On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?

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

Marko Rosenmüller
Department of Computer Science
University of Magdeburg, Germany
rosenmueller@iti.cs.uni-magdeburg.de

Abstract

While it is well known that crosscutting concerns occur in many software projects, little is known about the inherent properties of these concerns nor how *aspects* (should) deal with them. We present a framework for classifying the structural properties of crosscutting concerns into (1) those that benefit from AOP and (2) those that should be implemented by OOP mechanisms. Further, we propose a set of code metrics to perform this classification. Applying them to a case study is a first to step toward revealing the current practice of AOP.

1. Introduction

While many studies have examined the capabilities of *aspect-oriented programming (AOP)* to improve the modularity, customization, and evolution of software [8, 9, 13, 14, 21, 38], little is known on *how* AOP has been used. We are interested in knowing which language mechanisms are used in current aspect-oriented programs, to what extent, and for what kinds of problems. Knowing this helps (1) build AOP tools that reflect the programmer's needs; (2) provide programming guidelines for exploiting AOP mechanisms better, i.e., what kind of crosscutting concern is implemented best using which programming mechanism; and (3) discover misuse of AOP mechanisms, which may lead to significant problems and penalties [11, 13, 14, 16, 19, 24, 28].

To address these issues we propose a framework for classifying crosscutting concerns (a.k.a. *crosscuts*). Our framework enables us to assign individual crosscuts to two distinct categories: (1) crosscuts that really demand AOP mechanisms and (2) crosscuts that can be implemented appropriately using well-known OOP mechanisms. This distinction follows a long line of prior work on *collaboration-based designs* [31,32,35], *feature-oriented programming* [4], and *design patterns* [12]. All of them advocate object-oriented mechanisms for a certain class of design and implementation problems, so called *collaborations*, which fall into one category.

We propose four metrics to analyze aspect-oriented programs to make the above distinctions, i.e., do the aspects of a program implement crosscutting concerns that really demand AOP language mechanisms? We are building a tool that will collect data from a representative spectrum of software projects that employ AOP. We discuss the data for one *AspectJ*[1] project exemplarily.

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

2. Crosscut Classification Framework

2.1 Homogeneous and Heterogeneous Crosscuts

A *homogeneous crosscut* extends a program at multiple join points by adding one *extension*, which is a coherent piece of code [10]. For example, an advice may advise a whole set of method executions or an inter-type declaration may introduce a field to a set of target classes (left column of Table 1).

A *heterogeneous crosscut* extends multiple join points by adding multiple extensions, where each individual extension is implemented by a distinct piece of code that affects exactly one join point [10]. For example, an aspect might bundle a set of advice that extends a set of methods, whereby each advice extends exactly one method; or an aspect bundles a set of inter-type declarations – each intended for a distinct class (right column of Table 1).

2.2 Static and Dynamic Crosscuts

A *static crosscut* extends the structure of a program statically [29], i.e., it adds new classes and interfaces as well as injects new fields, methods, interfaces, and super-classes etc.[2] Inter-type declarations are examples of static crosscuts (first row of Table 1).

A *dynamic crosscut* affects the runtime control flow of a program [29]. The semantics of a dynamic crosscut can be understood and defined in terms of an event-based model [36]: it runs additional code when predefined events occur during program execution. Such events are also called *dynamic join points* [27, 36]. A piece of advice implements a dynamic crosscut (second row of Table 1).

Basic and advanced dynamic crosscuts. Dynamic crosscuts are especially interesting when they exceed the level of known events such as method calls or executions. Work on AOP suggests that expressing a program extension in terms of sophisticated events increases the abstraction level and captures the programmer's intension more directly. There are proposals for new language constructs for defining and catching new kinds of events during the program execution [26, 30]. In order to distinguish these new kinds of events and the novel language mechanisms that support them from known events in OOP, we distinguish between *basic dynamic crosscuts* and *advanced dynamic crosscuts*, which are defined as follows:

1. A basic dynamic crosscut addresses only events that are related to method calls and executions; advanced dynamic crosscuts address all other events, e.g., throwing an exception or assigning a value to a field.
2. Basic dynamic crosscuts affect a program control flow unconditionally; advanced dynamic crosscuts may specify a condition

[2] Some AOP languages do not support adding classes by aspects, e.g., AspectJ. While it is correct that one can just add another class to an environment, this is at the tool level, and is not at a model level [23].

