JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 9, No. 1, January–February 2010

How AspectJ is Used: An Analysis of Eleven AspectJ Programs

Sven Apel, Department of Informatics and Mathematics, University of Passau, Germany

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 crosscutting mechanisms (introductions and method extensions), and 2 % advanced crosscutting mechanisms (homogeneous advice or advanced dynamic advice). Based on these results we initiate a discussion on the trade-off between expressiveness and simplicity of languages that support the modularization of crosscutting concerns.

1 INTRODUCTION

While many studies have explored the capabilities of *aspect-oriented programming (AOP)* [36] to improve the modularity, customization, and evolution of software [19, 63, 18, 40, 27, 26, 29], 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 (25,300 downloads of aspectj-1.6.x.jar and 33,547 downloads of aspectj-1.5.x.jar – status March 2, 2009), we and others [55] 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 [33, 3, 39, 42]. 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 [35]. *Basic crosscutting mechanisms* are simple *introductions* (a.k.a. inter-type declarations) and method extensions; *advanced crosscutting mechanisms* include pieces of advice that advise whole sets of join points and that are triggered by runtime events [2, 4]. But how often are basic and advanced mechanisms actually used? In this paper, we define metrics to answer this question.

Our analysis of the eleven AspectJ programs shows on average 86 % of the code

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

Cite this article as follows: Sven Apel: *How AspectJ is Used: An Analysis of Eleven AspectJ Programs*, in Journal of Object Technology, vol. 9, no. 1, January–February 2010, pages 117–142,
http://www.jot.fm/issues/issue_2010_01/article2/

is object-oriented, 12 % uses basic crosscutting mechanisms, and 2 % uses advanced crosscutting mechanisms. This is the first time to our knowledge that a reasonable number of AspectJ programs has been analyzed and actual percentages reported.

Based on these findings we want to initiate a discussion on the trade-off between expressiveness and simplicity of languages that support the modularization of crosscutting concerns. On the one hand, there are languages like AspectJ that provide a wide variety of sophisticated language constructs, such as pointcuts and advice, that allow programmers to modularize concerns that crosscut the dynamic computation of a program and that affect many join points [45]. On the other hand, there is a wide variety of languages that support only basic crosscutting mechanisms, such as traits [22], mixins [25, 15], and virtual classes [43, 49], which can be used to extend single classes and methods. These languages are simpler but also less expressive as they do not support dynamic and homogeneous crosscutting. In current language design, there seems to be a trade-off between expressiveness and simplicity, which we will discuss here.

Several researchers have criticized the advanced crosscutting mechanisms of languages like AspectJ. Mechanisms like pointcuts and advice violate the principle of information hiding and hamper program comprehension, maintenance, and evolution [54, 1]. Looking at the usage distribution of crosscutting mechanisms in the analyzed AspectJ programs (12 % basic crosscutting mechanisms and 2 % advanced crosscutting mechanisms) the question arises whether or not a simpler language would have been more appropriate – a language that is less expressive, but sufficient for 98 % of the code of the analyzed programs (86 % object-oriented mechanisms plus 12 % basic crosscutting mechanisms). Recently, there has been an advent of programming languages that are simpler than AspectJ but still support basic crosscutting mechanisms, e.g., Jak [11], FeatureC++ [5], ContextL [31], Scala [49], Jiazzi [46], Classbox/J [14], and Jx [48], so that it is worth analyzing and discussing their relation to languages like AspectJ and their relative benefits and drawbacks.

Furthermore, a significant volume of prior work in the areas of programming languages [11, 5, 31, 49, 46, 14, 48], generative programming [62, 39, 58, 12, 9, 8, 10], and software design [52, 59, 32] has shown that programs can be created *largely* without advanced crosscutting mechanisms. The statistics that we report on AspectJ usage are consistent with this observation and fertilizes the discussion of the trade-off between language expressiveness and simplicity. We document and discuss 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) [20], (static/dynamic) [47], and (basic/advanced) [4]. We use an example to illustrate them all.

```
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() { ... } }
```

Figure 1: A simple graph implementation.

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 black and the code of Weight is *slanted and red*.

