How AspectJ is Used: An Analysis of Eleven AspectJ Programs

Sven Apel[†] and Don Batory[‡]

[†] Department of Informatics and Mathematics, University of Passau
apel@uni-passau.de

[‡] Department of Computer Sciences, University of Texas at Austin
batory@cs.utexas.edu

Technical Report, Number MIP-0801
Department of Informatics and Mathematics
University of Passau, Germany
April 2008

How AspectJ is Used: An Analysis of Eleven AspectJ Programs

Sven Apel[†] and Don Batory[‡]

[†] Department of Informatics and Mathematics, University of Passau
apel@uni-passau.de
[‡] Department of Computer Sciences, University of Texas at Austin
batory@cs.utexas.edu

Abstract. While it is well-known that *crosscutting concerns* occur in many software projects, little is known on how *aspect-oriented programming* and in particular AspectJ have been used. In this paper, we analyze eleven AspectJ programs by different authors to answer the questions: which mechanisms are used, to what extent, and for what purpose. We found the code of these programs to be on average 86 % object-oriented, 12 % basic AspectJ mechanisms (introductions and method extensions), and 2 % advanced AspectJ mechanisms (homogeneous advice or advanced dynamic advice). There is one class of crosscutting concerns – which is mostly concerned with introductions and method extensions – that matches this result well: *collaborations*. These results and our discussions with program authors indicate the bulk of coding activities was implementing collaborations. Several studies and researchers suggest that languages explicitly supporting collaborations are better suited than aspects à la AspectJ for this task.

1 Introduction

While many studies have explored the capabilities of *aspect-oriented programming (AOP)* [1] to improve the modularity, customization, and evolution of software [2, 3, 4, 5, 6, 7, 8], little is known on *how* AOP has been used. As *AspectJ*[1] is the most widely used AOP language, we want to know which AspectJ mechanisms are used, to what extent, and for what kinds of crosscutting concerns.

Although the first versions of AspectJ were released over seven years ago and there have been a large number of downloads of the *ajc* tool (in January 2007 alone there were 13,021 downloads), we and others [9] have noted that there are only a few published, non-trivial programs using AspectJ in open literature. With the help of colleagues, we have been able to locate eleven different AspectJ programs authored at different universities, deliberately excluding our own case studies [10, 11, 12, 13]. These programs range in size from small programs of 1 KLOC to larger programs of almost 130 KLOC.

AspectJ offers a variety of programming mechanisms [14]. *Basic mechanisms* are simple *introductions* (a.k.a. inter-type declarations) and method extensions;

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

advanced mechanisms include homogeneous advice and advanced advice, which are intended for a wide variety of sophisticated dynamic join points [15]. But how often are basic and advanced mechanisms actually used? And for what purpose?

In this paper, we define metrics to answer these questions. Our analysis of the eleven programs shows on average 86 % of the code is object-oriented, 12 % uses basic mechanisms, and 2 % uses advanced mechanisms. This is the first time to our knowledge that a reasonable number of AspectJ programs has been analyzed and actual percentages reported.

The usage distribution (86 %, 12 %, 2 %) raises a question: what kind of crosscutting concern is represented predominantly by introductions and simple method extensions? Our answer is collaborations. A *collaboration* is a protocol that defines the inter-class communication necessary to complete a task [16, 17, 18, 19, 20, 21, 22, 23]. A *role* encapsulates the protocol that a class provides when a collaboration with other classes is established. When a role is added to a class, new members are introduced and existing methods are extended.

These results and our discussions with the program authors indicate that the bulk of coding activities is implementing collaborations. This is important, as there are programming languages that are explicitly support collaborations, though realized in different ways, e.g., *Jak* [20], *FeatureC++* [24], *ContextL* [25], *Scala* [22], *Jiazzi* [26], *Classbox/J* [27], *Jx* [28]. Although AspectJ can be used to implement collaborations, as well [29, 30, 31, 32], it is designed to modularize all kinds of crosscutting concerns, at the expense of more complex and less specialized language mechanisms. So, we found that AspectJ is suboptimal in the modularization of roles, which can be certainly credited to its general focus on all kinds of crosscutting concerns. In contrast, collaboration languages such as Classbox/J provide explicit means for the modularization of roles but are suboptimal with regard to other kinds of crosscutting concerns [33].

Furthermore, a significant volume of prior work in the areas of programming languages [20, 24, 25, 22, 26, 27, 28], generative programming [34, 12, 35, 36, 37, 38, 39], and software design [16, 40, 41] has shown that programs can be created *solely* by composing collaborations without advanced mechanisms, as provided by AspectJ. The statistics that we report on AspectJ usage – that 98 % of a program are collaborations (i.e., 86 % for the base program, *which is itself a collaboration*, plus the 12 % for basic AspectJ mechanisms) are consistent with this observation. We document these and other findings in this paper. We begin with a classification of crosscutting concerns.

2 Classification of Crosscuts

In the literature crosscutting concerns (a.k.a. *crosscuts*) have been classified along three dimensions (homogeneous/heterogeneous) [42], (static/dynamic) [18], and (basic/advanced) [33]. We use an example to illustrate them all.

2.1 An Example

Consider a program that implements a graph data structure (Fig. 1). It consists of a base program BasicGraph plus two features Color and Weight, which crosscut the implementation of BasicGraph. BasicGraph refers to the graph implementation without code implementing Weight and Color. The code of Color is underlined and blue and the code of Weight is *slanted and red*.

```
1  package BasicGraph;
2  class Graph {
3    Vector nv = new Vector(); Vector ev = new Vector();
4    Edge add(Node n, Node m) {
5      Edge e = new Edge(n, m);
6      nv.add(n); nv.add(m); ev.add(e);
7      e.weight = new Weight();
8      return e;
9    }
10   Edge add(Node n, Node m, Weight w) {
11     Edge e = new Edge(n, m);
12     nv.add(n); nv.add(m); ev.add(e);
13     e.weight = w; return e;
14   }
15   void print() {
16     for(Edge edge : ev) { edge.print(); }
17   }
18 }
19 class Edge {
20   Node a, b;
21   Color color = new Color();
22   Weight weight;
23   Edge(Node _a, Node _b) { a = _a; b = _b; }
24   void print() {
25     Color.setDisplayColor(color);
26     a.print(); b.print();
27     weight.print();
28   }
29 }
30 class Node {
31   int id = 0;
32   Color color = new Color();
33   void print() {
34     Color.setDisplayColor(color);
35     System.out.print(id);
36   }
37 }
38 class Color { static void setDisplayColor(Color c) { ... } }
39 class Weight { void print() { ... } }
```

Fig. 1. A simple graph implementation.

2.2 Classifying Crosscuts

Homogeneous and Heterogeneous Crosscuts

A *homogeneous crosscut* extends a program at multiple join points by adding the same piece of code at each join point. In our example, the Color feature is

homogeneous since it introduces the same piece of code to `Edge` (Lines 21, 25) and `Node` (Lines 32, 34).

A *heterogeneous crosscut* extends multiple join points each with a unique piece of code. The WEIGHT feature is heterogeneous since it extends `Graph` and `Edge` at different join points with different pieces of code (Lines 7, 10–14, 22, 27).

Static and Dynamic Crosscuts

A *static crosscut* extends the structure of a program statically, i.e., it adds new classes and interfaces and injects new fields, methods, and interfaces, etc. Overall, the features COLOR and WEIGHT introduce 2 classes (Lines 38, 39) and inject a method (Lines 10–14) to `Graph` and 3 fields (Lines 21, 22, 32) to `Edge` and `Node`.

