Reducing the Complexity of AspectJ Mechanisms for Recurring Extensions

Martin Kuhlemann
School of Computer Science,
University of Magdeburg, Germany
kuhlemann@iti.cs.uni-magdeburg.de

Christian Kästner
School of Computer Science,
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

ABSTRACT

Aspect-Oriented Programming (AOP) aims at modularizing crosscutting concerns. AspectJ is a popular AOP language extension for Java that includes numerous sophisticated mechanisms for implementing crosscutting concerns modularly in one aspect. The language allows to express complex extensions, but at the same time the complexity of some of those mechanisms hamper the writing of simple and recurring extensions, as they are often needed especially in software product lines. In this paper we propose an AspectJ extension that introduces a simplified syntax for simple and recurring extensions. We show that our syntax proposal improves evolvability and modularity in AspectJ programs by avoiding those mechanisms that may harm evolution and modularity if misused. We show that the syntax is applicable for up to 74 % of all pointcut and advice mechanisms by analysing three AspectJ case studies.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [22] gains momentum in current research and practice [33]. AOP aims at reducing complexity by modularizing crosscutting concerns. AspectJ is a popular AOP language extension for Java [26, 21]. AspectJ includes over different 40 keywords for very different extensions, from static inter-type declarations, over possibilities to modify the inheritance hierachy, over dynamic extensions of the control flow and many more. Even though the language is already fairly complex and allows sophisticated extensions, ongoing research still proposes additional AspectJ mechanisms and keywords to further improve separation of concerns, e.g., [1, 16, 24, 11].

AspectJ was frequently suggested for implementing *software product lines (SPL)* [13, 10, 39, 27, 19, 37, 9], that aim at creating tailored programs distinguished by features. In a previous case study we decomposed the embedded database engine Berkeley DB and implemented it as a SPL with AspectJ [20]. However, during implementation we noticed that the sophisticated AspectJ mechanisms are used rarely (22-31 %) [20]. Such observations were also confirmed in case studies by others [4, 3, 25]. Most AspectJ mechanisms used in practice implement simple extensions (68-78 %), however those simple extensions are still implemented with the complex syntax of AspectJ although they don't need the advanced capabilities. Especially *pointcut and advice (PCA)* mechanisms are written in an overly complex syntax

To support implementation of simple extensions, as they are especially needed for SPLs, we suggest an AspectJ extension with a simplified syntax. We focus on PCA mechanisms because we observed that they are most affected by unnecessary overhead. We observed that developers often 'misuse' AspectJ mechanisms to abbreviate the AspectJ syntax and thus cause problems of modularity and evolution [36, 14, 38, 20]. To overcome problems resulting from complex syntax we propose to add a simplified syntax for existing mechanisms to AspectJ. We propose to use our simplified syntax for implementing 'simple' method extensions while the common AspectJ syntax should be used for remaining AOP tasks. By only adding to existing mechanisms, we do not neglect the need for sophisticated AspectJ mechanisms.

We show that our syntax reduces the allurement of misusing mechanisms and thus improve evolvability and modularity. We show that our syntax covers a wide range of implementation problems by analyzing three AspectJ case studies among them two SPL and one general purpose AspectJ software.

2. BACKGROUND

AspectJ includes numerous mechanisms; among them PCA and introductions [26, 21]. A *pointcut* selects events during the execution of the program (*join points*) that should be extended with *advice*.

The set of join points to advise is described in pointcuts using so called *pointcut designators* where each designator defines a property of advised join points, e.g., the designator `within` matches only join points located inside one specific class but not inside its subclasses [26], while the `cflow` designator defines that matched join points must be in the control flow of some other join points. Pointcut designators can also expose context information of the join point [26, 21]. For example, the pointcut designator `args` allows advice to access method arguments and the designators `this` and `target` provide context information on which object the join point occurred. Pointcut designators can be composed logically and equipped with wildcards. Both possibilities enable to *quantify* over multiple join points with a single pointcut. If a pointcut quantifies over different join points, the implemented crosscutting concern is *homogeneous*, otherwise it is *heterogeneous* [6]. If the join points a pointcut matches can be determined statically, the PCA is called *basic*, all PCAs that are based on runtime conditions, need dynamic reasoning, e.g., when using the *cflow* or *if* designators, are called *advanced* [4].