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 homo-

geneous 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 [61,47]: 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 languages express such advice as *method extensions* via subclassing (virtual classes or mixins), method overriding, and related mechanisms [43, 15, 25, 11, 48, 14, 23]. Dynamic crosscutting mechanisms in AspectJ transcend object-oriented mechanisms when they effect events other than singleton method executions (e.g., [50, 44]). 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 simple 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?), we will argue that the 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 advanced pattern-matching mechanisms of AspectJ.

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. Suppose 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]

[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.

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

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 or the static structure of a program.

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 pieces of advice implement simple method extensions.

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 [33]. 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 have 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

[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/

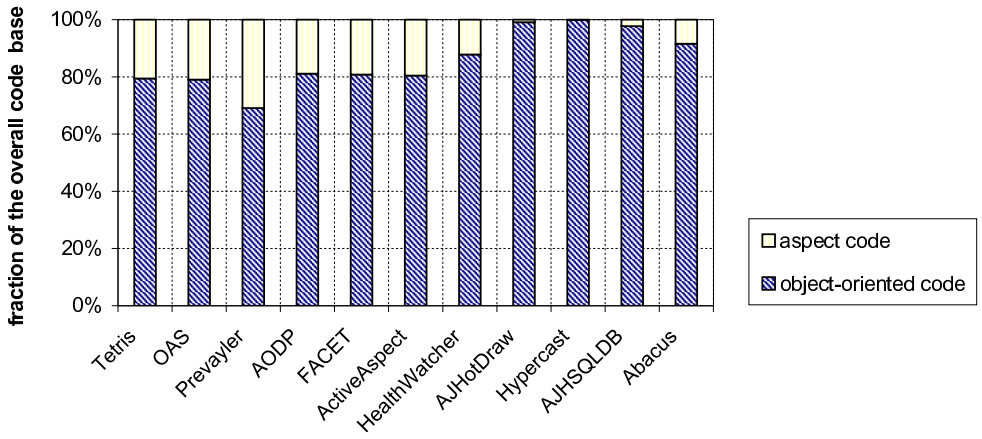


Figure 2: Fractions of aspect code and object-oriented code of the overall code base.

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).

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 no longer exist. Our statistics and results should be interpreted with this fact in mind. A comparison of development aspects and aspects actually deployed could be a topic of further research.

Statistics and Interpretation

The raw data that our statistics are based on can be requested from the authors. We begin with presenting the statistics and subsequently we interpret them.

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 ($3 \pm 4\,\%$).

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)

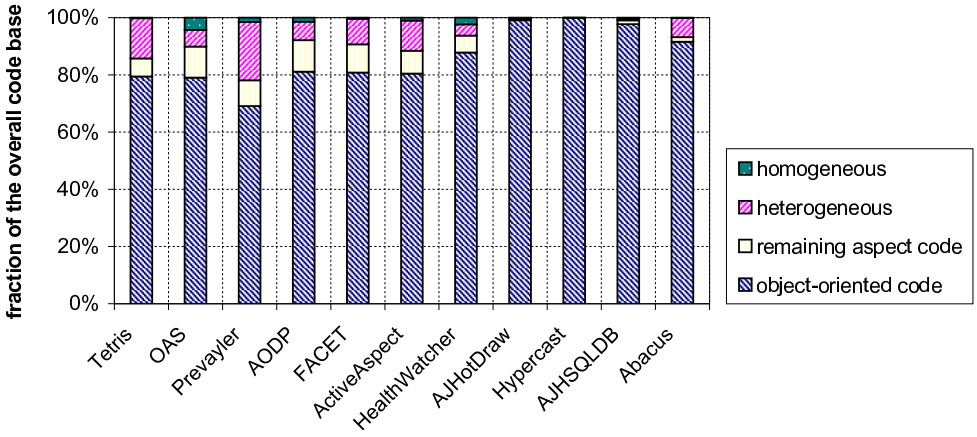


Figure 3: Fractions of homogeneous and heterogeneous crosscuts of the overall code base.

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 $6\,\%$ 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 ($2 \pm 1\,\%$) than in larger programs ($0.2 \pm 0.3\,\%$).

