of Magdeburg, Germany (2005)
17. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proceedings of International Conference on Software Engineering (ICSE). (2006)
18. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proceedings of European Conference on Object-Oriented Programming. (1997)

19. Buchmann, E., Apel, S., Saake, G.: Piggyback Meta-Data Propagation in Distributed Hash Tables. In: Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST). (2005)
20. Apel, S., Buchmann, E.: Biology-Inspired Optimizations of Peer-to-Peer Overlay Networks. Practice in Information Processing and Communications (Praxis der Informationsverarbeitung und Kommunikation) **28**(4) (2005)
21. Apel, S., Böhm, K.: Self-Organization in Overlay Networks. In: Proceedings of CAISE Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA). (2005)
22. Böhm, K., Buchmann, E.: Free-Riding-Aware Forwarding in Content-Addressable Networks. VLDB Journal (2006)
23. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering (TSE) **SE-5**(2) (1979)
24. Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: An AOP Extension for C++. Software Developer's Journal (2005)
25. Kniesel, G., Rho, T.: A Definition, Overview and Taxonomy of Generic Aspect Languages. L'Objet (Special issue on Aspect-oriented Software Development) **11**(3) (2006)
26. Pulvermüller, E., Speck, A., Rashid, A.: Implementing Collaboration-Based Design Using Aspect-Oriented Programming. In: Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-USA). (2000)
27. Hanenberg, S., Unland, R.: Roles and Aspects: Similarities, Differences, and Synergetic Potential. In: Proceedings of International Conference on Object-Oriented Information Systems (OOIS). (2002)
28. Biggerstaff, T.: A Perspective of Generative Reuse. Annals of Software Engineering **5** (1998)
29. Lieberherr, K.: Controlling the Complexity of Software Designs. In: Proceedings of International Conference on Software Engineering (ICSE). (2004)
30. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: Proceedings of International Conference on Aspect-Oriented Software Development (AOSD). (2003)
31. Zhang, C., Jacobsen, H.A.: Resolving Feature Convolution in Middleware Systems. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2004)
32. Coady, Y., Kiczales, G.: 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)
33. Lohmann, D., et al.: A Quantitative Analysis of Aspects in the OS Kernel. In: Proceedings of ACM SIGOPS EuroSys Conference. (2006)
34. Batory, D., Thomas, J.: P2: A Lightweight DBMS Generator. Journal of Intelligent Information Systems (JIIS) **9**(2) (1997)
35. Leich, T., Apel, S., Saake, G.: 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)
36. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology (TOSEM) **1**(4) (1992)
37. Batory, D., et al.: Creating Reference Architectures: An Example from Avionics. In: Proceedings of Symposium on Software Reusability (SSR). (1995)

38. Batory, D., 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)
39. Lopez-Herrejon, R., Batory, D.: From Crosscutting Concerns to Product Lines: A Function Composition Approach. Technical Report TR-06-24, University of Texas at Austin (2006)
40. Mezini, M., Lieberherr, K.: Adaptive Plug-and-Play Components for Evolutionary Software Development. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1998)
41. Mezini, M., Seiter, L., Lieberherr, K.: Component Integration with Pluggable Composite Adapters. Software Architectures and Component Technology: The State of the Art in Research and Practice (2000)
42. Kendall, E.A.: Role Model Designs and Implementations with Aspect-Oriented Programming. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (1999)