A *dynamic crosscut* affects the runtime control flow of a program and can be understood and defined in terms of an event-based model [43, 18]: a dynamic crosscut executes additional code when predefined events occur during program execution. An example construct that implements a dynamic crosscut is an extension of a method, as we explain shortly. Overall, the features COLOR and WEIGHT extend 3 methods (Lines 7, 25, 27, 34).

Basic and Advanced Dynamic Crosscuts

The most primitive crosscut in AspectJ is a piece of advice that advises executions of a single method. *Object-oriented (OO)* languages and OO researchers express such advice as *method extensions* via subclassing (virtual classes or mixins), method overriding, and related mechanisms [44, 45, 46, 20, 28, 27, 23]. Dynamic crosscutting mechanisms in AspectJ transcend OO when they effect events other than singleton method executions (e.g., [47, 48]). Hence, we distinguish two classes of dynamic crosscuts, *basic dynamic crosscuts* and *advanced dynamic crosscuts*. Basic dynamic crosscuts:

1. affect executions of a single method,
2. access only runtime variables that are related to a method execution, i.e., arguments, result value, and enclosing object instance, and
3. affect a program control flow unconditionally.

All other dynamic crosscuts are advanced. A rule of thumb is that the join points of basic dynamic crosscuts can be determined statically; the join points of advanced dynamic crosscuts are determined at runtime. With AspectJ, an advanced dynamic crosscut is implemented by *advanced advice* and a basic dynamic crosscut by *basic advice*. This distinction helps identify which pieces of advice make use of advanced AspectJ mechanisms and which pieces merely implement OO method extensions.

3 Code Metrics

To see how programmers use AspectJ, we define five metrics that distinguish the use of aspects in terms of the classifications discussed in the last section. For

each metric, we count the number of *lines of code (LOC)* for different categories and determine the fraction of the program's source for each category.

While using LOC (eliminating blank and comment lines) as a metric might be controversial (e.g., how are 'if' statements counted?), our yielded statistics would be no different than, say, using a metric that counts statements. At the end, we compare just fractions of a program's code base associated with our categories. The essential results would remain valid.

Classes, Interfaces, and Aspects (CIA)

With the CIA metric we measure the fraction of classes, interfaces, and aspects of a program. It tells us whether aspects implement a significant or insignificant part of the code base (as opposed to classes and interfaces).

We simply traverse all source files included in a given AspectJ project and count the LOC of aspects, classes, and interfaces. Upfront we eliminate blank lines and comments.

Heterogeneous and Homogeneous Crosscuts (HHC)

The HHC metric explores to what extent aspects implement heterogeneous and homogeneous crosscuts. Specifically, we determine the fractions of the LOC associated with advice and inter-type declarations (introductions) that are heterogeneous and homogeneous. The HHC metric tells us whether the implemented aspects take advantage of the wildcard and pattern-matching mechanisms of AspectJ or merely emulate OO concepts.

We analyze each piece of advice and inter-type declaration: if its number of join points is greater than one it is a homogeneous crosscut; otherwise it is heterogeneous.

Code Replication Reduction (CRR)

Homogeneous advice and homogeneous inter-type declarations are useful for reducing code replication in a program. Imagine an aspect that advises 100 join points and executes at each join point 10 lines of code encapsulated in one piece of advice. This aspect would reduce the code base by approximately 990 lines of code. In order to quantify this benefit, the CRR metric counts the LOC that could be reduced in an aspect-oriented version compared to an object-oriented equivalent.[2]

We multiply the number of LOC of each homogeneous advice and inter-type declaration with the number of join points it affects (minus one).

[2] We do not consider the fact that tangled code in the analyzed programs could be refactored first using the 'extract method' refactoring before using aspects, thus, decreasing the CRR. For a better comparability, we analyze the programs as they are.

Static and Dynamic Crosscuts (SDC)

The SDC metric determines the code fraction associated with static and dynamic crosscuts. That is, it counts the LOC of inter-type declarations (static crosscutting) and pieces of advice (dynamic crosscutting). Note that heterogeneous and homogeneous crosscuts can be either static or dynamic. The SDC metric tells us to what extend aspects crosscut the dynamic computation of a program, which cannot be expressed well in OO languages, or the static structure of a program, which is also supported by advanced OO mechanisms such as mixins or virtual classes.

In AspectJ, we calculate the fraction of static and dynamic crosscuts by counting the LOC associated with inter-type declarations and pieces of advice and comparing them with the overall code base.

Basic and Advanced Dynamic Crosscuts (BAC)

The BAC metric determines the LOC associated with pieces of basic and advanced advice. The BAC metric tells us to what extent the aspects of a program take advantage of the advanced capabilities of AspectJ for dynamic crosscutting. Basic advice are method extensions, which can be expressed in OO languages.

In AspectJ, we consider a piece of advice to be advanced if its pointcut involves more than simply a combination of `execution` (or `call`[3]), `target`, and `args`.[4]

Tool Support

We have developed the *AJStats*[5] tool to calculate general statistics such as the number of LOC of classes, aspects, advice, inter-type declarations, etc. To identify homogeneous crosscuts and the number of affected join points we have used the *AJDTStats*[6] tool [10]. We are not aware of a tool that identifies advanced advice. In order to do so, we had to examine the code by hand.

4 Case Studies

We analyzed a diverse selection of AspectJ programs (see Table 1). The first 7 are small programs (< 20 KLOC); the last 4 are larger (≥ 20 KLOC). In our tables and figures, the programs are listed from smallest (Tetris) to largest (Abacus). We also indicate in Table 1 if the program was developed from scratch (Type S), or if it was an AOP refactoring of an existing application (Type R).

[3] Although the semantics of `call` is to advise the client side invocations of a method, it can be implemented as method extension – provided that *all* calls to the target method are advised.

[4] `execution` can be combined with `this`, `within`, and `withincode`.

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

[6] `http://wwwiti.cs.uni-magdeburg.de/iti_db/ajdtstats/`

The percentages that we report are averaged over the individual percentages of the eleven programs and rounded to the nearest integer, unless a fraction of a percent is reported. We use $a \pm s$ to mean average a with standard deviation s. So $14 \pm 10\,\%$ means an average of $14\,\%$ was observed with a standard deviation of 10. In certain situations, we will consider only a subset of the eleven programs, e.g., large-sized programs only, in order to explore the specific properties of an individual program or subset of programs.

Note that we do not consider development aspects, i.e., aspects that had been used during program development and removed before deployment. The reason is that the aspects are typically not available or even not existing anymore. Our statistics and results should be interpreted with this fact in mind. A comparison of development aspects and aspects actually deployed is a topic of further research.

4.1 Statistics and Interpretation

The raw data that our statistics are based on can be requested from the authors.

CIA Metric

Figure 2 shows that aspect code occupies on average $14 \pm 10\,\%$ of a program's code base; the bulk are classes and interfaces. Aspects account for more of the code base in small programs ($20 \pm 6\,\%$) than in larger programs ($3 \pm 4\,\%$).