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 ($6 \pm 9\,\%$). On average, the small programs achieve a slightly larger reduction ($7 \pm 8\,\%$) than larger programs ($6 \pm 11\,\%$). Notice, the smallest program (Tetris) had no reduction, while the second largest program (AJHSQLDB) had the highest $23\,\%$.

SDC Metric

Figure 5 shows the fractions of inter-type declarations (static) and pieces of advice (dynamic). 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

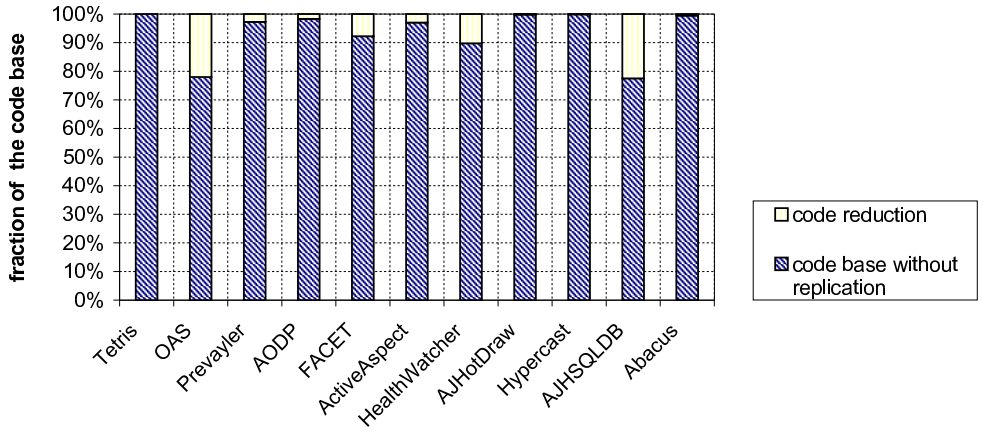


Figure 4: Code reduction achieved by aspects.

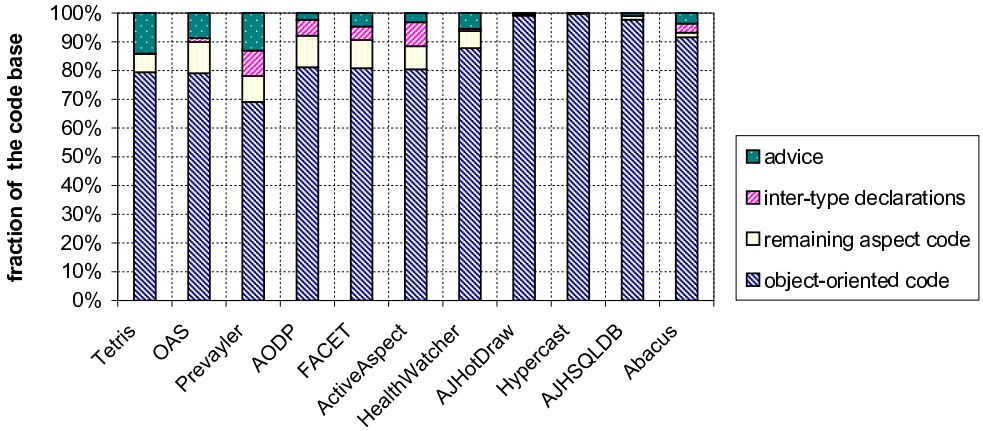


Figure 5: Fractions of static and dynamic crosscuts of the overall code base.

code base of large programs. In smaller programs advice ($7 \pm 5\,\%$) accounts for a slightly larger fraction that inter-type declarations ($4 \pm 4\,\%$).

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 ($0.2 \pm 0.3\,\%$).

