Journal of Universal Computer Science, vol. 9, no. 8 (2003), 970-983
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

Aspect Weaving Strategies

Eduardo Kessler Piveta
(CEULP/ULBRA, Brazil
piveta@ulbra-to.br)

Luiz Carlos Zancanella
(Federal University of Santa Catarina - UFSC, Brazil
zancanella@inf.ufsc.br)

Abstract: We propose a model to support aspect-oriented programming in object-oriented languages, expressing general purpose aspects. To apply this model, the developer should implement the abstraction and composition mechanisms as well as one or more strategies defined in it. It could be applied to regular OO languages.
Key Words: aspect-oriented programming, object-oriented programming
Category: D.3.0, D.3.4

1 Introduction

Software engineering and programming languages exist in a mutual relationship support. The most used design processes break a system down into a set of small units. To implement these units, programming languages provide mechanisms to define abstractions to represent them and composition mechanisms in order to implement the desired behavior [Becker and Geihs, 1998].

A programming language coordinates well with a software design when the provided abstraction and composition mechanisms enable the developer to clearly express the design units. The most used abstraction mechanisms of languages (such as procedures, functions, objects, classes) are derived from the system functional decomposition and could be grouped into a generalized procedure model [Kiczales et al., 1997].

However, there are many properties that do not fit well into generalized procedures, such as: exception handling, real-time constraints, distribution and concurrency control. They are usually spread over into several system modules, affecting performance and/or semantics systematically.

When these properties are implemented using an object-oriented or a procedural language, their code is often tangled with the basic system functionality. It is hard to separate one concern from another, see or analyze them as single units of abstraction.

This code tangling is responsible by part of the complexity found in computer systems today. It increases the dependencies among the functional modules, deviating them from their original purposes, making them less reusable and error-prone.

Aspect-oriented programming allows separation of these crosscutting concerns, in a natural and clean way, using abstraction and composition mechanisms to produce executable code.

This paper defines abstractions to represent aspects and aspect-weaving strategies, helping in the task of weaving classes and aspects together. This work is related to AspectJ [Kiczales et al., 2001] and D [Lopes, 1997] in the requirements and theoretical foundation concerns. AspectJ offers mechanisms to define general purpose aspects and D provides implementation insights.

The Aurelia weaver [Piveta and Zancanella, 2001] implements the meta-class strategy described here, being influenced by JST [Seinturier, 1999] (that uses OpenJava's metaclasses to create aspects in the system), and by [Pryor and Bastán, 1999] (that implements a reflective architecture in SmallTalk to support aspect-oriented programming).

2 Aspect-Oriented Programming

Usual programming techniques are not expressive enough to implement in a satisfactory way the application requirements needed today. There are some characteristics that affect the system behavior, modifying the semantics and performance criteria in several points, both in systems static and dynamic structure [Lopes, 1997].

When these characteristics should be implemented, they are usually inserted into the system using arbitrary methods, distracting from what the original software component was supposed to do, making reuse and readability harder. This increases the complexity regarding to functional modules implementation, which become more confuse that they should be.

Having an important concern expressed in a localized manner (i.e. in an unique unit or code section) promotes an easier understandment of how the system is affected (because you do not have to look at several places neither separate it from others). Moreover, it is easier to analyze, modify, extend and reuse that concern [Czarnecki and Eisenecker, 2000].

This separation of concerns is a fundamental issue in software engineering and it is used in analysis, design and implementation of computer systems. However, the most used programming techniques do not always present themselves in a satisfactory way regarding to this separation.

This is mainly due to the fact that the most used approaches focus in finding and composing functional units, while other important concerns are not well addressed in the functional design [Becker and Geihs, 1998]. When two properties must be composed in different ways and still getting coordinated we say that one crosscut the other.

In [Kiczales et al., 1997] is defined that a system property that should be implemented could be seen as an aspect or as a component:

- the property could be seen as a component if it could be encapsulated in a generalized procedure (class, method or procedure) . Components tend to be units derived from the system functional decomposition. Examples: bank accounts, users, messages etc.