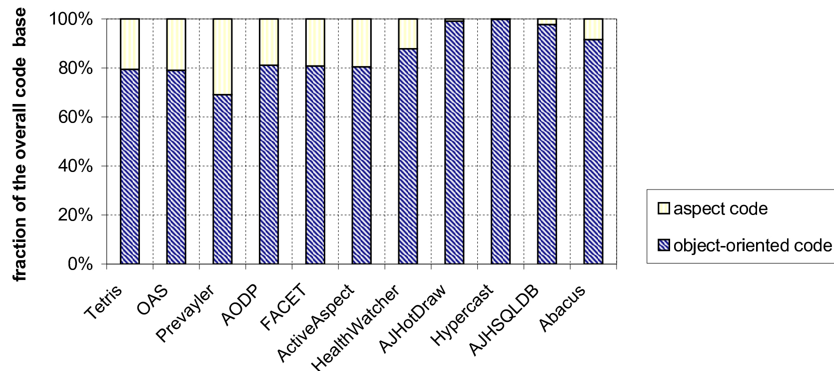

Fig. 2. Fractions of aspect code and object-oriented code of the overall code base.

HHC Metric

Figure 3 reveals the fractions of homogeneous and heterogeneous crosscuts. We found $1 \pm 1\,\%$ of the code base implements homogeneous advice and homogeneous inter-type declarations. In contrast, heterogeneous advice and heterogeneous inter-type declarations occupy a larger fraction $7 \pm 6\,\%$. The remaining

Table 1. Overview of the AspectJ programs analyzed.

Name	**LOC**	**Source**	**Description**	**Type**[l]
Tetris	1,030	Blekinge Inst. of Technology[a]	Implementation of the popular game	S
OAS	1,623	Lancaster University[b]	Online auction system	S
Prevayler	3,964	University of Toronto[c]	Main memory database system	R
AODP	3,995	University of British Columbia[d]	AspectJ implementation of 23 design patterns	R
FACET	6,364	Washington University[e]	CORBA event channel implementation	S
ActiveAspect	6,664	University of British Columbia[f]	Crosscutting structure presentation tool	S
HealthWatcher	6,949	Lancaster University[g]	Web-based information system	S
AJHotDraw	22,104	open source project[h]	2D Graphics Framework	R
Hypercast	67,260	University of Virginia[i]	Protocol for multicast overlay networks	R
AJHSQLDB	75,556	University of Passau[j]	SQL relational database engine	R
Abacus	129,897	University of Toronto[k]	CORBA Middleware Framework	R

[a] http://www.guzzzt.com/coding/aspecttetris.shtml

[b] The sources were kindly released by A. Rashid.

[c] http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem/

[d] http://www.cs.ubc.ca/~jan/AODPs/

[e] http://www.cs.wustl.edu/ doc/RandD/PCES/facet/

[f] The sources were kindly released by W. Coelho and G. Murphy.

[g] The sources were kindly released by A. Garcia.

[h] http://sourceforge.net/projects/ajhotdraw/

[i] The sources were kindly released by Y. Song and K. Sullivan.

[j] http://sourceforge.net/projects/ajhsqldb/

[k] The sources were kindly released by C. Zhang and H.-A. Jacobsen.

[l] Developed from scratch (S) or refactored an existing program (R)

6 % of the total 14 % of aspect code deals with local members in aspects. In general, homogeneous crosscuts are used infrequently.

Homogeneous advice and homogeneous inter-type declarations occupy a larger part in small programs (2 ± 1 %) than in larger programs (0.2 ± 0.3 %).

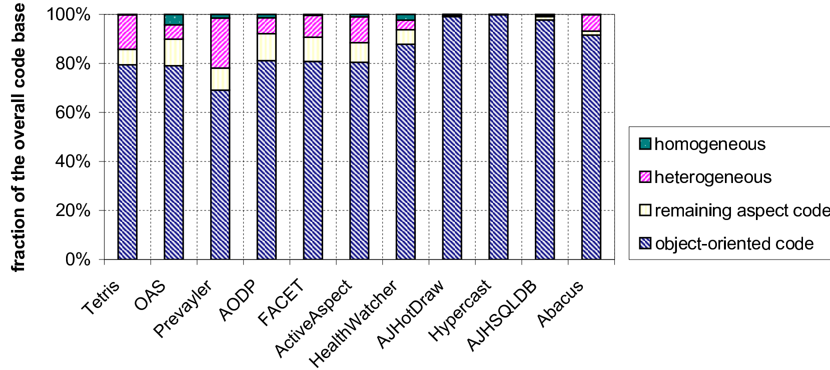

Fig. 3. Fractions of homogeneous and heterogeneous crosscuts of the overall code base.

CRR Metric

Figure 4 shows the different percentages of code reduction, i.e., cloned code that was eliminated by homogeneous advice and homogeneous inter-type declarations (6 ± 9 %). On average, the small programs achieve a slightly larger reduction (7 ± 8 %) than larger programs (6 ± 11 %). Notice, the smallest program (Tetris) had no reduction, while the second largest program (AJHSQLDB) had the highest 23 %.

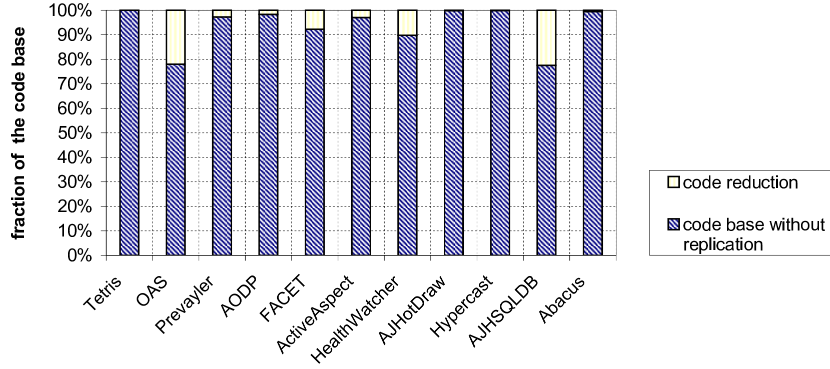

Fig. 4. Code reduction achieved by aspects.

SDC Metric

Figure 5 shows the fractions of inter-type declarations and pieces of advice. We found $3 \pm 3\,\%$ implements inter-type declarations and $5 \pm 5\,\%$ implements advice. The remaining $6\,\%$ of the total $14\,\%$ of aspect code deals with local members in aspects. On average, inter-type declarations and pieces of advice have been used to similar extents.

Since aspects have been used to a lesser extent in larger programs, also advice ($1 \pm 2\,\%$) and inter-type declarations ($1 \pm 2\,\%$) account for a smaller fraction of the code bases of large programs. In smaller programs advice ($7 \pm 5\,\%$) accounts for a slightly larger fraction that inter-type declarations ($4 \pm 4\,\%$).

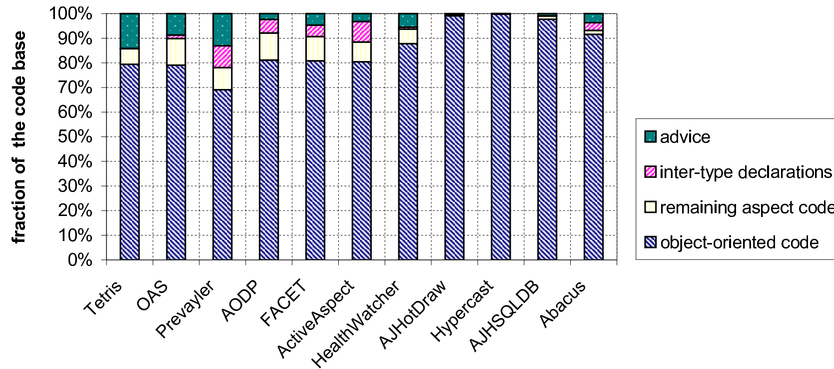

Fig. 5. Fractions of static and dynamic crosscuts of the overall code base.