Figure 1 shows a class `Label` and an aspect `TriggerLabelEvent`. The aspect contains basic advice (Lines 20–22 and 26–28) that extends the class at different points to invoke the method

Workshop AOPLE '07 Oct. 4, 2007 Salzburg, Austria

```
1  public class Label{
2    String _text;
3    Observer _observer;
4    public void setText(String newText){
5      this._text = newText;
6    }
7
8    public void updateObservers(String message){
9      this._observer.notify()
10   }
11
12   public static void main(String[] args) {
13     Label l= new Label();
14     l.setText(args[0]);
15   }
16 }
```

```
17 public aspect TriggerLabelEvent {
18   pointcut LabelChangeExec(Label l, String newText):
       execution(void Label.setText(String))
       && within(Label) && this(l) && args(newText);
19
20   before(Label l, String newText):
       LabelChangeExec(l, newText){
21     l.updateObservers(newText);
22   }
23
24   pointcut LabelChangeCall(Label l, String newText):
       call(void Label.setText(String)) && target(l)
       && args(newText);
25
26   before(Label l, String newText):
       LabelChangeCall(l, newText){
27     l.updateObservers(newText);
28   }
29 }
```

Figure 1: Pointcut and advice in AspectJ.

```
1  pointcut parameterProvider(Button b) :
     execution(* Button.*(..)) && this(b) && within(Button);
2
3  pointcut operationToExtend(Label l, String newText):
     execution(void Label.setText(String)) && this(l)
     && args(newText) && within(Label);
4
5  pointcut wormhole(Button b, Label l, String newText) :
     operationToExtend(l, newText)
     && cflow(parameterProvider(b));
6
7  after(Button b, Label l, String newText) returning :
     wormhole(b, l, newText) {
8    l.print("Button"+b+" set my text to: "+newText);
9  }
```

Figure 2: Passing parameters using the AOP wormhole pattern.

```
1 public class SpecialLabel extends Label {
2   public void setText(String newText){
3     super.setText(newText);
4     updateObservers(newText);
5   }
6 }
```

Figure 3: Method extension in OOP.

`updateObservers` (Lines 21 and 27). The advice associated to the `execution` pointcut (Lines 20–22) extends class `Label` and is invoked before the execution of the method `setText` in class `Label` (Lines 4–6). The advice associated to the `call` pointcut (Lines 26–28) is performed before the method `setText` is called (e.g., Line 14). Note that both PCA effect the same observable behavior at runtime [25].

In Figure 2 we logically compose pointcut designators to pass parameters to a method using the *Wormhole Pattern* [26]. We intercept the execution of each method of class `Button` (Line 1) and define the `Button` object as a parameter for the method `setText` of the class `Label` (Line 5). In associated advice (Lines 7–9) we use the passed `Button` object to manipulate the `setText` computation process, i.e., the `Button` object is used by the method extension as a parameter (Line 8). The depicted PCA frequently occur in combination to pass parameters to methods in AOP.

3. WHY IS ASPECTJ DIFFICULT?

In this section we review difficulties when implementing simple extensions with AspectJ that we tackle in this paper. We observed these problems in several AspectJ programs and SPLs and some are well known in current research.

Misused Quantification.

Quantification allows to match multiple join points with a single pointcut, e.g., using wildcards. Quantification is often considered as a fundamental concept of AOP and often used to modularize previously scattered code [12]. However, we observed that wildcards are frequently misused to abbreviate the syntax of AspectJ without wanting to match multiple join points, e.g., when implementing single method extensions with pointcuts like `execution(* setText(..))` [20, 2].

Another extension that can be regarded as misuse is related to `call` pointcuts. While method extensions are usually implemented using `execution` advice, `call` advice that quantifies all calls to the method and does not reference the calling object has an equivalent effect [25].

Using quantification unneeded due to syntax abbreviation hampers subsequent evolution of the software and may result in incorrect advice application, e.g., the developer is not warned by the compiler if an incorrect set of join points gets advised. This and similar problems are called *arranged pattern problem*, *fragile pointcut problem*, or *evolution paradox* [14, 36, 34, 31, 38].

Broken Encapsulation.