<table>
<tr><th></th><th>homogeneous</th><th>heterogeneous</th></tr>
<tr><td>static</td><td><code>declare parents : (Line || Point) implements Shape</code></td><td><code>void Point.setX(int x) { /* ... */ }</code></td></tr>
<tr><td>basic dynamic</td><td><code>before() : execution(* set*(..)) { /* ... */ }</code></td><td><code>before() : execution(void Point.setX(int)) { /* ... */ }</code></td></tr>
<tr><td>advanced dynamic</td><td><code>before() : execution(* set*(..)) && !cflow(execution(* rotate(..))) { /* ... */ }</code></td><td><code>before() : execution(void Point.setX(int)) && !cflow(execution(void Line.rotate(double))) { /* ... */ }</code></td></tr>
</table>

Table 1. A classification framework for crosscutting concerns (AspectJ examples).

that is evaluated at runtime, e.g., a method execution is only advised if it occurs in the control flow of another method execution.

3. Basic dynamic crosscuts address events known from OOP; advanced dynamic crosscuts can specify composite events that trigger the execution of an extension, e.g., *trace matches* are executed when events fire in a specific pattern thereby involving the history of computation [1].

With AOP, an advanced dynamic crosscut is implemented by an *advanced advice* and a basic dynamic crosscut by a *basic advice*. The distinction between basic and advanced advice is useful to identify which pieces of advice make use of advanced AOP mechanisms and which pieces of advice mimic well-known OOP method extensions.

3. Two Categories of Crosscutting Concerns

We argue it is crucial to decide which crosscutting concerns should be implemented as aspects, and how, and which using traditional object-oriented techniques. For that purpose, we divide the space of possible crosscuts that is defined by our classification framework into two categories, (1) those crosscuts that abstract collaborations and (2) those that address the dynamic program semantics and/or that are homogeneous. The two categories map roughly to the two programming paradigms, OOP and AOP.

3.1 Collaborations

A *collaboration* of classes is a set of classes that communicate with one another to implement a semantically coherent piece of functionality. Classes of a program play different *roles* in different *collaborations* [35]. A set of collaborating classes being added to a program can be understood as a *feature* of that program [4]. That is, a collaboration extends a program by adding new classes and by applying new roles to existing classes, whereby each role is implemented as a refinement (e.g., using *virtual classes* [25] or *mixins* [6]). From that perspective, a role adds new elements to a class and extends existing elements, such as methods.

Figure 1 depicts a collaboration-based design of a graph implementation, where the classes *Graph*, *Node*, and *Edge* collaborate together.[3] The feature *WeightedGraph* adds the class *Weight* and extends the classes *Graph* and *Edge* simultaneously. For example, the class *Edge* plays two roles, one in the *BasicGraph* and one in the *WeightedGraph*.

A significant body of work has observed that collaborations of classes are predominantly of a heterogeneous structure [4, 5, 20, 29, 32–35]. That is, the roles and classes added to a programs differ in their functionality, as in our graph example. Hence, a collaboration is a heterogeneous crosscut and a heterogeneous crosscut can be understood as collaboration applied to a program. Therefore, it is straightforward to employ from techniques for encapsulating

[3] The diagram follows the UML notation with some extensions: white boxes represent classes *or* roles; gray boxes denote collaborations; filled arrows mean refinement, i.e., to apply a role to a class.

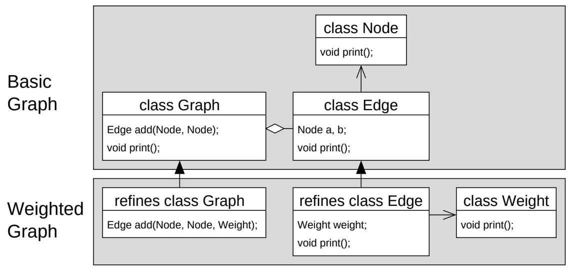

Figure 1. Collaboration-based design of a graph implementation.

and composing object-oriented collaborations when implementing heterogeneous crosscuts [6, 25, 29, 32, 35].