BAC Metric

Figure 6 shows the fractions of basic and advanced advice. We found $1 \pm 1\,\%$ of the code base implements advanced advice. In contrast, basic advice occupies a larger fraction of $4 \pm 4\,\%$. The remaining $9\,\%$ of the $14\,\%$ that aspects occupy deals with inter-type declarations and local members in aspects.

As with the HCC metric, we observed that advanced advice is used more frequently in small programs ($1 \pm 1\,\%$) than in larger programs ($0.2 \pm 0.3\,\%$).

4.2 Discussion

Figure 7 depicts the fractions of the code base of the AspectJ programs that exploit advanced mechanisms (i.e., homogeneous advice and homogeneous introductions, and advanced advice) and basic mechanisms (heterogeneous basic advice and heterogeneous introductions). On average, only a minor fraction of $2 \pm 2\,\%$ of the analyzed code exploits the advanced capabilities of AspectJ; $12 \pm 9\,\%$ implements basic aspects, and the remaining $86\,\%$ is OO code.

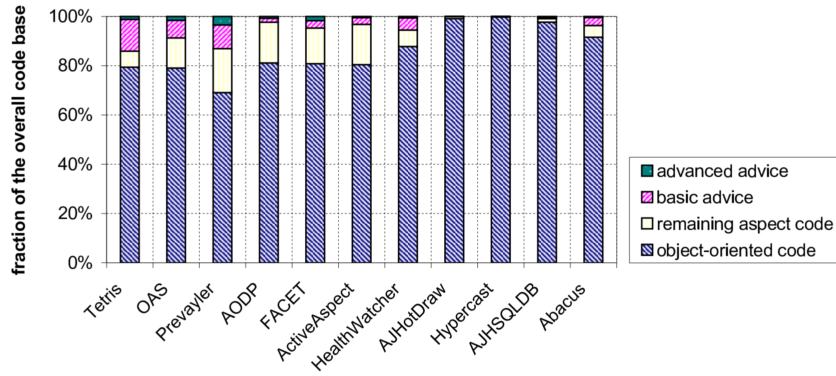

Fig. 6. Fractions of basic and advance advice of the overall code base.

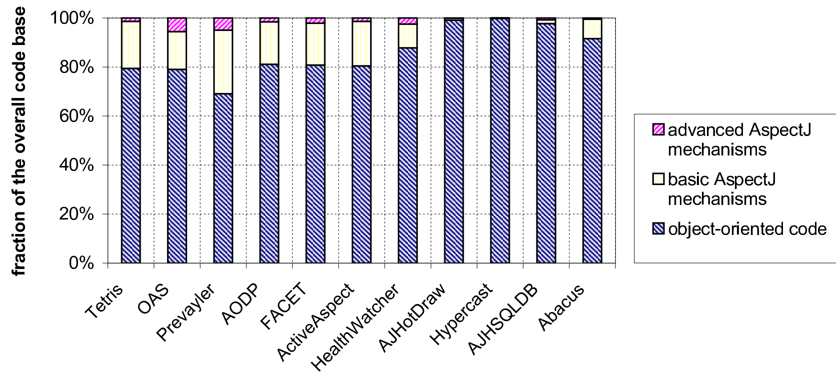

Fig. 7. Fractions of basic AspectJ and advanced AspectJ mechanisms of the overall code base.

Our use of percentages of the overall code base may not tell the whole story. An alternative is to examine the use of advanced AspectJ mechanisms *within* the aspect code of a program, which could be argued as the fraction of the program's base that has been 'factored-out'. This too does not tell the whole story, as entire classes and interfaces may be (a potentially large) part of a concern implementation that is ignored. Thus, percentages based only on aspect code would provide an overestimation. Nevertheless, we show in Figure 8 that even with this overestimation, only $15 \pm 9\,\%$ of the aspect code accounts for advanced AspectJ mechanisms; the rest is basic AspectJ mechanisms.

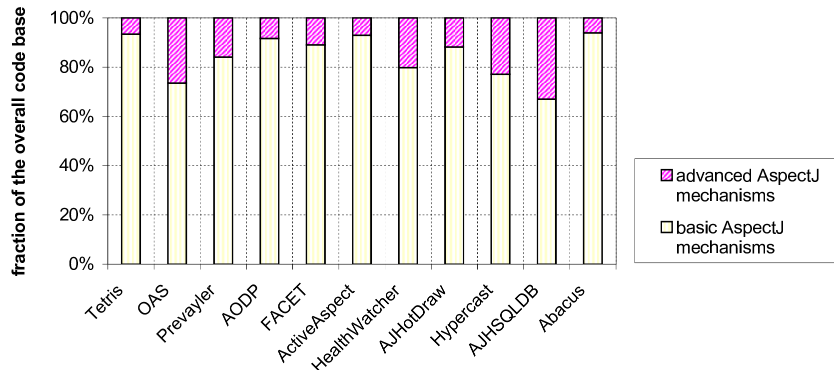

Fig. 8. Fractions of AspectJ and advanced AspectJ mechanisms of a program's aspect code.

The $6 \pm 9\,\%$ code reduction that we have observed is in line with prior work on clone detection that conjectures that 5–15 % of large software projects are clones, i.e., replicated code fragments [49]. So there might be an untapped potential (further 9 %=15 %-6 %) of AspectJ to reduce code replication further because not all clones have been discovered. Also, some clones are not exact matches [50], so 5–15 % may be an upper bound that aspects can not reach.

5 Collaborations

In this section, we argue that the reason that basic AspectJ mechanisms (introductions and method extensions) account for notable 85 % of all aspect code is that crosscutting concerns typically manifest themselves in the form of *collaborations* (a.k.a., *role-based designs*). (As we mentioned earlier, 98 % of a program are collaborations – the 86 % for the base program, *which is itself a collaboration*, plus the 12 % for basic AspectJ mechanisms.) Collaborations are fundamental forms of crosscutting concerns that have appeared in many forms over the last twenty years (e.g., *object-oriented collaborations* [16], *role components* [17], *object-oriented frameworks* [41], *design patterns* [51], *layers* [21], *subjects* [52],

slices [53], *family classes* [54], *higher-order hierarchies* [23], *units* [46], *refinements* [20]). In recent work, collaborations have been identified to be a class of crosscutting concerns that can be implemented by AOP languages such as AspectJ [30, 29]. The next section outlines the basic ideas.

5.1 Illustrating Collaborations

A *collaboration* of classes is a role-based protocol that defines the inter-class communication necessary to complete a predefined task [16, 17, 18, 19, 20, 21, 22, 23]. Classes play different roles in different collaborations [17]. A *role* encapsulates the protocol that a class provides when a corresponding collaboration with other classes is established.

When a collaboration is added to a program, new classes and roles are introduced. A role adds new members (e.g., fields, methods) to a class and extends existing methods. Hence, a collaboration crosscuts several places in a base program [21]. Furthermore, a collaboration is predominantly characterized by *heterogeneous basic advice* and *heterogeneous introductions*. This is not surprising since typically a communication between objects of different classes is asymmetrically. That is, in order to let the objects perform different tasks in a collaboration they have to be extended in different ways. Note that a base program is itself a collaboration: roles are pre-bound to existing classes.