- aspects are not usually units derived from functional decomposition, they affect classes in a systematic way. Examples of aspects: concurrency control in bank account operations, account transactions recording, access security policy applied to systems users, real time constraints associated with messages delivery.

The aspect-oriented programming main goal is to help the developer in the task of clearly separate crosscutting concerns, using mechanisms to abstract and compose them to produce the desired system. The aspect-oriented programming extends other programming techniques (object oriented, structured, functional etc) that do not offer abstractions to deal with crosscutting [Kiczales et al., 1997].

An implementation based on the aspect oriented programming paradigm is usually composed of:

- a component language to program components (i.e. classes);

- one or more aspect languages to program aspects;

- an aspect weaver to compose the programs written in these languages;

- programs written in the components language;

- one ore more programs written in the aspect language.

2.1 Components

Components (in AOP) are abstractions provided by a language to implement systems basic functionality. Procedures, function, classes and objects are components in aspect-oriented programming. They are originated from functional decomposition. The language used to express componnents could be, for instance, an object-oriented, an imperative or a functional one[Tekinerdoğan and Akşit, 1998].

In this paper we are going to use *class* instead of *component* to avoid terminology mismatches between the concept in aspect-oriented programming and in component-based software development.

2.2 Aspects

Properties affecting several classes could not be well expressed using current notations and languages. They are usually expressed through code fragments that spread over the system classes [Czarnecki and Eisenecker, 2000].

Some concerns that are frequently aspects: concurrent objects synchronization [Dempsey and Cahill, 1997], distribution [Lopes, 1997], exception handling [Ossher and Tarr, 1998], coordination of multiple objects [Harrison and Ossher, 1993], persistence, serialization, replication, security, visualization, logging, tracing, load balance, fault tolerance amongst others.

2.3 Component language

The component language should provide developers with mechanisms to write programs implementing the basic requirements and also do not predict what is implemented in the aspects, (this property is called obliviousness [Kiczales et al., 1997], [Filman and Friedman, 2000]). Aspect-oriented programming is not limited to object orientation, although, the most used component languages are object oriented ones, such as: *Java*, *SmallTalk* or *C#*.

2.4 Aspect language

The aspect language defines mechanisms to implement crosscutting in a clear way, providing constructions describe the aspects semantics and behavior [Kiczales et al., 1997].

Some guidelines should be observed in the specification of an AO language: syntax should be related to the component language syntax (making easier the tools acceptance), the language should be designed to specify the aspect in a concise and compact way and the language grammar should have elements to allow composition of classes and aspects [Böllert, 1999].

2.5 Aspect Weaver

The aspect weaver main responsibility is to process aspect and component languages, in order to produce the desired operation. To do that, it is essential the *join-point* concept. A join-point is a well defined point in the execution or structure of a program. For instance, in object-oriented programs join-points could be method calling, constructor calling, field read/write operations etc.

The representation of those points could be generated in runtime using a reflective environment. In this case, the aspect language is implemented through meta-objects, activated at method invocations, using join-points and aspects information to weave the arguments [Mendhekar et al., 1997].

An aspect-oriented system design requires knowledge about what should be in classes and in aspects, as well as characteristics shared in both. Although aspect-oriented and object-oriented languages have different abstraction and composition mechanisms, they should use some common terms, allowing the weaver to compose the different programs.

The weaver parses aspect programs and collects information about the (join) points referenced by the program. Afterwards, it locates coordination points between the languages, weaving the code to implement what is specified in them [Böllert, 1998].

An example of a weaver implementation is a pre-processor that traverse the classes parsing tree, looking for joint-points and inserting sentences declared in the aspects. This weaving process could be of two types: static (compile time) or dynamic (load and runtime).

2.6 Implementation Technologies to Aspect-Oriented Programming

Two activides should be performed to support aspects: definition and implementation of abstractions to express aspects and implementation of a weaver to compose the languages.