3.2 Homogeneous and Advanced Dynamic Crosscuts

Crosscuts that do not fall in the above category are either homogeneous crosscuts and/or advanced dynamic crosscuts.

Aspects perform well in extending a set of join points using one coherent advice or one localized inter-type declaration, thus, modularizing a homogeneous crosscut. Thereby, programmers avoid code replication. Figure 2 depicts an aspect that implements the feature *Color*, which is homogeneous. It defines an interface for colored entities (Line 2) and declares via inter-type declaration that *Node* and *Edge* implement that interface (Line 3). Furthermore, it introduces via inter-type declarations a field *color* (Line 4) and two accessor methods to *Node* and *Edge* (Lines 5-7,8-10).[4] Finally, it advises the execution of the method *print* of all colored entities to change the display the color (Lines 11-13).

```
1  aspect Color {
2    interface Colored { Color getColor(); }
3    declare parents: (Node || Edge) implements Colored;
4    Color (Node || Edge).color;
5    void (Node || Edge).setColor(Color c) {
6      color = c;
7    }
8    public Color (Node || Edge).getColor() {
9      return color;
10   }
11   before(Colored c) : this(c) && execution(* print()){
12     Color.changeDisplayColor(c.getColor());
13   }
14 }
```

Figure 2. The feature *Color* implemented as aspect.

Advice is well-suited for implementing advanced dynamic crosscuts [29]. When advising the printing mechanism of our graph implementation we can take advantage of the sophisticated mechanisms of AOP. Background is that the *print* methods of the par-

[4] Our notation of inter-type declarations differs from AspectJ. Declaration *int (A || B).i* means that field *i* is introduced to both classes, *A* and *B*.

ticipants of the graph implementation call each other (especially, composite nodes that call *print* of their inner nodes). To make sure that we do not advise all calls to *print*, but only the top-level calls, i.e., calls that do not occur in the dynamic control flow of other executions of *print*, we can use the *cflowbelow* pointcut as conditional (Fig. 3). This is an example of an advanced advice.

```
aspect PrintHeader {
  before() : execution(void print()) &&
  cflowbelow(execution(void print())) { header(); }
  void header() { System.out.print("header:␣"); }
}
```

Figure 3. Advising *print* advanced advice.

Though language abstractions such as *cflow* and *cflowbelow* can be implemented (emulated) using traditional OOP, usually that results in code replication, tangling, and scattering.

3.3 Discussion

Table 2 depicts the guidelines for using AOP and OOP mechanisms based on their individual strengths. First, aspects should be used for modularizing homogeneous crosscuts to avoid code replication. Second, aspects avoid code scattering and tangling in case of using advanced advice for implementing advanced dynamic crosscuts. For heterogeneous crosscuts which extend only methods and classes, OOP techniques for collaboration-based designs suffice. It has been observed that although both approaches are able to implement the crosscuts of the other, they cannot do so elegantly [2, 3, 29].

	heterogeneous	**homogeneous**
static	set of roles that add elements	inter-type declaration
basic dynamic	set of roles that override methods	basic advice
advanced dynamic	set of advanced advice	advanced advice

Table 2. What implementation technique for what kind of crosscutting concern?

4. Metrics

We propose a set of metrics to provide insight into the current practice of AOP. They enable to decide in which category a given aspect falls. The metrics are quantified by the *number of occurrences (NOO)* of a certain software artifact and/or the *lines of code (LOC)* associated with it.

Classes, interfaces, and aspects (CIA). The CIA metric determines the NOO of classes, interfaces, and aspects, as well as the LOC associated with each. It tells us if aspects (as opposed to classes and interfaces) are a small or a large fraction of the used modularization mechanisms in a software project, and if these implement a significant or only a small part of the code base of that project.

Homogeneous crosscuts (HC). The HC metric measures the extent in which homogeneous and heterogeneous crosscuts are used. We calculate the fraction of advice and inter-type declarations that implement homogeneous crosscuts (NOO) and the fraction of the code base that is associated with them (LOC). The HC metric tells us if the aspects of a program exploit the pattern-matching mechanisms of AOP or merely emulate OOP mechanisms.

