Connecting Aspects in AspectJ: Strategies vs. Patterns

Stefan Hanenberg
University of Essen, Institute for Computer Science
Schützenbahn 70, D-45117 Essen, Germany
shanenbe@cs.uni-essen.de

Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstraße 164, D-53117 Bonn, Germany
costanza@cs.uni-bonn.de

ABSTRACT

Aspects in AspectJ can be connected to existing classes and applications in order to amend them with additional ancestors, methods and advice to existing methods. However, for concrete usage scenarios there are different options of how to use AspectJ's features, and these options deeply impact the opportunities for further evolution of both base classes and aspects.

The purpose of this paper is two-fold. First, it introduces *strategies* that describe these options and their specific tradeoffs. These strategies provide a common terminology and support developers in deciding which option to use in what situation. Second, their presentation obviously resembles the structure of well-known design patterns, but it is not clear to what extent they can rightfully be regarded as patterns themselves. This issue is discussed by giving two oppositional position statements.

1. INTRODUCTION

AspectJ developed at the Xerox Palo Alto Research Center is currently the most popular general purpose aspect language built on top of the programming language Java and offers additional composition mechanisms to modularize cross-cutting concerns. It supplies a class-like construct called *aspect* that permits to define code that cross-cuts a given application. Furthermore, it offers means to define *how* this code cross-cuts a given application. The usage of these language constructs has a direct impact on how reusable an aspect is and how easy it can be applied to new situations. So the developer has to be careful when designing aspects using those constructs because it might influence the evolution of the resulting software or the applicability of the developed aspects in an undesired way. Since the definition of how aspects cross-cut applications means to describe connections between aspects and applications, the main focus of developing aspects in AspectJ lies on these connections.

In this paper we describe strategies for connecting aspects to applications in AspectJ. These strategies are recurring in different contexts, so this collection of strategies can be regarded as a catalogue that gives developers an overview of techniques that are used often. Since they are presented in a form that resembles the structure of (design) patterns it seems reasonable to discuss the relationship between such strategies and patterns.

In section 2 and 3 we propose recurring strategies and exemplify their benefit in section 4. In section 5 and 6 we discuss in two oppositional statements the relationship between the proposed strategies and patterns. Finally, we summarize this paper.

2. STATIC CROSS-CUTTINGS

According to the AspectJ terminology, we use the term *static crosscutting* to describe crosscuttings that influence the interfaces of the involved types [2]. AspectJ provides a mechanism called *introduction* to achieve this kind of influence.

Direct Ancestor Introduction

It is often observable that different objects have common properties from a certain perspective. A perspective is a subjective view on the system, this means such mutuality is not intrinsic to those objects. From this perspective, all of those objects should be treated in the same way and therefore should be substitutable. In object-oriented programming substitutability is achieved by classification. Here classification does not occur because of intrinsic common properties of such objects, but because of an aspect specific view on the system. Therefore the classification is not part of the object definition, but part of an aspect definition. A *direct ancestor introduction* directly introduces an aspect-related, extrinsic ancestor to objects, i.e. the desired mutuality of objects is not intrinsic to those objects.

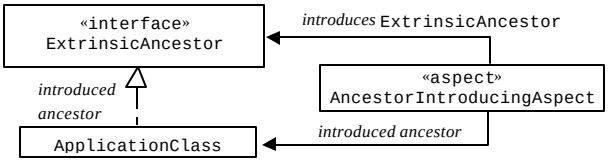

Figure 1: Direct Ancestor Introduction

The participants of this strategy are:

- *extrinsic ancestor*: the class, interface or aspect that contains the common properties.
- *ancestor-introducing aspect*: the aspect that defines the classification.

The consequences of using a *direct ancestor introduction* are:

- *extrinsic classification*: classification of objects is not only determined by the class definition, but also by the introducing aspect.
- *matching signatures*: when applying this strategy the developer must guarantee that the application related class is able to establish the introduced ancestor. For example, if the ancestor is an interface the developer has to guarantee that the class implements the methods of that interface.

AspectJ directly supports the direct ancestor introduction on the language level. This strategy just corresponds to the usual application of static cross-cutting for declaring an *implements* or *extends* relationship where the type pattern in the introduction directly corresponds to existing classes.

Direct Member Introduction

Sometimes it is desirable to add properties to selected objects because of a certain perspective. This means that from a certain perspective, different objects have common properties. A *direct member introduction* introduces extrinsic properties directly to objects without achieving substitutability of those objects.