Encapsulation of modules hides implementation details of classes behind an interface. This allows to reason about a class in isolation while the implementations of other classes remain hidden [23, 35]. If the developer uses AspectJ's `call` pointcut designator, the developer needs to know all method implementations of classes. Since method implementations are hidden from interfaces, AspectJ's `call` advice can break encapsulation of extended classes. This raises complexity because broken encapsulation does not allow to reason about modules in isolation [23, 18]. `Execution` advice is performed before, after, or instead whole methods. If extended methods are declared inside the interface of a class, `execution` advice does not break encapsulation and thus does not raise complexity [18]. Hence, we reason that it is beneficial to integrate `execution` advice into our simplified syntax but not `call` advice.

Verbose Syntax.

In AspectJ, programmers need to compose several pointcut designators to access join point parameters [20], e.g., to implement a simple method extension that accesses arguments of the advised method, these arguments have to be defined in the pointcut desig-

```
1 (before|after|around) (
    <returntype of method> <class>.<method>
2 ( (<type of parameter> <name of parameter>)* )
3 [throws <type of exception>]){
4   <statement>*
5 }
```

Figure 4: Grammar of simplified PCA syntax.

```
1 public aspect ExtendedTriggerLabelEvent {
2   before(void Label.setText(String newText)){
3     updateObservers();
4   }
5
6   before(void Label.setText(String newText)){
7       updateObservers();
8   }
9 }
```

Figure 5: Simplified syntax of pointcut and advice.

nators `execution` and `args` and once more in the advice declaration. We found designators to be verbose and to repeat informations. Parameters get further repeated if named and abstract pointcuts are used (cf. Fig. 1). These repetitions entrap the developer to misuse mechanisms like wildcards to abbreviate the code (e.g., in [15]). This causes the evolution paradox of AOP and pointcuts that are hard to read.

In Figure 1 we extend the method `setText` of class `Label` using the aspect `TriggerLabelEvent` in two alternative ways (Lines 18-22 and Lines 24-28). The syntax for both simple method extensions is verbose compared to the syntax of equivalent method extensions in other languages like Java [8]. For a simple method extension we need four pointcut designators: `execution`, `within`[1], `args`, and `this`. In Figure 3 we extend the method `setText` equivalently using the less verbose syntax of Java inheritance.

In a prior case study of an aspect-oriented product line [20] we experienced that we needed 3.45 designators per pointcut on average (many of them introducing a repetitive syntax of parameters) although we mainly implemented simple method extensions.

Third Person Perspective.

Another difficulty in implementing extensions is caused by what we call the *third person perspective (3PP)*. The 3PP means that advice is written from an outside perspective where the extended object is passed as a parameter. The language keyword *this* does not refer to the extended object as programmers familiar with OOP might expect but to the aspect itself. The 3PP is necessary to be able to specify homogeneous extensions that quantify over several classes. It allows different kinds of pointcuts (`call` and `execution` pointcuts; basic and advanced AspectJ mechanisms) to look similar. However, it hampers the writing of simple mechanisms. The 3PP may cause problems for Java programmers that want to extend methods using AspectJ, because they must switch between different representations constantly to implement Java base code, inter-type declarations (both written in first person perspective) and advice (written in 3PP) [20]. We also experienced that references to the extended object were by far more frequent than references to the aspect, again increasing the verbosity of the code because the developer has to include the `this` or `target` designator in most pointcuts.

4. IMPROVING THE ASPECTJ SYNTAX

To tackle the difficulties discussed in Section 3 we propose to add a simplified syntax for basic heterogeneous `execution` PCA and equivalent `call` PCA to AspectJ. That is, we introduce and additional syntax for simple method extensions. In Figure 5 we exemplify the aspect from Figure 1 with our syntax. After the *before*, *after* or *around* keyword we directly specify the method signature of the method to extend. This signature can be copied almost directly from the extended method. There is no further need to capture context information with *args* or *this* designators, in fact it is not possible to use any designators at all. Furthermore, we do not support the use of wildcards or the extension of private methods. Our syntax hides pointcut designators completely. To simplify the AspectJ syntax for these extensions further we use a first person perspective, i.e., we define the self-reference (`this` in Java) to reference the extended object (e.g., `Label` in Fig. 1) instead of the containing aspect. Methods of the containing aspect can still be referenced using the existing aspect method `aspectOf`. We present the grammar of this syntax in Figure 4 where italic tokens represent identifiers and blue and underlined tokens are keywords. Overall, the syntax integrates tokens familiar from AspectJ with a syntax familiar from overriding methods in Java. The new syntax is much briefer than the original AspectJ syntax.

