On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns

Sven Apel, Christian Kästner
University of Magdeburg
{apel, kaestner}@iti.cs.uni-magdeburg.de

Salvador Trujillo
University of the Basque Country
struji@ehu.es

Abstract

Collaborations are a frequently occurring class of crosscutting concerns. Prior work has argued that collaborations are better implemented using Collaboration Languages (CLs) *rather than* AspectJ-like Languages (ALs). *The main argument is that aspects flatten the object-oriented structure of a collaboration, and introduce more complexity rather than benefits – in other words, CLs and ALs differ with regard to program comprehension. To explore the effects of CL and AL modularization mechanisms on program comprehension, we propose to conduct a series of experiments. We present ideas on how to arrange such experiments that should serve as a starting point and foster a discussion with other researchers.*

1. Introduction

As well-established programming paradigms such as *object-oriented programming (OOP)* fail to modularize crosscutting concerns appropriately a plethora of new modularization mechanisms emerges. Unfortunately, recent advances have been made along different lines of research and the lack of a uniform taxonomy contributed to a general confusion about crosscutting concerns and their relationship to *collaborations*, which we address here.

In prior work we [2,4,23] and others [11,26] helped to alleviate this situation by categorizing crosscutting concerns. We identified two main classes of crosscutting concerns: while *collaborations* are sets of changes applied to multiple classes that introduce new classes, new members to classes, and that extend existing methods, *advanced aspects* apply either homogeneous changes (applying the same changes to multiple points in a program) or dynamic changes (applying changes depending on the runtime control flow or the value of runtime variables).

Furthermore, we suggested programming guidelines that assist programmers in deciding when to use what modularization mechanism [2, 4]. We inferred these guidelines from the individual strengths and weaknesses of contemporary CLs and ALs: (1) use *collaboration languages (CLs)*, e.g., *Jak* [6], *Jx* [27], *Classbox/J* [7], *Scala* [28], for collaborations and (2) use *AspectJ-like languages (ALs)*, e.g., *AspectJ* [18], *AspectC++* [36], *Eos* [29], for advanced aspects.[1]

While these programming guidelines could be evaluated successfully by means of a non-trivial case study [3], a couple of open issues remains. Some conclusions incorporated were based on arguments that are controversial [30,37]. For example, the claim that a collaboration should be explicit instead of merged into an aspect is reasonable but not supported well. There are many such arguments in favor of CLs or of ALs. We realized that the discussion is often driven by personal preferences and plausibility arguments, often leading to entrenched positions. So we ask, is there any way to support or even prove such claims? Are CLs or ALs equally suited to implement collaborations or is one superior? These issues are important because prior studies indicated that collaborations occur frequently in contemporary programs [2, 3].

In this paper we illustrate the commonalities and subtle differences of implementing collaborations via CLs and ALs. We conclude that CLs and ALs are very similar in this matter. CLs and ALs differ only in the arrangement and interplay of language constructs and their concrete syntax. While these differences cannot be quantified by code metrics, they are significant for program comprehension especially for large-scale collaborations [35]. In the end, it is the programmer who has to understand the resulting software. Thus, we argue that empirical studies that incorporate programmers are the key to decide between CLs and ALs.

The aim of this paper is to sensitize researchers to the necessity for empirical studies in this field. We outline our first ideas on how to arrange empirical studies that should serve as a starting point for further discussion.

[1] For simplicity, we do not consider languages that can be assigned to both, CLs and ALs, e.g., *CaesarJ* [26], *FeatureC++* [5], *Hyper/J* [40]

First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)
0-7695-2967-4/07 $20.00 © 2007 IEEE

2. Background

We begin with the introduction of the concept of a collaboration and how it can be implemented using CLs and ALs. Note that we do not consider crosscutting concerns that demand capabilities of advanced aspects, i.e., pattern matching or control flow quantification. For those, ALs clearly provide better language support [2, 3, 26].