The participant of this strategy is:

- *member-introducing aspect*: the aspect that contains the introduction. The introduction directly refers to the classes of those objects that should get the new members.

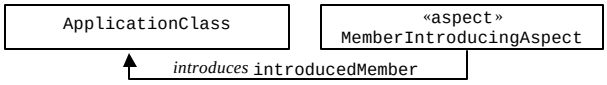

Figure 2: Direct Member Introduction

The consequences of using a direct member introduction are:

- *limited reusability*: the introductions themselves a hardly reusable since AspectJ does not permit to override introductions incrementally in an "object-oriented programming way". For example, if the developer decides later on that the introductions should be applied to further classes or interfaces, the aspect itself has to be modified.
- *members inherent to the aspect:* the introduced members are extrinsic to the objects. Therefore the developer has to guarantee that only clients that are aware of the introducing aspect can use them.
- *no substitutability:* although different classes get common properties their instances are still not substitutable.
- *member conflicts*: The developer has to guarantee that there are no conflicts between the extrinsic and intrinsic members. For example, no extrinsic member's identifier is allowed to be equal to an intrinsic member's identifier.

AspectJ directly supports direct member introductions on the language level.

Indirect Introduction

Sometimes it is necessary to apply several introductions to certain objects from different perspectives. This means that there are several extrinsic characteristics that originate from different perspectives and should be combined to be applied to certain objects later on. Although it is known which perspectives are to be combined the definition of the objects that they are to affect should be deferred. An *indirect introduction* collects several extrinsic properties from different perspectives within one unit and defers the binding to existing objects.

The participants of this strategy are:

- *introduction container*: the unit that is used as the target for the introductions. The container contains the property definitions and the ancestor relationships.
- *introduction loader*: the aspect that introduces properties and ancestors to the container.
- *container connector*: the aspects that connects the container to application classes.

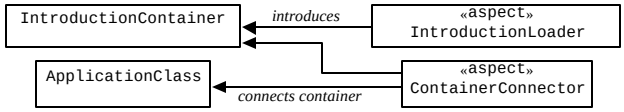

Figure 3: Indirect Introduction

The consequences of using an indirect member introduction are as follows:

- *reusable introductions*: the introductions are defined independent of the classes they influence and their application just consists of a container connection without the need to re-implement the introductions itself.
- *little aspect-related knowledge required at connection time:* the container connection does not need to know about introduction loaders.
- *member conflicts:* because the container connector does not know about the concrete introductions to the container there is some danger of possible conflicts between class members. The container connector cannot resolve conflicting introductions because the introducing aspects are transparent.

In AspectJ there are mainly two ways of implementing a indirect introduction. First, it is possible to introduce members and ancestors directly to an interface. In this case the ancestor introduction is limited. For example, it is not possible to introduce a class as an ancestor to an interface. Second, it is possible to introduce members and ancestors to each class that implements the container interface. Then the interface can be applied to classes by a direct ancestor introduction. The difference to the former implementation is that the container is not changed by the introduction loaders. Instead it is only used for identifying the classes that are to be affected by the introductions. Both approaches have in common that they make use of a direct ancestor introduction.

3. DYNAMIC CROSS-CUTTINGS

The previous sections describe strategies for adding attributes or ancestors that do not influence the behavior of applications. This can only be achieved by so called *dynamic cross-cuttings*. AspectJ provides two language constructs for dynamic cross-cutting: *advice* and *pointcuts*. Advice define the adapted behavior and pointcuts the places where advice crosscut existing structures. Like in the sections before, the names of the strategies are directly derived from the AspectJ terminology.

Direct Pointcut Connection

Sometimes is it desirable to adapt the existing behavior of certain objects well known to the developer. The behavior to be added is

extrinsic to such objects and it is not assumed that the behavior of any further objects not mentioned in this context should be amended in the same way. A *direct pointcut connection* directly influences the behavior of application objects.

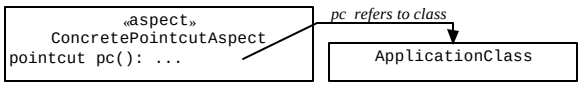

Figure 4: Direct Pointcut Connection

The strategy consists of the following participants:

- *concrete pointcut aspect*: the aspect that contains the behavior to be added and the definition of the situations when the additional behavior takes place.