Discussion

Figure 7 depicts the fractions of the code base of the AspectJ programs that exploit advanced crosscutting mechanisms (i.e., homogeneous advice and homogeneous

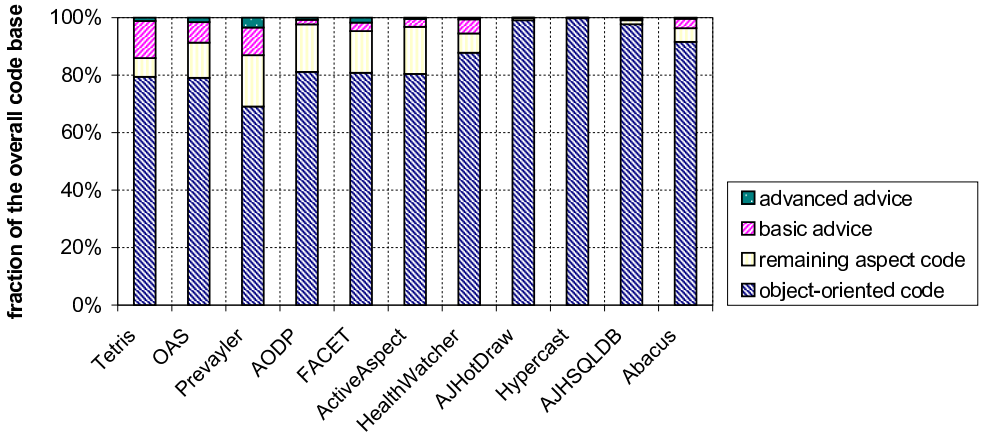


Figure 6: Fractions of basic and advance advice of the overall code base.

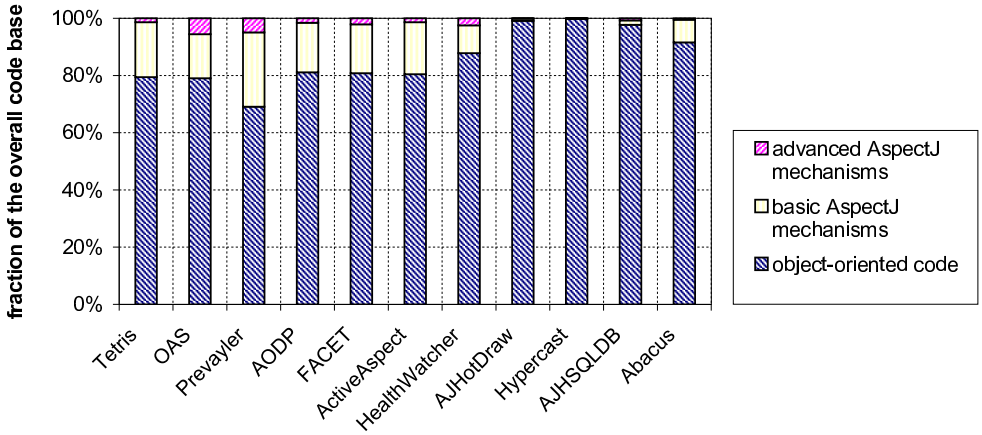


Figure 7: Fractions of basic AspectJ and advanced AspectJ mechanisms of the overall code base.

introductions, and advanced advice) and basic crosscutting 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 object-oriented code.

Our use of percentages of the overall code base does 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 crosscutting mechanisms; the rest is basic crosscutting mechanisms.

The $6 \pm 9\,\%$ code reduction that we have observed is in line with prior work on

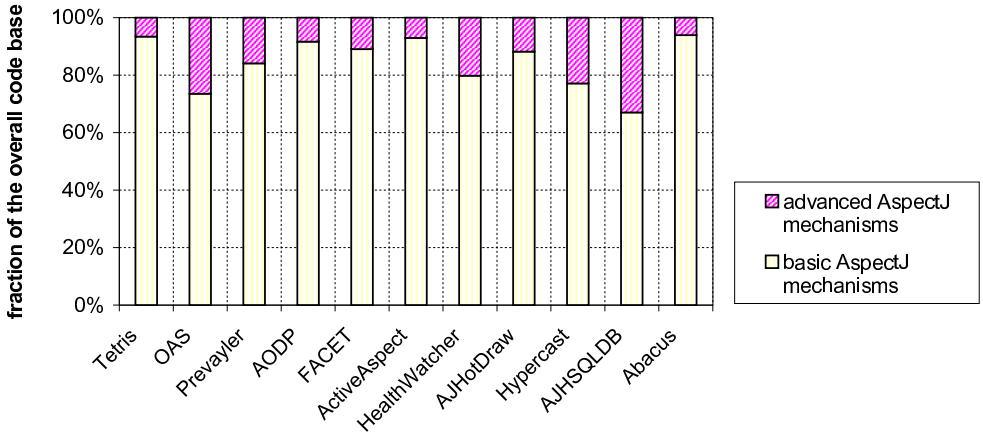


Figure 8: Fractions of AspectJ and advanced AspectJ mechanisms of a program's aspect code.

clone detection that conjectures that 5–15 % of large software projects are clones, i.e., replicated code fragments [13]. 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 [7], so 5–15 % may be an upper bound that aspects can not reach.