2.1. Collaborations

An object in OOP is a self-contained entity that encapsulates data and a set of accompanying operations. In order to complete a task, objects need to cooperate with other objects. Such intended interactions are also called *collaborations*. A collaboration is a set of objects and a protocol to communicate with one another [6, 12, 13, 21, 26, 28, 31, 35, 42]. While in a dynamic interpretation objects are the relevant entities, we favor a static view which emphasizes the interactions given by their classes [35]. Hence, the protocol is fixed at compile time and defines how objects interact.

Classes play different *roles* in different collaborations [42]. A role encapsulates the behavior or functionality that a class provides when a corresponding collaboration with other classes is established. That is, a role is that part of a class that implements the communication protocol with other classes participating in a particular collaboration.

In Figure 1 we show four classes participating in three collaborations. For example, class *A* participates in collaboration *I* and *II*, i.e., two distinct roles implement the communication protocol necessary for these collaborations.

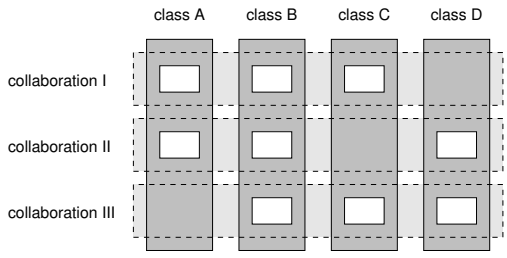

Figure 1. Collaboration-based design.

Usually when added to a program, a collaboration adds several new classes and applies several new roles to existing classes simultaneously. A role applied to a class may add new elements to a class and may extend existing elements, such as methods. Hence, a collaboration cuts across several places in a base program.

In prior work we noticed the close relationship to crosscutting concerns [3, 4, 23]. We defined collaborations as one of two fundamental categories of crosscutting concerns – the other category consists of advanced aspects, which use mechanism for pattern-matching and control flow quantification. Since a role adds new elements to a single class and extends methods only, there is no need for advanced aspect mechanisms.

In Figure 2 we depict the collaboration-based design of a simple program that deals with graph data structures. The diagram uses the *UML* notation [9] with some extensions: white boxes represent classes or roles; gray boxes denote collaborations; solid arrows denote the application of a new role to a class.

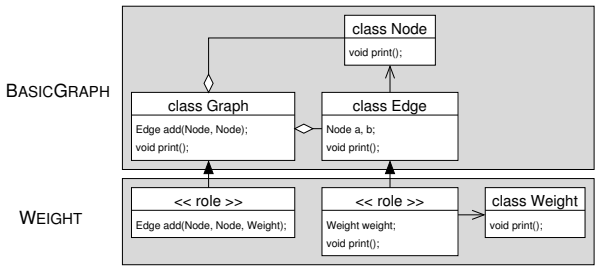

Figure 2. Graph example.

The collaboration BASICGRAPH consists of the classes `Graph`, `Node`, and `Edge` that together provide functionality to construct and display graph structures[2]. The collaboration WEIGHT adds a role to `Graph` and to `Edge` as well as a class `Weight`, which all together implement a weighted graph.

2.2. Implementation

We discuss two alternative ways to implement collaboration-based designs.

Collaboration Languages. To illustrate how CLs implement collaborations we choose *Jak*[3] as an archetype [6]. *Jak* extends Java by a special language construct to apply roles to classes, called *refinements*. Classes in Jak are implemented as standard Java classes. In Figure 3 we depict our collaboration BASICGRAPH implemented in Jak. It consists of the classes `Graph` (Lines 1–14), `Node` (Lines 15–19), and `Edge` (Lines 20–27).

A refinement in Jak implements a role intended for a class. It is declared by the keyword `refines`. It can add new members to a class and extend existing methods, which is called a *method extension*. A method extension is implemented by method overriding and calling the overridden method via the keyword `Super`.

[2] In this paper we write collaboration names in SMALL CAPS and names of classes, methods, and fields in `Typewriter`.