The consequences of this strategy depend on its implementation:

- *no incremental modifications*: if the aspect itself includes concrete pointcuts (that are not inherited from a superaspect), there is no possibility to modify them incrementally (see [6] for a further discussion).

Direct pointcut connections are directly supported by AspectJ and correspond to the standard application of concrete pointcuts.

Indirect Pointcut Connection

An *indirect pointcut connection* defines a uniform way for adapting object behavior without naming the concrete objects.

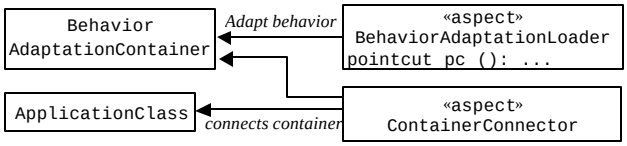

Figure 5: Indirect Pointcut Connection

The participants of this strategy are

- *behavior adaptation container*: the container that collects several behavior adaptations.
- *behavior adaptation loader*: the aspect that contains the new behavior and the description at what join points the new behavior should occur.
- *container connector*: the aspect that connects the behavior adaptation container to the application classes.

The consequences of using an indirect pointcut connections are:

- *typespecific cross-cuttings*: the dynamic cross-cutting code can be attached to arbitrary types. However, it is not possible to attain behavior-specific cross-cuttings.
- *few aspect-related knowledge required at connection time:* the pointcut definitions are transparent to the container connector. No information is needed about the behavior adaptation loaders to perform the connection.
- *aspect conflicts*: the developers that implement the behavior adaptation loaders must guarantee the consistency of the loaders. The container connector cannot detect or solve any consistency problems.

In AspectJ an indirect pointcut connection is achieved by defining (concrete) aspects with (concrete) pointcuts for a specific interface. Afterwards, this interface can directly be introduced to application classes.

Template Advice

A *template advice* separates the definition of behavior adaptation from the definition of how this behavior crosscuts a given structure. The crosscut is available as a hook for later specification, independent of the actual behavior. In that way, a template advice allows advice to be reused in different situations.

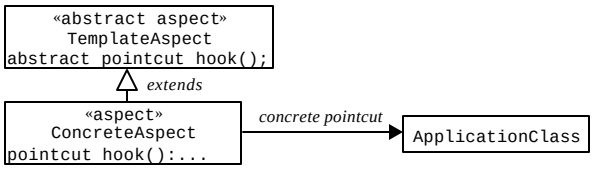

Figure 6: Template Advice

A template advice consists of:

- *template aspect*: the aspect that contains new behavior without specifying where this new behavior occurs. The behavior should take place at a certain hook pointcut.
- *concrete aspects*: the aspect that extends the template aspect and specifies the corresponding join points where the behavior should take place.

The consequences of applying a template aspect are:

- *pointcut independent aspect reuse*: it is possible to apply the behavior adaptation to situations that have not been foreseen at the time of aspect definition.
- *non-transparent pointcut*: in contrast to the indirect pointcut connection the developer responsible for connecting the dynamic cross-cutting code to an application has to know something (the hook pointcut) about the aspect to connect.

In a straight forward implementation of a template advice in AspectJ, the template aspect has to be abstract and the concrete aspect has to extend the concrete aspect. In [6], the application of the template advice in AspectJ is discussed in more detail.

Composite Pointcut

It is often observable that the way dynamic crosscutting occurs can be expressed by a combination of independently defined dynamic cross-cuttings. A *composite pointcut* separates a pointcut into two logically independent pointcuts.

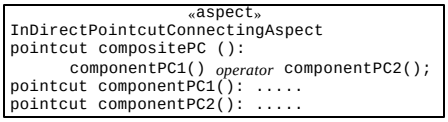

Figure 7: Composite Pointcut

A composite pointcut consists of the

- *component pointcuts*: the logically independent pointcuts.
- *composite pointcut*: the pointcut that combines several component pointcuts.

The consequences of using a composite pointcut are:

- *independent pointcut modification*: the logically independent component pointcuts can be modified without knowing the complete (composite) pointcut.
- *pointcut consistency*: the composite pointcut cannot guarantee the consistency of the pointcuts, so the developer must be aware of how to define the component pointcuts correctly.

In AspectJ a composite pointcut can be implemented by defining a pointcut that consists of a combination of pointcuts and does not use any pointcut designators on its own. Usually a composite pointcut is used in the context of a template advice.