2.6.1 Abstraction to Represent Aspects

There are two commonly used alternatives to define abstractions to express aspects, according to [Czarnecki and Eisenecker, 2000]: use conventional libraries, design a domain specific aspect language. The advantages on using domain specific languages are: declarative and explicit representation regarding to the implemented aspect, utilization of domain terms and that they could capture the developers intention in a more appropriate way. Using a specific domain language allows optimizations and error checking regarding to the language domain.

However, they are not always the best choice, due to the fact that users acceptance is harder that on using general purpose languages. When the aspects are not regarding to one domain, the advantages of using a specialized language are not so clear as in the cases that aspects are domain specific, such as in `D` [Lopes, 1997].

2.6.2 Aspect Weaver Implementation

Weaving aspects and classes involves program transformation techniques. This can be done using many technologies, but the most common approaches are: source code transformation and dynamic reflection.

The first could be implemented using a compiler or a pre-processor, that usually provide an interface to add and edit nodes generated in the parsing tree. Usually the woven code is reorganized in compile time, statically, efficiently.

Runtime reflection needs the explicit existence of mechanisms to interfering into running computations in the base level. The use of these meta-representations enable developers to modify the entities behavior in runtime [Czarnecki and Eisenecker, 2000].

3 A model to aspect-oriented programming

Here, we describe a model to support aspect-oriented programming by defining abstraction and composition mechanisms to offer support to express crosscutting concerns in a general purpose way: aspects according to the model are not domain specific.

This model focus on features derived from AspectJ and D. The first implements general purpose aspects and is the *de facto* standard for AOP in Java, and the second was choosen because of the theoretical foundation about weaving mechanisms.

3.1 Abstractions to represent aspects

The model is comprised by a set of class describing mechanisms to represent aspects in an application. Objects of these classes are referenced and used in the same way that others objects in the system, with the particularity that there are no explicit method calls to these objects. An aspect could be applied into different classes, acting in many systems events. When an aspect is inserted in the application, the developer associates it with the classes.

Each aspect has several pointcuts defining a set of situations when aspects are activated and advices that are executed in the occurrence of each situation. The join-points are defined using information about the classes and affected methods. In figure 1, we could see a class diagram of the model.

Each aspect is defined as an instance of a class called *Aspect* or one of its subclasses. In this context, there is the possibility to add an instance of the *Aspect* class to each property that the aspect crosscut. As each aspect could have different actions associated to it, it is divided into a set of pointcuts. A pointcut has information about the points affected by the aspect and the methods that should be invocated every time the join-points in the pointcut are reached.

A pointcut is an instance of the *PointCut* class, where the property *JoinPoints* is a list of classes and methods affected by the pointcut. In this model we could express join-points to execution of methods, constructors and destructors.

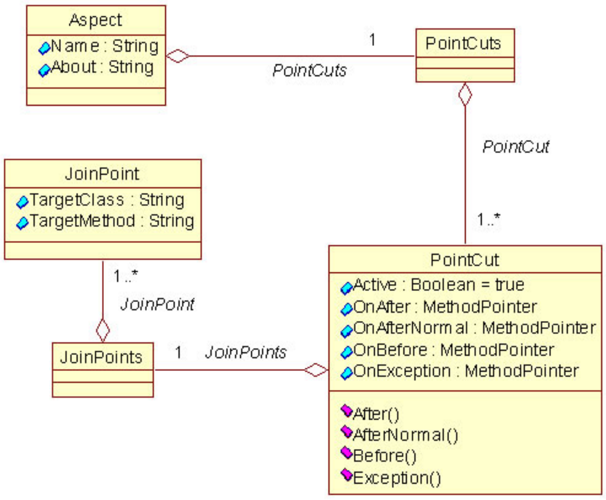

Figure 1: Class diagram of the model

Sentences expressed in aspects could be inserted in the following situations: before the joint-point occurence, after the join-point (in a normal method return), after a join-point (no matter the success of the method execution) and in an exception occurrence.