[3] http://www.cs.utexas.edu/users/schwartz/ATS.html

0-7695-2967-4/07 $20.00

```
1  class Graph {
2    Vector nodes = new Vector();
3    Vector edges = new Vector();
4    Edge add(Node n, Node m) {
5      Edge e = new Edge(n, m);
6      nodes.add(n); nodes.add(m); edges.add(e);
7      return e;
8    }
9    void print() {
10     for(int i = 0; i < edges.size(); i++) {
11       ((Edge)edges.get(i)).print();
12     }
13   }
14 }
15 class Node {
16   int id = 0;
17   Node(int _id) { id = _id; }
18   void print() { System.out.print(id); }
19 }
20 class Edge {
21   Node a, b;
22   Edge(Node _a, Node _b) { a = _a; b = _b; }
23   void print() {
24     a.print(); System.out.print(", ");
25     b.print(); System.out.print("; ");
26   }
27 }
```

Figure 3. A simple graph (BASICGRAPH).

In Figure 4 we depict the collaboration WEIGHT implemented in Jak. It introduces a new class `Weight` that represents the weight of an edge (Lines 19) and refines (applies a role to) the class `Graph` (Lines 1–10) by introducing a new method `add` that assigns a given weight value to an edge (Lines 2–5) and by extending the existing method `add` to assign a default weight value (Lines 6–9). Furthermore, it refines the class `Edge` (Lines 11–18) by adding a field (Line 12) and a method for assigning the weight value (Line 13) and by extending the `print` method to display the weight (Lines 14–17).

```
1  refines class Graph {
2    Edge add(Node n, Node m, Weight w) {
3      Edge res = Super.add(n, m);
4      res.setWeight(w); return res;
5    }
6    Edge add(Node n, Node m) {
7      Edge res = Super.add(n, m);
8      res.setWeight(new Weight(0)); return res;
9    }
10 }
11 refines class Edge {
12   Weight weight;
13   void setWeight(Weight _w) { w = _w; }
14   void print() {
15     Super.print();
16     weight.print();
17   }
18 }
19 class Weight { /* ... */ }
```

Figure 4. A weighted graph (WEIGHT).

In Jak a collaboration of roles and classes is represented by a *containment hierarchy* [6], which is a directory that contains all software artifacts that implement a program feature. Thus, it has no textual representation at the code level but is explicit for the programmer. The artifacts found inside a containment hierarchy are the classes and roles of the enclosing collaboration.

AspectJ-like Languages. The same collaboration can be implemented with ALs. We use *AspectJ*[4] for illustration. As with CLs, BASICGRAPH is implemented by three classes (cf. Fig. 3). What differs is the way to apply a new collaboration to an existing program and to apply new roles to existing classes. There are two alternative ways to achieve this.

One approach is to introduce an aspect that bundles a set of inter-type declarations and pieces of advice. In Figure 5 we show an aspect for WEIGHT that introduces the method `add` (Lines 2–5), the field `weight` (Line 10), and the method `setWeight` (Line 11) via inter-type declarations and extends the method `add` (Lines 6–9) and `print` method via advice (Lines 12–15). The class `Weight` is defined as static inner class (Line 16).

```
1  aspect AddWeight {
2    Edge Graph.add(Node n, Node m, Weight w) {
3      Edge res = add(n, m);
4      res.setWeight(w); return res;
5    }
6    after() returning(Edge res) :
7      execution(Edge Graph.add(Node, Node)) {
8      res.setWeight(new Weight(0));
9    }
10   Weight Edge.weight;
11   void Edge.setWeight(Weight _w) { w = _w; }
12   after(Edge e) :
13     execution(void Edge.print()) && this(e) {
14     e.weight.print();
15   }
16   public static class Weight { /* ... */ }
17 }
```

Figure 5. One aspect per collaboration.

Another approach to implement a collaboration is to implement each role by a distinct aspect [15, 17, 34]. In our example we would implement the roles applied to `Graph` and `Edge` as two distinct aspects and the class `Weight` as top-level class, as shown in Figure 6.