4. EXAMPLE

In this section we analyze an implementation of the Observer pattern [5] in AspectJ similar to the one proposed in [2] by applying the strategies described above.

For example, we would like to provide graphical representations of application-specific objects that are automatically revised whenever the corresponding subjects changes. To support the Observer pattern, subjects must provide an interface for attaching and detaching observers. So the mechanism of member introduction in AspectJ can be used to equip classes with the (extrinsic) methods for attaching and detaching observers without actually changing the application's source code. The question is if a direct member or a indirect introduction should be used. Since the subject related methods can be used for a number of different classes (even though we are right now just interested in a few of them) an indirect introduction provides more flexibility. So we build the interface `Subject` (introduction container) that includes the methods `addObserver()` and `removeObserver()`. Moreover, we create an interface `Observer` that contains the method `update()` that should be invoked whenever a subject changes.

In order to allow an indirect introduction, we create the introduction loader `SubjectObserverProtocol` that introduces appropriate implementations to `Subject`:

```
aspect SubjectObserverProtocol {
    public Vector Subject.observers = new Vector();
    public void Subject.addObserver(Observer obs) {
        observers.addElement(obs);
        obs.setSubject(this);}
    public void Subject.removeObserver(Observer obs) {
        observers.removeElement(obs);}
}
```

Additionally, we are able to implement actions that should happen whenever a subject's state changes in this aspect: the `update()` method of every attached observer must be invoked. This code is part of the dynamic cross-cutting because it should be executed whenever the join points have been reached that immediately follow a change of the subject's state. However, it is hardly possible to define a consistent connection strategy for all possible subject classes in this case (cf. [3], [4], [6]). Therefore, we make use of the template advice that defers this decision. We regard it as a good idea to implement the advice in `SubjectObserver-Protocol` and thus we have to define the aspect abstract:

```
abstract aspect SubjectObserverProtocol {
  ...
  ... pointcut stateChanges(Subject s) ...
  after(Subject s): stateChanges(s) {
      for (int i = 0; i < s.observers.size(); i++)
        ((Observer) s.observers.elementAt(i)).update();
  }
}
```

We still have to decide how to implement the pointcut connection. Obviously, we are able to define that the observed target is of type `Subject`. However, we cannot decide what message receptions change a subject's state. Therefore the needed pointcut consists of a known part (target is of type `Subject`) and an unknown part. Therefore, we should use a composite pointcut.

```
abstract aspect SubjectObserverProtocol {
    ...
    abstract pointcut stateChanges();
    after(Subject s):
        target(Subject) && stateChanges(s) {...}
...}
```

The implementation in [2] does not use a *composite pointcut* and just uses an abstract pointcut. The result is that developers that want to apply the protocol have to guarantee that the pointcut parameter `s` refers to the right subject instance in their pointcut definition. Instead, the use of the composite pointcut already restricts developers to targets of type `Subject`, and therefore reduces errors when connecting the protocol to an application.

This example illustrates how the strategies for connecting aspects introduced above allow us to design a concrete high-level subject/observer protocol in AspectJ. Nevertheless, there are still some variation points of how the protocol can be applied to existing applications.

Whereas the usage of the indirect introduction needs `Subject` to be used as the extrinsic ancestor in a direct ancestor introduction, the actual implementation of the method `update()` in `Observer` is not fixed. It can either be added by a direct member introduction (the implementation in [2] uses this strategy), or by a simple Java implements relationship where the developer of the observer class is responsible for the definition. Furthermore, it is not prescribed how the concrete pointcut (`stateChanges()`) of `SubjectObserverProtocol` is connected to the application within the template advice. Usually, this is achieved by a direct pointcut connection.

5. Hanenberg: Strategies, no Patterns

In the previous sections, we have introduced recurring strategies that are used when developing AspectJ applications. I use the term strategy intentionally to delimit it from the term "pattern". In the following sections, I argue that these strategies are no patterns.

The main purpose of identifying these strategies was to find out what language features of AspectJ are usually used in what situations. Afterwards, we wanted to provide a catalogue of strategies that supports developers to decide what strategy to use in certain situations. Thereto, it is necessary to organize the strategies in a way that allows developers to easily identify them

and find out when and how to use them. Furthermore, developers must be aware of the consequences when using a certain strategy.

We organized the strategies in the following way. Every strategy has a *unique name*, a *description of its essence*, a *description of a situation* where it is typically used (skipped in this paper), a *description of the strategy's participants*, an *illustration of its form*, a *discussion of the consequences* of its application and a *discussion of how the strategy can be used* in AspectJ.