The BasicGraph program and the Weight feature of Section 2 are collaborations. Weight extends BasicGraph at several points by different pieces of code and it extends methods only. Figure 9 depicts a modular implementation of Weight based on *classboxes* written in *Classbox/J*, a collaboration language on top of Java [27]. The Weight collaboration extends the base program BasicGraph. It introduces the class `Weight` (Line 16) and extends the imported classes `Graph` (Lines 1–11) and `Edge` (Lines 12–15) by refinements, which are declared by the keyword `refine`. These refinements introduce new fields and methods and extend existing methods. Within a method extension the keyword `original` refers to the method that is being refined (Lines 5, 16).

Refinements can be implemented with several mechanisms, e.g., by using *mixins* [17,21,46,45], *virtual classes* [44,22], *nested inheritance* [28], *refinements* [20], and advice and inter-type declarations [30, 29]. All refinements and classes of a collaboration are encapsulated inside a single module.

5.2 Advanced Aspects

Not all concerns can be compactly expressed just by simple introductions and method extensions (e.g., basic AspectJ mechanisms). Sometimes there is a redundancy in the introductions or in the method extensions implementing a concern, which makes the crosscut being addressed homogeneous. Consider the Color feature of the BasicGraph program of Section 2. Its representation in Classbox/J is shown in Figure 10.

```
1  package Weight;
2  import BasicGraph.Graph; import BasicGraph.Edge;
3  refine class Graph {
4    Edge add(Node n, Node m) {
5      Edge e = original.add(n, m);
6      e.weight = new Weight(); return e;
7    }
8    Edge add(Node n, Node m, Weight w) {
9      Edge e = new Edge(n, m);
10     nv.add(n); nv.add(m); ev.add(e);
11     e.weight = w; return e;
12   }
13 }
14 refine class Edge {
15   Weight weight;
16   void print() { original.print(); weight.print(); }
17 }
18 class Weight { void print() { ... } }
```

Fig. 9. Implementing WEIGHT as collaboration.

```
1  package Color;
2  import BasicGraph.Node; import BasicGraph.Edge;
3  interface Colored { ... }
4  class Color { ... }
5  refine class Node implements Colored {
6    Color color = new Color();
7    void print() {
8      Color.setDisplayColor(color);
9      original.print();
10   }
11 }
12 refine class Edge implements Colored {
13   Color color = new Color();
14   void print() {
15     Color.setDisplayColor(color);
16     original.print();
17   }
18 }
```

Fig. 10. Implementing COLOR as collaboration.

```
1 aspect AddColor {
2   interface Colored { ... }
3   declare parents: (Node || Edge) implements Colored;
4   Color Node.color = new Color();
5   Color Edge.color = new Color();
6   before(Colored c) : execution(void print()) &&
7     this(c) { Color.setDisplayColor(c.color); }
8   static class Color { ... }
9 }
```

Fig. 11. Implementing COLOR as aspect.

Homogeneous Crosscuts

Note that the `print` methods of both `Node` and `Edge` are extended identically (Figure 10; Lines 7–10 and 14–17), and also an identical field `color` is added to each class (Lines 6 and 13). Color could be expressed as an advanced aspect taking advantage of the wildcard and pattern-matching mechanisms of AspectJ (see Figure 11). The aspect `AddColor` defines an interface `Colored` (Line 2) and it declares that `Node` and `Edge` implement that interface (Line 3); it introduces a field `color` (Lines 4–5), and it advises the execution of the method `print` of `Edge` and `Node` (Lines 6–7); the class `Color` is introduced as static inner class (Line 8).

Advanced Dynamic Crosscuts

Occasionally concerns can use dynamic crosscuts, such as a collaboration applying a role to a class that is dependent on the program control flow, which is an advanced dynamic crosscut. For example, when implementing a new feature of our graph example that modifies the routine of printing graph structures (PrintHeader) we can take advantage of the advanced mechanisms of AspectJ for dynamic crosscutting. Suppose the `print` methods of the participants 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 a conditional (Fig. 12).

```
1 aspect PrintHeaderAspect {
2   before() : execution(void print())&&
3    !cflowbelow(execution(void print())) { header(); }
4   void header() { System.out.print("header: "); }
5 }
```

Fig. 12. Implementing PrintHeader as an Aspect.

Figure 13 depicts an excerpt and approximation of the behavior of PrintHeader implemented using Classbox/J; the complete implementation would be more complex. Omitting advanced AspectJ mechanisms results in a workaround (underlined and green) for tracing the control flow and executing the actual extension conditionally (Lines 6, 7).

While AspectJ code can be more compact, it is debatable whether the result is easier to understand and maintain [55, 56], especially in situations where few join points are affected (e.g., compare Figure 12 with Figure 13) [10]. Regardless, we found that such dynamic crosscuts occur rarely (1 %).

```
 1 package PrintHeader;
 2 import BasicGraph.Node;
 3 refine class Node {
 4   static int count = 0;
 5   void print() {
 6     if(count == 0) printHeader();
 7     count++; original.print(); count--;
 8   }
 9   void printHeader() { /* ... */ }
10 }
```

Fig. 13. Implementing PrintHeader via refinement.

5.3 Corroborating Evidence

A review of the eleven AspectJ programs reveals that aspects often implement collaborations: typically, an aspect extends multiple objects by multiple pieces of advice and inter-type declarations, and it extends the objects in different ways to let them interact with each other and with other objects. In other words, the extensions an aspect typically makes to a base program are heterogeneous. Furthermore, we observed that aspects seldom used advanced AspectJ mechanisms but mainly extended the static program structure by inter-type declarations and the execution of methods by pieces of basic advice. This is similar to the nature of collaborations as used in collaboration-based designs [16, 17, 40, 21].

Other researchers that examined some of the eleven AspectJ programs came to the same conclusion. For example, Liu et al. noted that the aspect refactoring of Prevalyer corresponds closely to a collaboration version [12]. Also, Xin et al. observed that the collaboration version of FACET is close to the aspect-oriented version [34]. Design patterns are classic collaborations, and by definition so too are their AODP implementations (Observer, Command, Visitor, Strategy, etc.).

Where there could be any doubt, we contacted the developers to ask if they had collaborations in mind, but we did not reveal our statistical findings. In our first correspondences with the authors we asked:

> During the implementation of the aspects, did you have collaborations between several objects in the mind. That is, did you think in terms of: "for this functionality I have to extend class A, B, C in order to let them collaborate and to implement a program feature". In the AOP community this is sometimes called a multi-object protocol.

In follow-up correspondences we clarified further (in this or a similar wording):

> The alternatives to a collaboration would be either the extension of many classes with the same code (homogeneous extension), which also means that these classes do not collaborate (or interact, if you want), or the extension of the dynamic control flow (advanced dynamic extension), in the sense that you thought mainly about the control flow graph and the crosscutting in this graph (e.g. using cflow).

We summarize below the authors' responses:

- The developer of Abacus (C. Zhang) confirmed that his aspect composition was driven by superimposition of views (roles). He extended existing classes by using AspectJ style mixins and implemented the methods declared by interfaces.
- The developer of FACET (R. Pratap) answered that he definitely had to think of collaborations while implementing features in FACET.
- The developer of ActiveAspect (W. Coehlo) said that his aspects were written with collaborations between several objects in mind. Generally, he did not think in terms of crosscutting dynamic computation.
- The developer of AJHotDraw (M. Marin) explained that there are a number of refactored concerns in AJHotDraw whose implementations consist of classes with multiple roles and various collaboration protocols.
- The developer of Prevayler (I. Godil) confirmed that there are aspects that implement collaborations, which is in line with [12].
- K. Sullivan reported that his work on Hypercast was not intended (and did not) explore the use of aspects to implement collaborations, but rather to take some easy/classical applications of aspects, such as logging, and to use them to evaluate the notion of 'obliviousness'. Sullivan noted that he had looked at the issue of collaborations and aspects in *classpects* [57].
- The developer of HealthWatcher (S. Soares) was unfamiliar with the concept of collaborations, and was unable to say whether collaborations were used or not.