According to these characteristics, an aspect could have several pointcuts. Each join-point specify the *TargetClass* and *TargetMethod* fields, that are respectively the name of the class and the signature of the method affected by the aspect. Actions that should be executed every time a join-point is reached are specified together with pointcuts.

4 Weaving Strategies

The aspect weaver works adding new classes, new methods or modifying existing program abstractions. This work proposes four alternatives to map the structures defined as aspects in the system to a classes only structure.

4.1 Weaving through inheritance

The inheritance mechanism could be used to implement the behavior described in the aspects. If an aspect affects a class in one or more points, a subclass of the affected class should be generated to accomplish the modified behavior. The source code of the affected classes is not modified and the aspect code is inserted in the generated sub-classes.

In figure 2, there is a class called `Person`, being affected by the aspects `Trace` and `Synchronize`. The affected methods are overriden by the woven method, as stated in figure 3. The woven method encapsulates the inherited behavior, executing the sentences related to the aspects (figure 4).

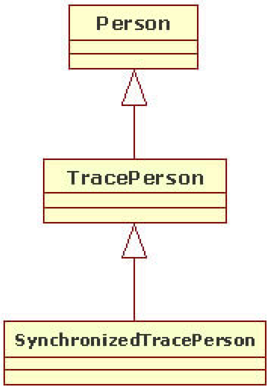

Figure 2: Class Person being affected by the Trace e Synchronize aspects

```
TTracePerson = class(TPerson)
   GetAge:integer; override;
   SetAge:(x:integer) ; override;
End;
```

Figure 3: Class created by the weaver

In order to the generated methods work properly, is necessary that the instances of `Person` be replaced by objects of the modified class (or explicit casts specified), in this case, `TTracePerson` class. If the aspects affect also the `Person` subclasses, the weaver should modify the subclasses referenced by the aspect. A similar approach is used by the AOP/ST weaver [Böllert, 1999]

4.1.1 Advantages

The weaving process is done in a non-invasive way. The original class is preserved, creating a subclass for each class affected by an aspect. However, the programs that use the affected classes should be modified in order to point to the right classes.

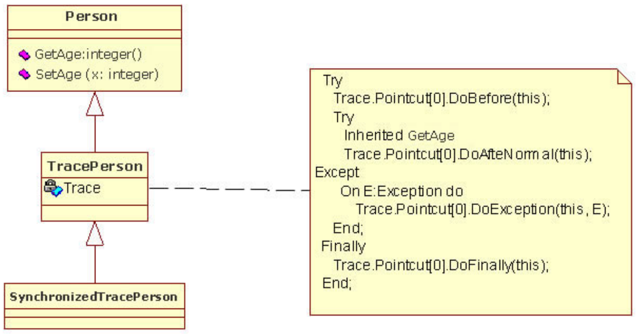

Figure 4: Method with the behavior modified by the Trace aspect

4.1.2 Related Problems

A problem related with this alternative is inheritance anomalies (the class is modified, but its subclasses still use the original class). Depending on the number of affected classes, as well as the number of aspects that affect the same class, the class hierarchy could become large. The classes created are not directly used but could consume an extra time due to searching time into the methods table.

If more than one aspect or more than a join-point of a single aspect act upon a class, there is the need to weave the system by steps, where a subclass is generated to the first aspect, another one to the second and so on. The problem is the definition of the weaving order. This order could by explicitly defined in the aspect.

4.2 Weaving through association

Association could be used to implement the behavior described in the aspects. For each affected class, a property (which type is `Aspect`) is created (figure 5). The affected methods are modified through calls to methods defined in the aspects (Figure 6)

4.2.1 Advantages

One of the advantages is that new classes are not created, neither objects of these classes are modified (just the affected classes are).