5 EXTENDED OBJECT-ORIENTED LANGUAGES

In this section, we illustrate how languages that support only basic crosscutting mechanisms can be used to implement most of the aspects found in the eleven AspectJ programs, namely 85 % of all aspect code. We call these languages *extended object-oriented languages* since they add only a few new concepts to the standard repertoire of object-oriented languages. Of course, we could have named them also basic aspect-oriented languages, although they do not support quantification [24].

Collaborations and Refinements

Like mainstream object-oriented languages, extended object-oriented languages, e.g., Jak [11], FeatureC++ [5], ContextL [31], Scala [49], Jiazzi [46], Classbox/J [14], and Jx [48], provide abstractions like classes, objects, and methods. Additionally, extended object-oriented languages allow a programmer to extend a given class without modifying it source code. This is also called the *open class pattern* [17] and there is a wide variety of mechanisms to implement this pattern, e.g., traits [22], mixins [25, 15], virtual classes [43, 49], nested inheritance [48], refinements [11]. For simplicity, we call such extensions *refinements*.

A further crosscutting mechanism of extended object-oriented languages is the ability to group multiple classes and refinements into an enclosing module. For

```
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() { ... } }
```

Figure 9: Implementing WEIGHT with a collaboration module in Classbox/J.

example, with Jx's nested inheritance we can introduce multiple new classes and refine multiple 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. For simplicity, we call such an enclosing module a *collaboration module*, inspired by the work on collaboration-based design [60, 53]. A collaboration module containing multiple classes and refinements crosscuts several places in a base program, so it is a basic crosscutting mechanism [53].

The BASICGRAPH program and the WEIGHT feature of Section 2 can be implemented with collaboration modules and refinements. 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. The classbox WEIGHT 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).

Advanced Aspects

Not all concerns can be compactly expressed just by simple introductions and method extensions (e.g., by basic crosscutting mechanisms such as collaboration modules and refinements). Sometimes there is a redundancy in the introductions or in the method extensions implementing a concern, which makes the crosscut homogeneous. Consider the COLOR feature of the BASICGRAPH program of Section 2. Its representation in Classbox/J is shown in Figure 10.

```
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 }
```

Figure 10: Implementing COLOR with a collaboration module in Classbox/J.

```
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  }
```

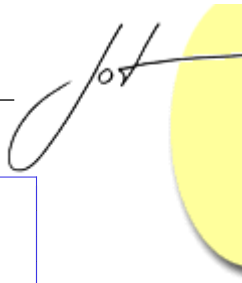
Figure 11: Implementing COLOR with an aspect in AspectJ.

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 advanced 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 module applying a refinement 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



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

Figure 12: Implementing PrintHeader with an aspect in AspectJ.

```
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 }
```

Figure 13: Implementing PrintHeader with a collaboration module in Classbox/J.

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).

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 [54, 1], especially in situations where few join points are affected (e.g., compare Figure 12 with Figure 13) [33]. Regardless, we found that such dynamic crosscuts occur rarely (1 %).

Corroborating Evidence

A review of the eleven AspectJ programs reveals that aspects often extend multiple objects by multiple pieces of advice and inter-type declarations. In other words, the extensions an aspect typically makes to a base program are heterogeneous. Furthermore, we have 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.

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 version written in Jak, a extended object-oriented language [39]. Also, Xin et al. observed that the Jiazzi version of FACET is close to the aspect-oriented version [62].

Furthermore, we contacted the developers of the eleven AspectJ programs to ask about their experience with the occurences and useability of basic and advanced crosscutting mechanisms and their relationship to mechanisms of collaboration modules and refinements, as explained above:

- The developer of Abacus (C. Zhang) confirmed that his aspect composition was driven by superimposition of views (collaboration modules). He extended existing classes by using AspectJ-style mixins (refinements) and implemented the methods declared by interfaces.