3. Discussion

Table 1 lists the mechanisms of CLs and ALs for implementing collaborations and their roles; there seem to be only a few differences between CLs and ALs:

1. With CLs we can add an encapsulated set of classes and roles. While an aspect can implement and encapsulate a set of roles it cannot introduce new classes by

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

0-7695-2967-4/07 $20.00

```
1  aspect AddWeightGraph {
2    Edge Graph.add(Node n, Node m, Weight w) {
3      Edge res = add(n, m);
4      res.weight = w; return res;
5    }
6    after() returning(Edge res) :
7      execution(Edge Graph.add(Node, Node)) {
8      res.setWeight(new Weight(0));
9    }
10 }
11 aspect AddWeightEdge {
12   Weight Edge.weight;
13   void Edge.setWeight(Weight _w) { w = _w; }
14   after(Edge e) :
15     execution(void Edge.print()) && this(e) {
16     e.weight.print();
17   }
18 }
19 class Weight { /* ... */ }
```

Figure 6. One aspect per role.

	roles	**collaborations**
AL	one aspect per role or one aspect per multiple roles	one aspect per collaboration or multiple aspects per collaboration
CL	one refinement per role	a set of refinements per collaboration

Table 1. Implementation mechanisms.

itself [25]. Hunleth et al. have shown that for managing collaborations a programmer needs to supply a non-trivial infrastructure, e.g., adjusting the build path, file system management, feature registry [16]. Grouping aspects and classes within packages or as inner elements hinders composing collaborations and using classes from outside [3].

2. While CLs represent roles as class-like entities that have the same name as the classes they extend, with ALs (1) the constituent parts of different roles are merged in one aspect, without direct mapping to the classes to be extended, or (2) each role is represented by a distinct aspect.
3. The syntax of defining roles differs significantly between CLs and ALs. While defining a role with a CL enables to access all members of the extended class ("*first-person perspective*"), with an AL a role has to declare explicitly what context is exposed ("*third-person perspective*"), e.g., by using pointcuts like `args`, `this`, and `target`.

Even though these differences have never been made explicit several researchers argued in favor or against CLs and ALs. The main argument pushed forward is the effect on program comprehension, which has a most significant impact on software development. In the remaining section we review some arguments.

3.1. Explicitness and Scalability

It has been noticed that not expressing a collaboration in terms of object-oriented design (i.e., roles implemented as refinements) decreases program comprehensibility [3,26, 38], although this is not generally accepted [17,30]. Detaching roles from classes is problematic because programmers cannot recognize the original structure of the base program within a subsequently applied collaboration – in our example the structuring in `Graph`, `Node`, and `Edge`. Especially, the direct and intuitive mapping between roles and classes and its hierarchical character is supposed to simplify program comprehension.

For our simple example, it does not really matter whether a programmer uses Jak or AspectJ. The difference between CLs and ALs becomes more obvious when considering collaborations at a larger scale. Suppose a base program consists of many classes (say 15) and a collaboration extends most of them (say 12). With CLs the programmer defines, per class to be extended, a new role with the same name (Fig. 7). This way the programmer is able to retrieve the program structure within the new collaboration. There is a one-to-one mapping between the structural elements of the base program and the elements of the collaboration to be applied; base program and new collaboration are merged recursively by name and type [6, 28, 35]. Roles are inherently object-oriented and CLs make them explicit [37].

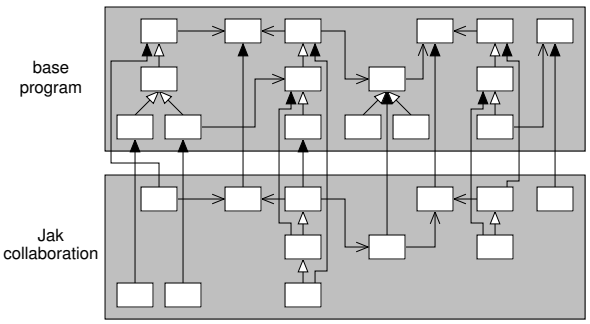

Figure 7. *N* refinements per collaboration.