We did not receive responses from the Tetris, OAS, and AJHSQLDB developers. The majority of responses indicate that the developers noticed collaborations in their work – although not all were aware of the concept or the term.

5.4 Validity Discussions

The statistics of our study should be interpreted with the fact in mind that we limited our attention to AspectJ. That is, we could not consider development aspects and other kinds of aspects such as used in container-based AOP.[7] Furthermore, there are three validity issues to our study: *construct*, *internal*, and *external*.

Construct

The distinctions between different concern classifications – homogeneous/heterogeneous, static/dynamic, and basic/advanced – are both fundamental and well-recognized in the literature [42, 2, 18, 15]. Our use of LOC (eliminating blank and comment lines) as a metric yielded statistics that would be different than, say, a metric that counts statements. Although any metric has problems (e.g., how are 'if' statements counted?), the essential result of our paper, namely that advanced and homogeneous crosscuts are used infrequently in our case studies, would remain valid.

[7] Examples of container-based AOP are JBossAOP (`http://jboss.com/products/aop`) and SpringAOP (`http://www.springframework.org/`).

Internal

There are always problems drawing significant conclusions from a small sample size. This is a problem with any such study with AspectJ: there are few published, non-trivial programs using AspectJ in open literature. If other programs exist, we are not aware of them. Moreover, the eleven programs have been mainly developed for academic purposes. A recent discussion in the AOSD.NET mailing list[8] reveals that there are only a few published industrial projects that use AspectJ, none of them available for download.

Furthermore, we were unable to contact authors of three of the programs to determine whether they had collaborations in mind. Even assuming negative responses, a majority of the authors indicated that they were aware of collaborations in their work.

Another possible issue is that because the term 'collaboration' is overloaded, there could have been misunderstandings between the developers and us. To minimize this, we defined collaborations in our correspondences with authors (cf. Sec. 5.3).

External

We believe our results are representative of AspectJ usage. We explicitly omitted our own internal case studies, whose statistics are nevertheless consistent with those we reported [10,11,12,13]. We cite in related work additional corroborating evidence. And finally, we and others have been building systems for years by composing collaborations without using advanced AspectJ mechanisms; AspectJ might have helped in the cases where advanced aspects could be used (e.g., to reduce code replication). We are aware of only a few such cases in our own code base.

The significance of collaborations in software development is addressed in the next section.

6 Perspective

In the 2 % of the code base where advanced mechanisms are used, let us assume that the use of AspectJ is appropriate. For the remaining 98 %, we need a language to express collaborations. There are two options: (1) use AspectJ or (2) use a language that is specifically designed to express collaborations, such as *Jak* [20], *FeatureC++* [24], *ContextL* [25], *Scala* [22], *Jiazzi* [26], *Classbox/J* [27], *Jx* [28]. Although the support of collaborations is implemented very differently in the latter group of languages, the key concepts with regard to collaborations are very similar (see Section 7). Even some aspect-oriented languages were inspired by work on collaboration-based designs [18, 19, 58]. Here is a summary of our findings and those of other researchers on this topic.

[8] `http://aosd.net/pipermail/discuss_aosd.net/2007-May/002163.html`

First, with a collaboration language, each role of a collaboration encapsulates a set of changes made to a single class and has the name of the class to which it is bound (cf. Fig. 10 for a Classbox/J example). In this way a programmer is able to infer the static OO structure of a resulting program as there is a one-to-one mapping between the structural elements of the base program and the elements of the collaboration; a base program and collaborations are merged recursively by name and type [59, 20, 21, 22].

In AspectJ and related languages [60, 57] collaborations are implemented differently: the members of a role are implemented by pieces of advice and inter-type declarations, which are not required to be listed consecutively in an aspect, or even placed in a single aspect. Thus, the contents of a collaboration can be scattered over and interspersed within many aspect files, which leads to a suboptimal role modularity. Imposing a class-or-role-based organization is left to the discipline of the programmer [31, 29, 30, 32].

Second, simple concepts like method extensions have a compact and easier-to-understand representation in collaboration-based languages. Fig. 14 shows a typical extension expressed in both Classbox/J and AspectJ. We note that some AOP languages even refrain from offering a pointcut-advice-mechanisms for this reason [19, 58, 61].

```
1 public void delete(Transaction txn, DbEntry key) {
2   original(txn, key);
3   Tracer.trace(Level.FINE, "Db.delete", this, txn, key);
4 }
```

```
1 pointcut traceDel(Database db, Transaction txn, DbEntry key) :
2   execution(void Database.delete(Transaction, DbEntry)) &&
3   args(txn, key) && within(Database) && this(db);
4 after(Database db, Transaction txn, DbEntry key): traceDel(db, txn, key) {
5   Tracer.trace(Level.FINE, "Db.delete", db, txn, key);
6 }
```

Fig. 14. A method extension in Classbox/J (top) and AspectJ (bottom); taken from [10]. (`within` is necessary to not affect subclasses of `Database`.)

Third, collaborations usually add new classes to a program. Collaboration languages offer mechanisms to modularize the changes to existing classes as well as the new classes a collaboration injects into a program. Here, AspectJ provides no infrastructure to group several aspects and classes together to form a module [62]. Stated differently, AspectJ does not support the modularization of typical collaborations [33]. Programmers have to manipulate the compiler's build path to include or exclude certain aspect files [3], and must implement a non-trivial workaround for modularization and composition of collaborations [63]. Alternatively, programmers can merge aspects and classes into packages. While this groups aspects and classes belonging to a collaboration, it is not possible to select and compose collaborations without difficulty [62].

Finally, researchers have argued that not expressing a collaboration in terms of object orientation (i.e., roles implemented as modules) decreases program comprehension [64,18]. This is because programmers cannot recognize the OO structure of the woven program [64, 55, 56]. However, for our BasicGraph example, it does not matter whether we use Classbox/J or AspectJ because BasicGraph and its features are too *small*. But, as collaborations scale, i.e.,

1. as the number of roles per collaboration increases,
2. as the number of collaborations increases, and
3. as the complexity of roles increases,

the difficulty of comprehending the implemented collaborations grows (with collaboration languages and AspectJ-like languages).

Adressing the above issues, some researchers proposed to use AspectJ in a more collaboration-oriented way: Hanenberg et al. [30] and Kendall [29] have suggested that each role be implemented by a distinct aspect, thus establishing a one-to-one mapping between the structural elements of the base program and the elements of the collaboration (provided reasonable naming conventions are used). Nevertheless, this mapping is not enforced by the language, and is left to the discipline of the programmer.

Ultimately, we believe the most fruitful approach will be to combine collaboration languages with AspectJ-like mechanisms: use a simple language to express the simple program extensions of collaborations. Only when more sophisticated program modifications are needed, use the advanced mechanisms in AspectJ. This approach offers the best of both worlds. Previous work in support of this position is [53, 54, 19, 58, 33].