- The developer of FACET (R. Pratap) answered that he definitely had to think of collaboration-style extensions while implementing features in FACET.

- The developer of ActiveAspect (W. Coehlo) said that the individual extensions of classes applied by his aspect were heterogeneous. Generally, he did not think in terms of crosscutting dynamic computation.

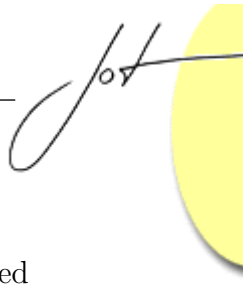
- The developer of AJHotDraw (M. Marin) explained that there are a number of refactored concerns in AJHotDraw whose implementations consist of classes with multiple different refinements and that interact in various ways.

- The developer of Prevayler (I. Godil) confirmed that there are aspects that use basic crosscutting mechanisms and that correspond to collaboration modules, which is in line with [39].

- K. Sullivan reported that his work on Hypercast was not intended (and did not) explore the use basic and advanced crosscutting mechanisms, but rather to take some easy/classical applications of aspects, such as logging, and to use them to evaluate the notion of 'obliviousness'.

- The developer of HealthWatcher (S. Soares) was unfamiliar with the concept of basic and advanced crosscutting mechanisms as well as of the mechanisms of collaboration modules and refinements. He was unable to say whether his aspects are related to collaboration modules and refinements or not.

We did not receive responses from the Tetris, OAS, and AJHSQLDB developers. The majority of responses indicate that the developers noticed the relationship and resemblance of aspect mechanisms to collaboration modules and refinements in their work – although not all were aware of the concept or the term.



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 [20,19,47,2]. 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.

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 only few published, non-trivial programs using AspectJ in open literature. Moreover, the eleven programs have been mainly developed for academic purposes. A 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. Even assuming negative responses, a majority of the authors indicated that they were aware of resemblance of their aspects with collaboration modules.

Another possible issue is that because the term 'collaboration' is overloaded and the terms 'basic' and 'advanced crosscutting mechanisms' are unclear, there could be misunderstandings between the developers and us. To minimize this, we defined the terms in our correspondences with authors, as explained in the previous sections.

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 [33,3,39,42]. We cite in related work additional corroborating evidence.

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

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

And finally, we and others have been building systems for years by composing collaboration modules and refinements 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.

The trade-off between language expressiveness and simplicity in software development is addressed in the next section.

6 DISCUSSION: EXPRESSIVENESS VS. SIMPLICITY

In the 2 % of the code base, where advanced crosscutting mechanisms are used, let us assume that the use of AspectJ is appropriate. For the remaining 98 %, an extended object-oriented language is appropriate. The question is, on the one hand, whether one could not simply use an extended object-oriented language for all crosscutting concerns, emulating advanced crosscutting mechanisms (2 % of the code base) and accepting a certain amount of boiler plate code and code replication, as explained in Section 5. On the other hand, one could simply use an (AspectJ-like) aspect-oriented language for all crosscutting concerns, as it has been done in the eleven AspectJ programs. Is there a difference between the two options?

We argue that there is indeed a difference. The two options reveal a trade-off between expressiveness and simplicity of programming languages that support the modularization of crosscutting concerns. Clearly, AspectJ-like languages are very powerful and expressive but this comes at a cost. It has been argued that the advanced crosscutting mechanisms violate the principle of information hiding and hamper program comprehension, maintenance, and evolution [54, 1]. Extended object-oriented languages are simpler but cannot express all kinds of crosscutting concern concisely.

We believe that our findings in the analysis of the eleven AspectJ programs can (re)initiate a discussion on this issue. The result of our study is that only 2 % of the code base have been implemented with advanced crosscutting mechanisms. Is this fraction worth of using AspectJ and inviting many problems caused by its advanced mechanisms? Or is this fraction relevant and justifies the use of AspectJ?

A compromise may be to use the basic crosscutting mechanisms of extended object-oriented languages (collaboration modules and refinements) for heterogeneous and basic dynamic crosscutting concerns and the advanced crosscutting mechanisms of aspect-oriented languages (pointcuts and advice) for homogeneous and advanced dynamic crosscutting concerns? Previous work in support of this position is [56, 6, 38, 30, 4].