Note, that we deliberately can only express heterogeneous and basic extensions. We made these restrictions because they significantly reduce complexity and because the majority of all extensions is heterogeneous and basic anyway [4], especially in SPLs [20].

The language extension can be implemented in the AspectJ compiler or even as an preprocessor that expands the new simplified syntax to an expression in the existing AspectJ syntax.

The new simplified syntax makes three contributions. First, by making PCA mechanisms less verbose simplifies their use and thus eliminates the allurement of dangerous syntax abbreviations that may lead to the pointcut fragility problem and related problems. Second, we encourage the use of those mechanisms that maintain encapsulation of classes and avoid complexity [18]. And third, the simplified syntax eliminates 3PP for simple extensions and simplifies the usage of AspectJ because OOP developers do not have to switch perspective. Although, we still need the 3PP for homogeneous and advanced crosscuts, we do not need it for major parts of aspect-oriented software development since researchers showed that 89–93 % of crosscutting extensions are heterogeneous [4, 20, 25, 30, 2] and 78 % are basic [4]. Beyond, we observed that remaining homogeneous `call` PCA oftentimes is equivalent to heterogeneous `execution` PCA [25] because `call` advice often does not reference the calling object (e.g., using designator `this`).

5. CASE STUDIES

We evaluate our syntax using three case studies. We pick the mentioned SPL implementation of Berkeley DB and two case studies by others to analyze what portion of extensions can be simplified in current AOP practice. The selected case studies of others are FACET[2] [19] and AJHOTDRAW[3]. In our own AspectJ SPL case study of Berkeley DB we observed that 74 % (358 of 482) of the used PCA mechanisms can be implemented in our simplified syn-

[1] Although surprising to new developers, the execution pointcut matches the execution of the specified method in the specified class and all subclasses. To prevent advising subclasses we have to include the additional within designator.

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

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

```
1 public aspect EnableCorbaAspect {
2   ConsumerAdmin around (Object poa, ConsumerAdminBase impl) throws Throwable: execution (ConsumerAdmin
      EventChannelImpl.CorbaGetConsumerAdminRef (Object, ConsumerAdminBase)) && args (poa, impl) {
3     POA Poa = (POA) poa;
4     org.omg.CORBA.Object obj = Poa.servant_to_reference (impl);
5     return ConsumerAdminHelper.narrow (obj);
6   }
7 }
```

(a)

```
1 public aspect EnableCorbaAspect {
2   around (ConsumerAdmin EventChannelImpl.CorbaGetConsumerAdminRef (Object poa, ConsumerAdminBase impl)) throws
      Throwable {
3     POA Poa = (POA) poa;
4     org.omg.CORBA.Object obj = Poa.servant_to_reference (impl);
5     return ConsumerAdminHelper.narrow (obj);
6   }
7 }
```

(b)

Figure 6: FACET aspect (excerpt) and its simplified appearance.

```
1 public aspect GroupCommandUndo {
2 after(Figure figure) : call(void DrawingView.addToSelection(Figure)) && withincode(void GroupCommand.groupFigures()) &&
    args(figure) {
3     gFigure = figure;
4   }
5 }
```

Figure 7: Excerpt of AJHOTDRAW aspect.

tax. FACET is an aspect-oriented SPL with 34 features that implements a CORBA event chanel. It includes 124 aspects that in sum include 49 PCA. We observed that 67 % (33 of 49) of these PCA can be implemented using our simplified syntax. On the other hand the AJHOTDRAW case study includes 31 aspects and 48 PCA. It was not designed as an SPL but still we observed that 46 % (22 of 48) of PCA can be implemented using our simplified syntax.

We observed that not all PCA could be transformed into our simplified syntax due to quantification and advanced PCA mechanisms. Figure 6 shows the aspect `EnableCorbaAspect` of FACET (Fig. 6-a) and its transformed code using our syntax (Fig. 6-b); we eliminate the designators `execution` and `args` and repeated declarations. Figure 7 excerpts PCA of the AJHOTDRAW aspect `GroupCommandUndo` which can not be converted because the PCA includes advanced pointcut mechanisms (`withincode`, Line 2) that refer to the method-calling object. If advice can not be transformed into basic `execution` advice – as in Figure 7 – we propose to use the verbose AspectJ syntax and 3PP. That way the developer gets alerted when he uses sophisticated AspectJ mechanisms which may harm the software if misused.