One aspect per collaboration. The primary argument against aspects is their lack of scalability with respect to an increasing number of roles [2]. In an AL solution all participating roles of a collaboration are merged into one aspect (Fig. 8). While this is possible, it flattens the inherent object-oriented structure of the collaboration and makes it hard to trace the mapping between the base program and the collaboration [26, 38]. Note that the difference between both solutions, as shown in the Figures 7 and 8, is not only a matter of visualization. The point is that the inner structure of the aspect does not reflect the structure of the base program; the mapping between structural elements of the base

0-7695-2967-4/07 $20.00

program and the collaboration is complex since the roles are not explicit and their constituents are scattered over the entire aspect in an arbitrary order. The programmer has to translate constantly between base program and collaboration. This is difficult and does not scale with large collaborations, which do occur [6, 41]

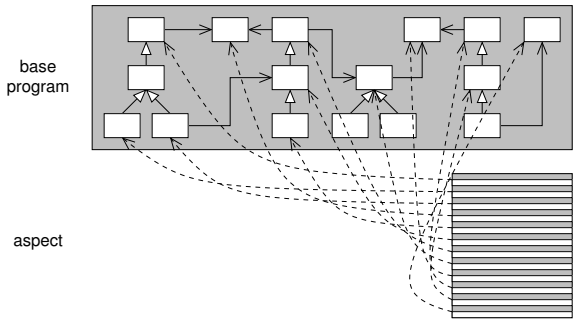

Figure 8. One aspect per collaboration.

One aspect per role. Implementing each role as a distinct aspect [15,17,30,34] would enable to establish a one-to-one mapping between the elements of the base program and the elements of the collaboration (provided reasonable naming conventions). Thereby, an AL solution would be very similar to a CL solution. However, the mapping between classes and roles is not enforced by the programming language and thus left to the discipline of the programmer.

Furthermore, the way inheritance and roles are replaced by aspect weaving does not provide any additional benefit. It has been argued that such a replacement of object-oriented techniques without any benefit is questionable [20, 26, 38], especially with respect to the additional program complexity introduced by aspects [1, 39].

3.2. Conciseness of Syntax

A further argument aims at the expressive but complex syntax of ALs such as AspectJ and at the different perspectives a role has on a class (first-person vs. third-person perspective). An argument in favor of ALs is that a program can be extended at many points by using sophisticated mechanisms for pattern matching and dynamic crosscutting. However, in a collaboration (in the sense of our definition) these are not necessary because they unnecessarily complicate the expression of a role. For example, when extending a method via advice we need 4 pointcut designators[5] and their arguments are repeated and bound in a non-trivial way. This is because of the third-person perspective of ALs. Due to the first-person perspective, the Jak solution appears simpler, as illustrated in Figure 9.

[5] `within` prevents advising methods of subclasses.

```
1 refines class Edge {
2   int compare(Edge e) {
3     int res = Super.compare(e); /* ... */
4   }
5 }
```

```
1 aspect CompareAspect {
2   int around(Edge t, Edge e) : this(t) && args(e) &&
3     execution(int Edge.compare(Edge)) && within(Edge) {
4     int res = proceed(t, e); /* ... */
5   }
6 }
```

Figure 9. Method extension (Jak vs. AspectJ).

4. An Empirical Approach

The discussion about the differences between CLs and ALs suggests that CLs are better suited for implementing collaborations than ALs and that ALs bear even the potential to introduce more problems than solutions. However, the arguments pushed forward are mainly plausibility arguments, even though reasonable. Furthermore, the entire discussion could be biased by personal preferences.

Nevertheless, this discussion is very important since collaborations occur frequently and this topic is still controversial. Previous studies could not prove that one solution (CL or AL) is superior, especially with regard to program comprehension. Thus, we need an empirical methodology. The programmer and his connection to the program code is in the heart of this discussion. Consequently, we believe that empirical studies help to find a definitive answer.

In the remaining paper we address the issue of how to conduct and arrange such studies. Our considerations should be regarded as a first step and are intended to serve as a starting point for discussion with other researchers.