Advanced dynamic crosscuts (ADC). This metric determines the NOO of advanced advice and the overall LOC associated with them.[5] It tells us to what extent aspects make use of the advanced capabilities of AOP for implementing dynamic crosscuts.

Code replication reduction (CRR). The CRR metric determines the reduction in LOC when using homogeneous advice and inter-type declarations, as opposed to the LOC resulting from using traditional OOP mechanisms. The code reduction for one piece of homogeneous advice / inter-type declaration is roughly the number of affected join points, multiplied by the LOC associated with them.

5. Collecting Statistics of AspectJ Programs

CIA metric. Collecting data for the CIA metric we traverse all source files of a given project and calculate the number and LOC of aspects, classes, and interfaces – excluding blank lines and comments.

HC metric. Homogeneous crosscuts are indicated by inter-type declarations and advice that contain wildcards (* and +). If we discover logical operators in pointcuts that combine two pointcuts of the same type (e.g., *execution(...)* || *execution(...)*) then the associated advice are also counted as homogeneous. Inter-type declarations that contain logical operators are considered homogeneous as well as advice that do not qualify a target method or field completely, e.g., by omitting the type or the arguments.

ADC metric. We define all advice as advanced advice except those associated to *call* and *execution* and that are not combined with any other pointcuts, except with *target* and *args* (*execution* can also be combined with *this*). This is an overestimation that might consider some pieces of advice that are not advanced as advanced advice, but not vice versa. The remaining advice are considered basic advice.[6]

CRR metric. For determining the code reduction due to eliminating replicated code, we determine the number of join points per homogeneous advice and inter-type declaration. We multiply the number of join points minus one for each advice or inter-type declaration, with the LOC associated. Finally, we sum up the saved LOC of all advice and inter-type declarations to get the overall code reduction.

6. A Case Study

As case study we analyzed *FACET* (6364 LOC), an AspectJ-based CORBA event channel, implemented at the Washington University [18]. We used our tool *AJStats*[7] for collecting the NOO and LOC of all artifacts of FACET. We determined the properties of advice / inter-type declarations and the caused code reduction by hand.

Table 3 depicts our collected statistics. Column *NOO* lists the number of artifacts we found of a specific type (e.g., homogeneous advice) and its fraction with regard to the overall number of this type (e.g., all pieces of advice). Column *LOC* depicts the LOC associated with a certain kind of artifact and its fraction of the overall code base. In the following paragraphs we examine the data in depth.

[5] Recall that advanced advice can be either heterogeneous or homogeneous (cf. Fig. 1).

[6] Although the semantics of *call* is to advise the client side invocations of a method, it can be implemented as method extension – preconditioned that *all* calls to the target method are advised; the above definition ensures that.

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

metric		NOO (% of artifacts)	LOC (% of code base)
CIA	classes/int.	181 (62%)	5143 (81%)
	aspects	113 (38%)	1221 (19%)
HC	heterogen.	150 (93%)	572 (9%)
	homogen.	12 (7%)	24 (0.4%)
ADC	basic	38 (78%)	187 (3%)
	advanced	11 (22%)	110 (2%)
CRR	adv. + itds	—	534 (8%)

Table 3. FACET statistics.

CIA metric. FACET uses relatively many aspects, compared to other studies [2, 8, 9, 21, 38]. This observation is remarkable since it demonstrates that aspects are used in different software projects to a different extent. 38% of all modularization mechanisms were aspects, which occupied 19% of the overall code base.

HC metric. In FACET we found 4 of 49 pieces of advice and 8 of 113 inter-type declarations were homogeneous.[8] That is, 7% of all implemented crosscuts were homogeneous, which occupied 0.4% of the overall code base. In contrast, 93% of all crosscuts were heterogeneous, occupying 9% of the code base.

ADC metric. We found 11 of 49 advice were advanced advice. They are associated to *cflow* pointcuts or use the *returning* clause. That is, 22% of all advice were advanced advice, which occupied 2% of the overall code base. The remaining 38 advice were basic advice, which occupied 3% of the overall code base.

CRR metric. 4 pieces of advice and 8 inter-type declarations are homogeneous. We calculated the effective code reduction of 534 LOC, which is a 8% reduction compared to a version that uses OOP mechanisms for implementing homogeneous crosscuts.