7 Related Work

Collaborations

Although the concept of collaborations predates AOP by quite some time [16, 39, 17], mainstream programming languages have been very slow to support them. AspectJ has filled the vacuum [65]. The weak support of collaborations in aspect and non-aspect mainstream languages has contributed to a general confusion regarding collaborations and their relationship to, importance to, and frequency in crosscutting concerns. Nevertheless, several studies demonstrate that collaboration languages suffice in implementing large applications [39, 36, 38, 37, 20, 35].

There are many languages that incorporate the concept of roles and collaborations in its language design, although sometimes called differently, e.g., *Jak* [20], *FeatureC++* [24], *ContextL* [25], *Scala* [22], *Jiazzi* [26], *Classbox/J* [27], *Jx* [28]. While the capabilities and mechanisms of these languages differ considerably, their aim at aggregating classes that collaborate in modules and extending or overriding existing classes is very similar. For example, with Jx's nested inheritance we can introduce and refine existing classes in almost the same way as in with classboxes in Classbox/J, abstract types in Scala, components in Jiazzi, or layers in ContextL.

Several approaches even combine collaboration-based design and advanced AspectJ mechanisms to benefit from both worlds, e.g., *Relationship Aspects* [66], *CaesarJ* [54], *FeatureC++* [24], *Aspectual Collaborations* [19], and *Object Teams* [58], but their use has not been extensive

Representative AOP Case Studies

Colyer and Clement refactored an application server using AspectJ (3 homogeneous and 1 heterogeneous crosscuts) [2]. 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 did not explore) a strong relationship of their aspects to collaborations.

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200–400 KLOC) [4]. They factored 4 crosscutting concerns into AspectC aspects; inherent properties of concerns were not explained in detail.

Lohmann et al. examine the applicability of AspectC++ to embedded systems (2 homogeneous and 1 heterogeneous crosscuts) [5]. Tesanovic et al. implemented 10 AspectC++ aspects for quality-of-service management in database systems [67]; the aspects implement predominantly collaborations.

Lopez-Herrejon et al. analyzed an AspectJ implementation of the AHEAD Tool Suite [13]. They found 1 % of the code base associated with advice; the rest consists of introductions. They did not consider advanced advice.

Greenwood et al. conducted a quantitative case study exploring the effects of an aspectual decomposition on design stability [8]. They implemented 8 crosscuts in the HealthWatcher system with AspectJ, but they did not say whether these are basic or advanced. Although they used AspectJ and CaesarJ they did not explore the relationship of collaborations and AOP concepts.

Classification Schemes

Alternative classification schemes of aspects and the crosscutting concerns they implement have been proposed in the literature. For example, spectative, regulative, and invasive aspects [68], harmless and harmful advice [69], or observers and assistants [70], that all classify aspects based on the invasiveness of their effects on the base program. Our distinction between heterogeneous and homogeneous as well as static, basic and advanced dynamic is orthogonal to these previous proposals. Our classification has been shown useful to compare two different lines of research in programming languages.

AOP Metrics

Zhang and Jacobson use a set of object-oriented metrics to quantify the program complexity reduction when applying AOP to middleware systems [3]. They show that refactoring a middleware system (23 KLOC code base) into aspects reduces

the complexity and leads to a code reduction of 2–3 %, which is in line with our results.

Garcia et al. analyzed several aspect-oriented programs (4–7 KLOC code base) and their OO counterparts [6, 71]. They observe that the AOP variants have fewer lines of code than their OO equivalents (12 % code reduction).

Zhao and Xu propose several metrics for aspect cohesion based on aspect dependency graphs [72]. Gelinas et al. discuss previous work on cohesion metrics and propose an approach based on dependencies between aspect members [73].

All of the above proposals and case studies take neither the structure of crosscutting concerns nor the difference between collaborations and other concerns into account.

Lopez-Herrejon et al. propose a set of code metrics for analyzing the crosscutting structure of aspect-based product lines [74]. They do not consider elementary crosscuts but analyze crosscutting properties of entire subsystems (features), which may have a substantial size. Thus, the crosscutting structure of a feature can be homogeneous, heterogeneous, or any value in between the spectrum of both. They do not distinguish between basic and advanced dynamic crosscuts.

8 Conclusions

We analyzed eleven AspectJ programs of different sizes and complexity. We found that on average 2 % of the code base is associated with advanced AspectJ mechanisms; 12 % is associated with basic AspectJ mechanisms; and 86 % is object-oriented. We presented evidence from many sources that a crosscutting concept whose characteristics (i.e., one that is predominantly introductions and method extensions) matches this result is collaborations. This is in line with prior results in programming languages [20, 24, 25, 22, 26, 27, 28], generative programming [34, 12, 35, 36, 37, 38, 39], and software design [16, 40, 41]. It is also in line with the experience of others who have distinguished aspects and collaborations [18,19,58,11,74,13,12], and is consistent with our own internal case studies on AspectJ [10, 11, 12, 13]. Furthermore, for a number of programs, prior work confirmed that aspects implemented collaborations. Where it was possible, we were able to confirm that a majority of authors of the AspectJ programs had notions of collaborations in mind.

As noted, it is not the case that all aspects used in these programs could be readily implemented or conceived as collaborations. Using aspects to express homogeneous crosscuts to reduce code redundancy, to express program-wide invariants [75], and contract enforcement [76] are examples. While such examples do indeed occur, they presently occupy a small fraction of aspect usage. Due to the arrangement of our analysis, other kinds of aspects such as development or container-based aspects have not been considered.

Modularizing concerns will continue to be a source of inspiration for advances in software engineering. The essential message of this paper is that languages and tools that focus on collaborations, with judicious use of mechanisms for

advanced dynamic and homogeneous crosscuts, will be more useful in future programming environments than using collaboration languages and AspectJ-like languages alone.

Acknowledgments

We thank K. Fisler, M. Grechanik, C. Kästner, P. Kim, S. Krishnamurthi, C. Lengauer, and D. Perry for their helpful comments on earlier drafts of this paper. Batory's research is sponsored by NSF's Science of Design Project #CCF-0438786.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. Europ. Conf. Object-Oriented Programming. (1997)
2. Colyer, A., Clement, A.: Large-Scale AOSD for Middleware. In: Proc. Int'l. Conf. Aspect-Oriented Software Development. (2004)
3. Zhang, C., Jacobsen, H.A.: Resolving Feature Convolution in Middleware Systems. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2004)
4. Coady, Y., Kiczales, G.: Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In: Proc. Int'l. Conf. Aspect-Oriented Software Development. (2003)
5. Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., Schröder-Preikschat, W.: A Quantitative Analysis of Aspects in the eCos Kernel. In: Proc. Int'l. EuroSys Conf. (2006)
6. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., v. Staa, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: Proc. Int'l. Conf. Aspect-Oriented Software Development. (2005)
7. Garcia, A., Sant'Anna, C., Chavez, C., Silva, V., v. Staa, A., Lucena, C.: Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications. (2003)
8. Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Santa-Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Proc. Europ. Conf. Object-Oriented Programming. (2007)
9. Störzer, M.: Impact Analysis for AspectJ – A Critical Analysis and Tool-based Approach to AOP. PhD thesis, School of Computer Science and Mathematics, University of Passau (2007)
10. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: Proc. Int'l. Software Product Line Conf. (2007)
11. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. In: Proc. Int'l. Conf. Generative and Component-Based Software Engineering. (2006)
12. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proc. Int'l. Conf. Software Engineering. (2006)
13. Lopez-Herrejon, R., Batory, D.: 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)