4.1. The Cognitive Distance

The appropriateness of a modularization mechanism to implement a design or implementation problem can be evaluated in terms of the intellectual effort required to use and understand it [19]. The *cognitive distance* is an intuitive gage that helps to assess modularization mechanism, especially when other metrics fail. It is defined as the amount of intellectual effort that a programmer must expend to understand and use a modularization mechanism [19].

Applied to our problem the cognitive distance is exactly the way of comparing CLs and ALs we are looking for. However, the cognitive distance is not a formal measurement that can be quantified by numbers. It is rather an intuitive measure that gages the effort to use and understand a mechanism and the resulting program. But if the cognitive distance is not a formal metric how can we measure

0-7695-2967-4/07 $20.00

it? The answer has been given many times in other fields of computer science and other sciences. If human beings are involved, empirical studies that observe and interview programmers can help. Regarding our problem we need to make experiments that allow to draw conclusions about the programmer's ability to understand a program.

4.2. Experiments

As methodology for the envisioned experiments we suggest an *ethnographic approach* [14]. In an ethnographic approach the researchers participate themselves in the experiments; they join conversations, attend meetings, and read documents. It minimizes preconceptions of experts and novices by considering all activities as "strange". There is no fixed set of data being observed during the experiments. Every *theme* that occurs during discussion is challenged. Any data occurs naturally by deriving it from the observation of the participants that gave no a priori significance to any particular issue. Ethnographic studies have been proven useful for empirical software engineering [8, 10, 32, 33].

We plan to conduct an ethnographic study to collect data about the differences of CLs and ALs with regard to program comprehension. There are many ways to organize such a study and here we outline first ideas.

An experiment that is part of a study confronts the programmer with program text. There are two ways to do so: (1) the programmer gets the task to implement a collaboration-based design from a specification; (2) the programmer has to derive the meaning (specification) of a collaboration-based design from a given implementation. In both ways either the programmers have to perform the task with an CL and AL or a group of programmer perform the task with an CL and another group with an AL.

Since we suggest an ethnographic approach the experiments are discussed afterward with the other programmers and the researchers that conduct the study. Of course, themes like the complexity of the program structure and the time the programmer needed for the task are the first themes of the discussion. Therefrom new themes will emerge that can shed a new light on this matter. For example, it might be that a programmer gets quicker to a preliminary result but the debugging effort is high; or a programmer believed quickly to understand a given collaboration-based design but he missed important points. These emergent themes are difficult to predict and in fact a goal of this study.

Having said this, it is important to discuss how a reasonable collaboration-based design (i.e., specification and implementation) should look like. Certainly, it is not enough to chose a design arbitrarily. We identified two parameters of collaboration-based design that can help to distinguish CLs and ALs: *scale* and *role mapping*.

Scalability is the main argument against using ALs. Experiments should vary the size of the base program, the number of collaborations, the number of roles per collaboration, and the size and complexity of the individual roles in terms of added and extended elements.

The direct mapping between a role and its class seems to be significant for the programmer to recognize this connection. While CLs enforce a strict, simple, and explicit mapping, with ALs it is left to the programmer but grants for flexibility. Varying the arrangement of roles in an aspect can reveal the impact of the clear mapping on program comprehension, compared to a strict CL solution.

Varying the scale and role mapping allows us to compare the effort programmers have to spend when writing and understanding CL and AL programs. Subsequent discussions might reveal new issues that emerge from the discussion of the influence of the different parameters.

For the purpose of this position paper we envision a first collaboration-based design. For a first series of experiments we suggest to use the graph example outlined in this paper. Even though the presented excerpt of the graph example is rather small, there already exists an collaboration-based design of an extended version. It consists of 441 lines of code implementing 13 collaborations that crosscut 19 Java classes, and there are already reference implementations in Jak and AspectJ available. Though this would be a first example it is still comparatively small. However, larger collaboration-based designs and their implementations are available [3, 22, 24, 41, 43].