In this way, the strategies are organized similar to the GoF-design patterns [5]. Furthermore, it seems as if the benefit of the strategies is similar to that of design patterns: developers get a common vocabulary that eases their communication, and a catalogue that permits to decide when to use what strategies. In section 4 we have shown how those strategies can be applied in concrete scenarios. Nevertheless, there are differences between patterns and the strategies mentioned here.

The success of patterns is based upon a common understanding of object-oriented programming. All of the GoF design patterns are directly build on top of object-oriented constructs. Such a common understanding permits the problem and solution to be visualized effectively, by using standard object-oriented notations. Finally, the solution part of patterns can easily be understood by all object-oriented developers. Although the underlying programming languages may differ, developers are familiar with concepts like *object* or *message*. A similar situation is yet not given in the aspect-oriented community. Until now, there is no common understanding of aspect-oriented programming and therefore, no commonly accepted design notation.

The strategies have directly arisen from the usage of AspectJ, so they are the result of observing AspectJ code. This means that the strategies depend highly on the language. Although other aspect-oriented approaches like HyperJ permit to implement these strategies as well, the form of their implementation completely changes - aspects do not exists on the language level, and for example, no inheritance relationship between aspects can be used as is required in the template advice. As the form of the strategies differs between different aspect-oriented approaches, there is no reasonable usage of them. For example, a HyperJ programmer would not understand how the illustration of a strategy relates to the tool at hand.

Furthermore, it should be mentioned that the distinction between static and dynamic cross-cutting has directly arisen from AspectJ's terminology. However, there is a parallel between a template advice (dynamic crosscutting) and an indirect pointcut connection (static crosscutting). Both allow crosscutting code to be defined without specifying what locations the code should be woven into. So the distinction between static and dynamic crosscutting does not seem to be "natural". If we would only distinguish between crosscutting strategy (the way how something crosscuts various structures) and crosscutting code the only difference would be that code might affect code within a method (dynamic crosscutting) or code within a class (static crosscutting). There does not seem to be a strict necessity to provide different language features for both kinds of crosscutting.

Because of the language dependency it seems to be more appropriate to discuss the relationship between the strategies and *idioms* that are "low-level patterns specific to a programming language (…) which describe how to implement particular aspects of components or the relationships between them using the features of the given language." [1].

The second major reason why the strategies are no patterns is based on their "quality". The "quality without a name" describes the "life and wholeness" of a product. Patterns are constructs for generating this quality, so software generated by a suitable application of patterns should have the quality. However, our strategies were identified from an appropriate usage of the language constructs provided by AspectJ. In fact it cannot be definitively determined if the usage of those constructs has this quality because of the following reasons: there is not enough experience in the area of aspect-oriented programming to determine the quality of an (aspect-oriented) solution. It is not even determined, if the composition mechanism in AspectJ are good at all, even though they seem to solve known problems in object-oriented programming. To determine if some of these strategies are patterns, it is necessary to have a lot of experience in the area of aspect-oriented programming.

At least there seems to be a difference in the abstraction of the strategies in comparison to known patterns. The strategies concentrate on how to connect aspects with existing software - how extrinsic properties can be attached to objects. In this way, the problem space handled by those strategies seems to be directly derived from the typical problem of aspect-oriented programming on the implementation level.

So the overall impression is as follows. Although there are similarities between the strategies and patterns, they are not equal. I doubt that trying to bring the strategies to a corresponding pattern form would really result in new aspect-oriented patterns, since they are too dependent on the language AspectJ. Nevertheless, there seem to be strategies which are more interwoven with AspectJ (like composite pointcut) than others (like direct member introduction). From my point of view, it seems to be appropriate to compare new aspect-oriented technologies which appear from time to time with the strategies. This might improve a common understanding on aspect-oriented programming and improve the understanding on the mutuality of different aspect-oriented techniques.