14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proc. Europ. Conf. Object-Oriented Programming, Springer (2001)
15. Apel, S.: The Role of Features and Aspects in Software Development. PhD thesis, School of Computer Science, University of Magdeburg (2007)
16. Reenskaug, T., Andersen, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., Stenslet, P.: OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. J. Object-Oriented Programming **5**(6) (1992)
17. VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1996)
18. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. Int'l. Symp. Foundations of Software Engineering. (2004)
19. Lieberherr, K.J., Lorenz, D., Ovlinger, J.: Aspectual Collaborations – Combining Modules and Aspects. Computer J. **46**(5) (2003)
20. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30**(6) (2004)
21. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Trans. Software Engineering and Methodology **11**(2) (2002)
22. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2005)
23. Ernst, E.: Higher-Order Hierarchies. In: Proc. Europ. Conf. Object-Oriented Programming. (2003)
24. Apel, S., Rosenmüller, M., Leich, T., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proc. Int'l. Conf. Generative and Component-Based Software Engineering. (2005)
25. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-Oriented Programming. J. Object Technology **7**(3) (2008)
26. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-Age Components for Old-Fashioned Java. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2001)
27. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2005)
28. Nystrom, N., Chong, S., Myers, A.C.: Scalable Extensibility via Nested Inheritance. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2004)
29. Kendall, E.A.: Role Model Designs and Implementations with Aspect-Oriented Programming. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1999)
30. Hanenberg, S., Unland, R.: Roles and Aspects: Similarities, Differences, and Synergetic Potential. In: Proc. Int'l. Conf. Object-Oriented Information Systems. (2002)
31. Pulvermüller, E., Speck, A., Rashid, A.: Implementing Collaboration-Based Designs Using Aspect-Oriented Programming. In: Proc. Int'l. Conf. Technology of Object-Oriented Languages and Systems. (2000)
32. Sihman, M., Katz, S.: Superimpositions and Aspect-Oriented Programming. The Computer Journal **46**(5) (2003)

33. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering **34**(2) (2008)
34. Xin, B., McDirmid, S., Eide, E., Hsieh, W.C.: A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, The University of Utah (2004)
35. Trujillo, S., Batory, D., Diaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proc. Int'l. Conf. Generative and Component-Based Software Engineering. (2006)
36. Batory, D., Thomas, J.: P2: A Lightweight DBMS Generator. J. Intell. Inf. Syst. **9**(2) (1997)
37. Batory, D., Johnson, C., MacDonald, B., v. Heeder, D.: Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. ACM Trans. Software Engineering and Methodology **11**(2) (2002)
38. Batory, D., Coglianese, L., Goodwin, M., Shafer, S.: Creating Reference Architectures: An Example from Avionics. In: Proc. Int'l. Symp. Software Reuse. (1995)
39. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Trans. Software Engineering and Methodology **1**(4) (1992)
40. VanHilst, M., Notkin, D.: Decoupling Change from Design. In: Proc. Int'l. Symp. Foundations of Software Engineering. (1996)
41. Johnson, R., Foote, B.: Designing Reusable Classes. J. Object-Oriented Programming **1**(2) (1988)
42. Colyer, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University (2004)
43. Wand, M., Kiczales, G., Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. ACM Trans. Programming Languages and Systems **26**(5) (2004)
44. Madsen, O.L., Moller-Pedersen, B.: Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1989)
45. Bracha, G., Cook, W.R.: Mixin-Based Inheritance. In: Proc. Europ. Conf. Object-Oriented Programming and Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1990)
46. Findler, R.B., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In: Proc. Int'l. Conf. Functional Programming. (1998)
47. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Proc. Europ. Conf. Object-Oriented Programming. (2005)
48. Masuhara, H., Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming. In: Proc. Asian Symp. Programming Languages and Systems. (2003)
49. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: Proc. Int'l. Conf. Software Maintenance. (1998)
50. Baker, B.S.: On Finding Duplication and Near-Duplication in Large Software Systems. In: Proc. Work. Conf. Reverse Engineering. (1995)
51. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
52. Harrison, W., Ossher, H.: Subject-Oriented Programming: A Critique of Pure Objects. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1993)

53. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int'l. Conf. Software Engineering. (1999)
54. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. Trans. Aspect-Oriented Software Development **1**(1) (2006)
55. Steimann, F.: The Paradoxical Success of Aspect-Oriented Programming. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (2006)
56. Alexander, R.: The Real Costs of Aspect-Oriented Programming. IEEE Software **20**(6) (2003)
57. Rajan, H., Sullivan, K.J.: Classpects: Unifying Aspect- and Object-Oriented Language Design. In: Proc. Int'l. Conf. Software Engineering. (2005)
58. Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Proc. Int'l. Net.ObjectDays Conf. (2002)
59. Ossher, H., Harrison, W.: Combination of Inheritance Hierarchies. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications. (1992)
60. Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: An AOP Extension for C++. Software Developer's Journal (2005)
61. Johansen, R., Sestoft, P., Spangenberg, S.: Zero-Overhead Composable Aspects for .NET. In: Proc. Lipari Summer School. (2008)
62. Lopez-Herrejon, R., Batory, D., Cook, W.R.: Evaluating Support for Features in Advanced Modularization Technologies. In: Proc. Europ. Conf. Object-Oriented Programming. (2005)
63. Hunleth, F., Cytron, R.: Footprint and Feature Management Using Aspect-Oriented Programming Techniques. SIGPLAN Not. **37**(7) (2002)
64. Steimann, F.: Domain Models are Aspect Free. In: Proc. Int'l. Conf. Model Driven Engineering Languages and Systems. (2005)
65. Masuhara, H., Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Proc. Europ. Conf. Object-Oriented Programming. (2003)
66. Pearce, D.J., Noble, J.: Relationship Aspects. In: Proc. Int'l. Conf. Aspect-Oriented Software Development. (2006)
67. Tesanovic, A., Amirijoo, M., Bjork, M., Hansson, J.: Empowering Configurable QoS Management in Real-Time Systems. In: Proc. Int'l. Conf. Aspect-Oriented Software Development. (2005)
68. Katz, S.: Aspect Categories and Classes of Temporal Properties. Trans. Aspect-Oriented Software Development **1**(1) (2006)
69. Dantas, D.S., Walker, D.: Harmless Advice. In: Proc. Int'l. Symp. Principles of Programming Languages. (2006)
70. Clifton, C., Leavens, G.: Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In: Proc. Int'l. Workshop Foundations of Aspect-Oriented Languages. (2002)
71. Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., v. Staa, A., Lucena, C.: Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In: Proc. Int'l. Conf. Software Maintenance. (2006)
72. Zhao, J., Xu, B.: Measuring Aspect Cohesion. In: Proc. Int'l. Conf. Fundamental Approaches to Software Engineering. (2004)
73. Gelinas, J.F., Badri, M., Badri, L.: A Cohesion Measure for Aspects. J. Object Technology **5**(7) (2006)

74. Lopez-Herrejon, R., Apel, S.: Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In: Proc. Int'l. Conf. Fundamental Approaches to Software Engineering. (2007)
75. Smith, D.R.: A Generative Approach to Aspect-Oriented Programming. In: Proc. Int'l. Conf. Generative and Component-Based Software Engineering. (2004)
76. Briand, L.C., Dzidek, W.J., Labiche, Y.: Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In: Proc. Int'l. Conf. Software Maintenance. (2005)