5. Conclusion

Collaborations are frequently occurring design and implementation problems. In this paper we illustrated that collaborations can be implemented either by using CLs or ALs. Currently, there is a lively discussion about the pros and cons of CL and ALs but most arguments are based on plausibility and personal preferences. We illustrated that beside analytical methods the programmer is a key factor and needs to be considered when comparing CLs and ALs.

We presented rationale to believe that CLs lead to more comprehensible programs than ALs. Consequently, we propose an empirical methodology to support our claim. A main subject to measure is the cognitive distance between programmer and program code. Since this is an informal measure we need to make several experiments varying certain parameters, e.g., scale and role mapping. We suggest an ethnographic approach for experiments to minimize the effect of expertise and preconceptions of programmer and to examine which issues are relevant for program comprehension.

Acknowledgements. We thank Don Batory for his fruitful comments on earlier drafts of this paper. Sven Apel

is sponsored by the German Research Foundation (DFG), project number SA 465/32-1.

References

[1] R. Alexander. The Real Costs of Aspect-Oriented Programming. *IEEE Software*, 20(6), 2003.

[2] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Science, University of Magdeburg, 2007.

[3] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proc. of GPCE*, 2006.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proc. of ICSE*, 2006.

[5] S. Apel, M. Rosenmüller, T. Leich, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. of GPCE*, 2005.

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

[7] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. of OOPSLA*, 2005.

[8] P. Beynon-Davies, D. Tudhope, and H. Mackay. Information Systems Prototyping in Practice. *Journal of Information Technology*, 14(1), 1999.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Professional, 2005.

[10] G. Button and W. Sharrock. Project Work: The Organisation of Collaborative Design and Development in Software Engineering. *Computer Supported Cooperative Work*, 5(4), 1996.

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

[12] E. Ernst. Family Polymorphism. In *Proc. of ECOOP*, 2001.

[13] E. Ernst. Higher-Order Hierarchies. In *Proc. of ECOOP*, 2003.

[14] M. Hammersley and P. Atkinson. *Ethnography: Principles in Practice*. London: Tavistock, 1983.

[15] S. Hanenberg and R. Unland. Roles and Aspects: Similarities, Differences, and Synergetic Potential. In *Proc. of OOIS*, 2002.

[16] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proc. of LCTES/SCOPES*, 2002.

[17] E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *Proc. of OOPSLA*, 1999.

[18] G. Kiczales et al. An Overview of AspectJ. In *Proc. of ECOOP*, 2001.

[19] C. W. Krueger. Software Reuse. *ACM CSUR*, 24(2), 1992.

[20] K. Lieberherr. Controlling the Complexity of Software Designs. In *Proc. of ICSE*, 2004.

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

[22] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proc. of ICSE*, 2006.

[23] R. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In *Proc. of FASE*, 2007.

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

[25] R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. of ECOOP*, 2005.

[26] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *Proc. of FSE*, 2004.

[27] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. of OOPSLA*, 2004.

[28] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. of OOPSLA*, 2005.

[29] H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *Proc. of ICSE*, 2005.

[30] A. Rashid and A. Moreira. Domain Models Are NOT Aspect Free. In *Proc. of MoDELS/UML*, 2006.

[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] H. Sharp and H. Robinson. An Ethnographic Study of XP Practice. *Empirical Software Engineering*, 9(4), 2004.

[33] H. Sharp, H. Robinson, and M. Woodman. Software Engineering: Community and Culture. *IEEE Software*, 17(1), 2000.

[34] M. Sihman and S. Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5), 2003.

[35] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.

[36] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's Journal*, 2005.

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

[38] F. Steimann. Domain Models are Aspect Free. In *Proc. of MoDELS/UML*, 2005.

[39] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Proc. of OOPSLA*, 2006.

[40] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. of ICSE*, 1999.

[41] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proc. of GPCE*, 2006.

[42] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proc. of OOPSLA*, 1996.

[43] B. Xin, S. McDirmid, E. Eide, and W. C. Hsieh. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, The University of Utah, 2004.