4.2.2 Related Problems

A problem with this approach is the modification of affected class, getting hard to find where aspects affect the class in the resulting code and making harder to get back to the original code.

```
TPerson = class
   Aspect: TAspect;
   GetAge:integer
   SetAge:(x:integer)
End;
```

Figure 5: Class modified by the weaver

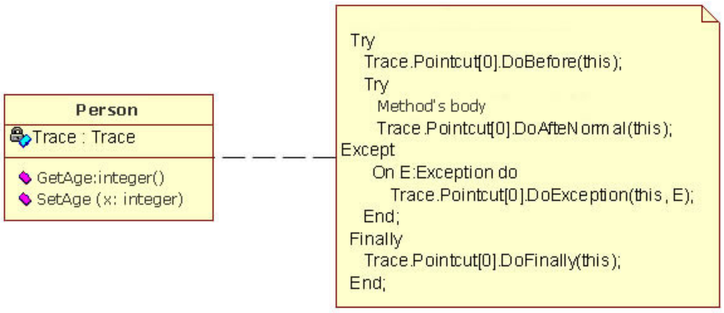

Figure 6: Modified class

4.3 Weaving through the Decorator Design Pattern

Another implementation alternative is using the Decorator design pattern. In this case, a subclass of the affected class is created together with a reference to the original class (figure 7). Using this subclass, called `Decorator` in the pattern, it is possible to create concrete decorators that implement the functionalities described in aspects. Use of this pattern is a flexible alternative to inheritance.

As an example we could see the `Person` class being affected by two aspects: `Trace` and `Synchronize`. A `Decorator` called `PersonDecorator` is created as a wrapper to the decorated object. Two more classes are created to implement the aspects functionality (figure 7).

Decorated objects must replace the Person instances. These objects are decorated by the `TraceDecorator` and `SynchronizeDecorator`. When implementing this strategy, aspect-ordering mechanisms should be defined to ensure that the behaviour is modified as desired.

4.3.1 Advantages

The advantage is mainly the flexibility inherent to the pattern: the order of decorators could be modified in runtime. It is also distinguished that changes are done in a non-invasive way.

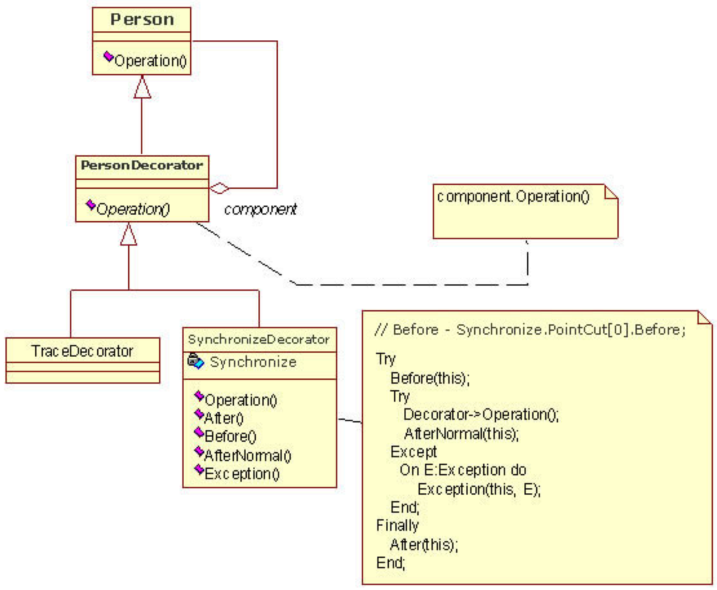

Figure 7: Person class being affected by the Trace and Synchronize aspects

4.3.2 Related Problems

One difficulty of this approach is that the identity of decorators is not the same as decorated objects.

The use of decorators to classes that are affected by more than one aspect, or in the cases that aspects could be removed/inserted dynamically is better than using the inheritance approach. However, there is not many advantages use decoration for classes affected by only one aspect.