7 RELATED WORK

Collaborations and Refinements

Although the concept of collaboration modules predates AOP by quite some time [52, 10, 60], mainstream programming languages have been very slow to support them. AspectJ has filled the vacuum [45]. The weak support of collaboration modules in aspect and non-aspect mainstream languages has contributed to a general confusion regarding their relationship to, importance to, and frequency in crosscutting concerns. Nevertheless, several studies demonstrate that extended object-oriented languages suffice in implementing large applications [10, 12, 8, 9, 11, 58].

There are many languages that incorporate the concept of refinements and collaborations in its language design, although sometimes called differently, e.g., Jak [11], FeatureC++ [5], ContextL [31], Scala [49], Jiazzi [46], Classbox/J [14], Jx [48]. 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 modules and advanced AspectJ mechanisms to benefit from both worlds, e.g., Relationship Aspects [51], CaesarJ [6], FeatureC++ [5], Aspectual Collaborations [38], and Object Teams [30], 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) [19]. 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 collaboration modules and refinements.

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200–400 KLOC) [18]. 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) [40]. Tesanovic et al. implemented 10 AspectC++ aspects for quality-of-service management in database systems [57]; the aspects implement predominantly heterogeneous and basic dynamic crosscutting concerns, which is in line with our study.

Lopez-Herrejon et al. analyzed an AspectJ implementation of the AHEAD Tool Suite [42]. 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 [29]. 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 basic and advanced crosscutting mechanisms

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 [34], harmless and harmful advice [21], or observers and assistants [16], 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 [63]. 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 object-oriented counterparts [27, 37]. They observe that the AOP variants have fewer lines of code than their object-oriented equivalents (12 % code reduction).

Zhao and Xu propose several metrics for aspect cohesion based on aspect dependency graphs [64]. Gelinas et al. discuss previous work on cohesion metrics and propose an approach based on dependencies between aspect members [28].

All of the above proposals and case studies take neither the structure of crosscutting concerns nor the difference between basic and advanced crosscutting mechanisms into account.

Lopez-Herrejon et al. propose a set of code metrics for analyzing the crosscutting structure of aspect-based product lines [41]. 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 CONCLUSION

We have analyzed eleven AspectJ programs of different sizes and complexity. We found that on average 2 % of the code base is associated with advanced crosscutting mechanisms; 12 % is associated with basic crosscutting mechanisms; and 86 % is object-oriented. These numbers indicate that languages that provide only basic crosscutting mechanisms are appropriate to implement a large extent of the analyzed programs. Instead, AspectJ, a more powerful language, has been used. Since voices have been raised that advanced crosscutting mechanisms, like the ones provided by AspectJ, may be too powerful, we initiated a discussion on the trade-off between expressiveness and simplicity. Currently, it is not clear if the advanced crosscutting mechanisms provided by AspectJ outweigh the problems caused by the violation of the principle of information hiding. Our studies can help to expedite the academic discussion and to conduct further case studies on this issue.

ACKNOWLEDGEMENTS

We thank D. Batory, K. Fisler, M. Grechanik, C. Kästner, P. Kim, S. Krishnamurthi, C. Lengauer, D. Perry, and C. T. Shepherd for their helpful comments on earlier drafts of this paper. The author's research is sponsored by the German Science Foundation (DFG), # AP 206/2-1.

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. Int'l. Conf. Generative and Component-Based Software Engineering*, 2006.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering*, 34(2), 2008.

[5] S. Apel, M. Rosenmüller, T. Leich, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l. Conf. Generative and Component-Based Software Engineering*, 2005.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Trans. Aspect-Oriented Software Development*, 1(1), 2006.

[7] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. Work. Conf. Reverse Engineering*, 1995.

[8] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating Reference Architectures: An Example from Avionics. In *Proc. Int'l. Symp. Software Reuse*, 1995.

[9] D. Batory, C. Johnson, B. MacDonald, and D. v. Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Trans. Software Engineering and Methodology*, 11(2), 2002.

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

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

[12] D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *J. Intell. Inf. Syst.*, 9(2), 1997.