Our results that up to 74 % of all extensions could be replaced by simpler mechanisms is not surprising. Earlier research by Apel et al. has already indicated that crosscutting concerns largely can be implemented using simple method extensions instead of complex PCA mechanisms of AspectJ. In line with them we argue that complex AspectJ mechanisms should not be used for tasks they are not suitable for, but instead should be replaced with mechanisms implementing method extensions differently to current PCA syntax of AspectJ.

6. RELATED WORK

Several researchers observed that AOP has difficulties implementing heterogeneous crosscuts and simple method extension and propose to combine AOP with different paradigms, e.g., collaborations [7, 32, 17], component-based programming [7, 28, 29], and feature-oriented programming [5].

In contrast we propose to simplify the AspectJ syntax itself to improve the implementation of heterogeneous crosscuts *within* AspectJ. We showed that – equivalently to the referenced approaches – our simplified AspectJ syntax helps to reduce the impact of known AOP limitations, like fragile pointcuts or evolution paradox, as well as AspectJ limitations we experienced. In line with these researchers we observed the need to simplify frequently used AspectJ mechanisms. In contrast to them we argue that one AspectJ syntax (although slightly modified) is easier to use and understand than combinations of different paradigms each having its own characteristics, keywords, and constraints.

7. CONCLUSIONS

In this paper we proposed to add a simplified syntax for frequently used AOP mechanisms to AspectJ. We argue that for a high percentage of all extensions very simple language mechanisms are sufficient, while the existing AspectJ language introduces to much overhead. We just add a new simplified syntax while keeping the existing language constructs to keep expressiveness. However, the sophisticated mechanisms should only be used when needed.

We showed that our syntax leads to reduced impact of AspectJ difficulties, e.g., our syntax facilitates to use mechanisms considered beneficial for evolution and complexity, reduces verbosity of AspectJ implementations, and sets AspectJ mechanisms in line to equivalent mechanisms of other well known paradigms like collaborations. We argue that our syntax eliminates the allurement of misusing AspectJ mechanisms which causes problems; instead the

developer is alerted of using mechanisms that may harm modularity or evolvability of his software due to special syntax. By analyzing existing AspectJ case studies we could confirm that our approach is sufficient for a high percentage of extensions and we could improve major parts of those case studies.

8. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, 2005.

[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 *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 59–68, 2006.

[4] S. Apel, D. Batory, and M. Rosenmüller. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations? In *Workshop on Aspect-Oriented Product Line Engineering*, 2006.

[5] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounding Quantification in Incremental Designs. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 796–804, 2005.

[6] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 122–131, 2006.

[7] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *LNCS Transactions on Aspect-Oriented Software Development I*, 3880:135–173, 2006.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[9] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 50–59, 2003.

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

[11] M. Eichberg. The Proxy Inter-Type Declaration. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2004.

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

[13] M. L. Griss. Implementing product-line features by composing aspects. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[14] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[15] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, 2002.

[16] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 63–74, 2006.

[17] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of the International Net.ObjectDays Conference*, pages 248–264, 2002.

[18] M. Horie and S. Chiba. An Aspect-Aware Outline Viewer. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 71–75, 2006.

[19] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 38–45, 2002.

[20] C. Kaestner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2007.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[23] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 49–58, 2005.

[24] G. Kniesel and T. Rho. A Definition, Overview and Taxonomy of Generic Aspect Languages. *L'Objet*, 11(3):9–39, 2006.

[25] M. Kuhlemann, M. Rosenmüller, S. Apel, and T. Leich. On the Duality of Aspect-Oriented and Feature-Oriented Design Patterns. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2007.

[26] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[27] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

[28] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, 1999.

[29] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46:542–565, 2003.

[30] 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.

[31] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77, 2006.

[32] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–116, 1998.

[33] D. Sabbah. Aspects: from promise to reality. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 1–2, 2004.

[34] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Companion of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 481–497, 2006.

[35] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[36] M. Stoerzer and J. Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 00:653–656, 2005.

[37] A. Tesanovic, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proc. International Database Engineering and Applications Symposium*, pages 291–301, Washington, DC, USA, 2004. IEEE Computer Society.

[38] T. Tourwé, J. Brichau, and K. Gybels. On the Existence of the AOSD-Evolution Paradox. In *Workshop on Software-Engineering Properties of Languages for Aspect Technologies*, 2003.

[39] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139, New York, NY, USA, 2003. ACM Press.