7. Related Work

AOP case studies. Colyer and Clement refactored an application server using aspects [9]. 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 collaborations.

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [8]. They factored 4 concerns and evolved them in three steps; inherent properties of concerns were not explained in detail.

Lohmann et al. examined the applicability of AOP to embedded infrastructure software [21]. For their study they factored 3 concerns of a commercial embedded operating system; 2 concerns were homogeneous and 1 heterogeneous.

Lopez-Herrejon et al. explored the ability of AOP to implement product lines [22]. They demonstrated how collaborations are translated automatically to aspects. They did note that less than 1% of their code base was attributable to heterogenous advice. They did 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 [37]. They reimplemented FACET by replacing aspects with *Jiazzi units*, which encapsulate collaborations. They do not examine the structure of the resulting collaborations. Our analysis of FACET revealed that some crosscuts should be implemented using aspects.

[8] Note that the code associated to advice and inter-type declarations (596 LOC) is only a subset of the overall aspect code (1221 LOC), which includes also fields, methods, etc.

Metrics for AOP. Zhang and Jacobson used a set of object-oriented metrics to quantify the program complexity reduction when using AOP for implementing middleware [38].

Garcia et al. applied seven metrics to Hannemann's [17] implementation of design patterns [15]. They found that most aspect-oriented solutions improve separation of pattern related concerns, although only 4 aspect-oriented implementations have exhibited significant reuse.

Zhao and Xu propose several metrics for aspect cohesion based in aspect dependency graphs [39]. Ceccato and Tonella propose metrics for measuring the coupling degree between program elements [7].

None of the above metrics and case studies take the different structure of crosscutting concerns into account. We argue that the structure of a concern decides over how it is implemented best.

8. Conclusions

Comparatively many aspects were used in FACET and they sum up to a significant part of the code base (19%) – but only 3% of the overall code base exploits the advanced capabilities of AOP to implement homogeneous and dynamic crosscuts. 97% can be implemented straightforward using traditional OOP and collaborations. Nevertheless, the used AOP mechanisms reduce the code base by 8% compared to an OOP implementation. It follows that the plain number of aspects, advice, etc. is not meaningful to judge the successful application of AOP to a software project.

Our classification framework, categories, and metrics form a quantitative basis for analyzing aspect-oriented programs in this respect, and it can assist in exploiting the benefits of AOP. In further work we intend to analyze and compare further AOP projects to collect more data.

Acknowledgments

Sven Apel is sponsored by the German Research Foundation (DFG), project number SA 465/31-1 and SA 465/32-1. Marko Rosenmüller is sponsored by the German Research Foundation (DFG), project number SA 465/32-1. The presented study was conducted when Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. Batory's research is sponsored by NSF's Science of Design Project #CCF-0438786.

References

[1] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[2] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of International Conference on Generative Programming and Component Engineering*, 2006.

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

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

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

[6] 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*, 1990.

[7] M. Ceccato and P. Tonella. Measuring the Effects of Software Aspectization. In *Workshop on Aspect Reverse Engineering*, 2004.

[8] 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*, 2003.

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

[10] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.

[12] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] A. Garcia et al. Separation of Concerns in Multi-agent Systems: An Empirical Study. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications*, 2003.

[14] A. Garcia et al. Modularizing Design Patterns with Aspects: a Quantitative Study. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[15] A. Garcia et al. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[16] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-based Crosscuts. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2003.

[17] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[18] 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*, 2002.

[19] K. Lieberherr. Controlling the Complexity of Software Designs. In *Proceedings of International Conference on Software Engineering*, 2004.

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

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

[22] 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.

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

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

[25] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[26] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2003.

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

[28] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[29] 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*, 2004.

[30] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of European Conference on Object-Oriented Programming*, 2005.

[31] T. Reenskaug et al. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6), 1992.

[32] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2), 2002.

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

[34] F. Steimann. Domain Models are Aspect Free. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, 2005.

[35] 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*, 1996.

[36] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 26(5), 2004.

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

[38] 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*, 2004.

[39] J. Zhao and B. Xu. Measuring Aspect Cohesion. In *Proceeding of International Conference on Fundamental Approaches to Software Engineering*, 2004.