6. Costanza: A First Step Towards AO Patterns

Before giving my position about the work introduced in this paper, I would like to recall the general idea of Patterns. At the present stage, there are mainly two views on what Patterns are all about. The one is to characterize Patterns as a literary form that is well-suited to communicate recurring problems and good solutions that resolve the forces of these problems. For this reason, a generally accepted "canonical form" has been established over the

past years. According to this form, each pattern consists of a name, a problem statement, a context, the forces that lie at the center of the problem, a solution, examples, the resulting context, a rationale (why the pattern works), related patterns and known uses [1]. Although the AspectJ strategies of this paper do not exactly match this form, they surely are very close. The only really missing elements are the forces, examples, the resulting context, related strategies and known uses. It is easily conceivable that examples can be drawn from introductory material, for example [2], and known uses from ongoing "real-world" projects that make use of AspectJ. The relation between certain strategies can already be seen to some extent in this paper – for example, the direct vs. indirect introduction are roughly used for the same purposes, with different tradeoffs (resulting contexts). It is equally conceivable to elaborate on the forces. An interesting case is the need to modularize crosscutting concerns which would be a force that all aspect-oriented strategies have in common. The fact that there might not be enough known uses for the strategies given here implies that they can only be regarded as "proto-patterns" [1], but on a more general level, from a "literary" point of view, they certainly qualify for being good examples of the pattern idea.

Another view on Patterns is the notion of achieving the so-called "Quality Without a Name" (QWAN). A "light-weight" paraphrase of this idea is the goal of making people feel more comfortable. For example, programs that employ object-oriented design patterns [5] make programmers more comfortable in changing their source code and, for example, adding new functionality. Again, the strategies of this paper qualify for having QWAN, at least in principle, because they also aim at easing the maintenance of software. Again, the lack of known uses, or other rationales, indicate that they can only be regarded as "proto-patterns" because their application in the "real world" might necessitate their modification in order to really achieve QWAN. However, this does not generally preclude their perception as patterns.

It is important to note that none of the views on Patterns presented here require them to be applied in object-oriented contexts only – the patterns from [5] just *happen* to be based on the building blocks of object-oriented programming, like composition, inheritance, overriding, and so on. However, many patterns also encompass other areas, for example other programming paradigms, as in hybrid languages like C++ and Lisp, up to methodological and organizational patterns that do not directly deal with programming at all [8]. So regardless of the view on patterns as a literary form or as a means to achieve QWAN, there is no reason at all to *not* apply them in an aspect-oriented setting. The only difference is that now aspect-oriented concepts are the building blocks, like pointcuts, introductions and advice.

So my conclusions are as follows. The strategies presented in this paper are a very valuable first step towards a catalogue of aspect-oriented patterns. Future steps include...

- ...elaboration of forces and known uses. These are the missing elements that probably require most of the work and investigation of existing projects.
- ...discovery of more advanced aspect-oriented patterns. The strategies of this paper are very elementary ingredients to aspect-oriented programming, but there will certainly arise more complex scenarios. For example, good candidates for solutions to be documented in pattern form are those that deal with feature interaction among different aspects.
- ...generalization of aspect-oriented patterns in order to be independent of a specific aspect-oriented approach. Apart from being more useful in different environments, such patterns would help to improve our understanding of the essence of the still emerging field of AOSD.

7. Summary

In this paper we have pointed out that the main focus of aspect-oriented software development lies in the connection between aspects and applications. We have described recurring strategies for connecting aspects and applications in AspectJ and we have illustrated how they can be used in a concrete example. Afterwards we have discussed to what extent these strategies can be regarded as patterns or not by giving two oppositional positions statements.

In conclusion, it is clear that developers who want to exploit the prominent features of aspect-oriented approaches need to gather good solutions and communicate them effectively. This paper provides strategies for AspectJ as good starting points and in doing so, hints to a feasible future practice of documentation for the aspect-oriented community, regardless of whether the proposed strategies will be perceived as patterns or not.

8. REFERENCES

[1] Appleton, B., Patterns and Software: Essential Concepts and Terminology, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[2] AspectJ-Team, The AspectJ Programming Guide, http://aspectj.org/doc/dist/progguide/

[3] Brichau, J., De Meuter, W., De Volder, K., Jumping Aspects, Workshop on Aspects and Dimensions of Concerns, ECOOP, 2000.

[4] Costanza, P., Vanishing Aspects, Workshop on Advanced Separation of Concerns, OOPSLA, 2000.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Addison-Wesley, 1995.

[6] Hanenberg, S., Unland, R. Using And Reusing Aspects in AspectJ. Workshop on Advanced Separation of Concerns, OOPSLA, 2001.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J. Aspect-Oriented Programming. Proceedings of ECOOP, 1997.

[8] Rising, L., The Pattern Almanac 2000, Addison Wesley, 2000.