[13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. Int'l. Conf. Software Maintenance*, 1998.

[14] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[15] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proc. Europ. Conf. Object-Oriented Programming and Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 1990.

[16] C. Clifton and G. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Proc. Int'l. Workshop Foundations of Aspect-Oriented Languages*, 2002.

[17] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145. ACM Press, 2000.

[18] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, 2003.

[19] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, 2004.

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

[21] D. S. Dantas and D. Walker. Harmless Advice. In *Proc. Int'l. Symp. Principles of Programming Languages*, 2006.

[22] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Trans. Programming Languages and Systems*, 28(2), 2006.

[23] E. Ernst. Higher-Order Hierarchies. In *Proc. Europ. Conf. Object-Oriented Programming*, 2003.

[24] R. Filman and D. Friedman. Aspect-Oriented Programming Is Quantification and Obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[25] R. Bruce Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. Int'l. Conf. Functional Programming*, 1998.

[26] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, A. v. Staa, and C. Lucena. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications*, 2003.

[27] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. v. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, 2005.

[28] J. F. Gelinas, M. Badri, and L. Badri. A Cohesion Measure for Aspects. *J. Object Technology*, 5(7), 2006.

[29] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Santa-Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proc. Europ. Conf. Object-Oriented Programming*, 2007.

[30] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proc. Int'l. Net.ObjectDays Conf.*, 2002.

[31] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *J. Object Technology*, 7(3), 2008.

[32] R. Johnson and B. Foote. Designing Reusable Classes. *J. Object-Oriented Programming*, 1(2), 1988.

[33] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features using AspectJ. In *Proc. Int'l. Software Product Line Conf.*, 2007.

[34] S. Katz. Aspect Categories and Classes of Temporal Properties. *Trans. Aspect-Oriented Software Development*, 1(1), 2006.

[35] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*. Springer, 2001.

[36] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, 1997.

[37] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. v. Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. Int'l. Conf. Software Maintenance*, 2006.

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

[39] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proc. Int'l. Conf. Software Engineering*, 2006.

[40] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proc. Int'l. EuroSys Conf.*, 2006.

[41] R. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In *Proc. Int'l. Conf. Fundamental Approaches to Software Engineering*, 2007.

[42] 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.

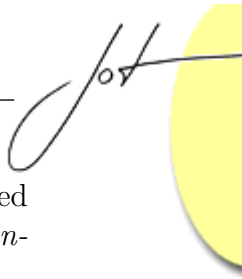
[43] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 1989.

[44] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proc. Asian Symp. Programming Languages and Systems*, 2003.

[45] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. Europ. Conf. Object-Oriented Programming*, 2003.

[46] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[47] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proc. Int'l. Symp. Foundations of Software Engineering*, 2004.



[48] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[49] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[50] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proc. Europ. Conf. Object-Oriented Programming*, 2005.

[51] D. J. Pearce and J. Noble. Relationship Aspects. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, 2006.

[52] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *J. Object-Oriented Programming*, 5(6), 1992.

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

[54] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[55] M. Störzer. *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.

[56] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l. Conf. Software Engineering*, 1999.

[57] A. Tesanovic, M. Amirijoo, M. Bjork, and J. Hansson. Empowering Configurable QoS Management in Real-Time Systems. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, 2005.

[58] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proc. Int'l. Conf. Generative and Component-Based Software Engineering*, 2006.

[59] M. VanHilst and D. Notkin. Decoupling Change from Design. In *Proc. Int'l. Symp. Foundations of Software Engineering*, 1996.

[60] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

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

[62] 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.

[63] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[64] J. Zhao and B. Xu. Measuring Aspect Cohesion. In *Proc. Int'l. Conf. Fundamental Approaches to Software Engineering*, 2004.

ABOUT THE AUTHORS

Sven Apel is a post-doctoral associate at the Chair of Programming at the University of Passau, Germany. He received a Ph. D. in Computer Science from the University of Magdeburg, Germany in 2007. His research interests include advanced programming paradigms, software product lines, and algebra for software construction. He can be reached at apel@uni-passau.de. See also http://www.infosun.fim.uni-passau.de/cl/staff/apel/.