4.4 Weaving through reflection

In this approach, a metaclass is created by the weaver for each class referenced by an aspect, providing message interception mechanisms, allowing the addition of sentences related to aspects in the parsing tree, using information about the base level (affected classes).

As an example, a class called `Person` that is affected by an aspect (Figure 8). For each affected class, a metaclass is generated, implementing the specified aspect semantics.

In [Lunau, 1998] a proposal to implement aspects as meta-objects is described through a mechanism of messages interception. The architecture originally

```
MetaPerson metaclass of Person

Method MetaPerson.InterceptMessages{
 try{
   Trace.Pointcut[0].DoBefore(this);
   try
     // Execute the base level code
     ExecuteBaseMethod(ParseTreeNode);
     Trace.Pointcut[0].DoAfterNormal(this);
   except
     On E:Exception do
       Trace.Pointcut[0].DoException(this,E);
 }
 finally{
   Trace.Pointcut[0].DoFinally(this);
 }
}
```

Figure 8: Metaclass with method interception

described was developed to allow computational reflection in control processes applications and allows to change meta-objects in runtime.

A reflective architecture is defined in [Pryor and Bastán, 1999] to support aspect-oriented programming in SmallTalk, using a metaclass aspects semantics representation. It's possible to add sentences before or after the original code. The weaver, in this strategy, is represented by the `AspectManager` class, responsible to intercept messages in the base level.

4.4.1 Advantages

One main advantages in this approach is the non-invasive addition/modification of classes. Another advantage is that using a meta-object protocol this approach is easier to implement than the others. Using interception and introspection mechanisms, part of the weaver responsibilities goes to the meta-object protocol.

4.4.2 Related Problems

One difficulty is the lack of commercial implementations using computational reflection concepts into object-oriented languages. Another question is that the addition of another abstraction level could affect performance.

5 Considerations

This work presents a representation to crosscutting concerns and a collection of strategies to the weaving process. Amongst these strategies, the better ones are the message interception and the decorator based strategy.

However, in simple contexts the inheritance and the association strategy could be used to implement aspects into object-oriented languages. To validate the message interception strategy, a tool called `Aurelia` was developed, using the Borland Delphi environment [Piveta and Zancanella, 2001] and a meta-object protocol.

The use of a reflective architecture as a basis to implement Aurelia contributed to the project result. The facilities provided by the meta-object protocol used, mainly due to the message interception mechanism, shorted the time to the systems implementation, as well as the system responsibility.

This model can be implemented in other programming environments, adapting the code generation module and the parser. Moreover, it's interesting to have mechanisms to perform structural or behavioral reflection, making easier to implement the strategies defined here.

6 Future Work

There are some proposals to continue this work, such as:

- Implementation in other programming languages and using other meta-object protocols.
- Definition and implementation of meta-object protocols to object-oriented languages;
- Implementation of all the strategies defined in the model and
- Development of a language to define aspects.

References

[Becker and Geihs, 1998] Becker, C. and Geihs, K. (1998). Quality of service - aspects of distributed programs. In *Int'l Workshop on Aspect Oriented Programming (ICSE 1998).*

[Böllert, 1998] Böllert, K. (1998). Aspect-oriented programming case study: System management application. In *Workshop on Aspect Oriented Programming (ECOOP 1998).*

[Böllert, 1999] Böllert, K. (1999). On weaving aspects. In *Workshop on Aspect Oriented Programming (ECOOP 1999)*

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston.

[Dempsey and Cahill, 1997] Dempsey, J. and Cahill, V. (1997). Aspects of system support for distributed computing. In *Workshop on Aspect Oriented Programming (ECOOP 1997)*.

[Filman and Friedman, 2000] Filman, R. E. and Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*.

[Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin. Springer-Verlag.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

[Lopes, 1997] Lopes, C. V. (1997). *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University.

[Lunau, 1998] Lunau, C. (1998). Is composition of metaobjects = aspect oriented programming. In [**?**].

[Mendhekar et al., 1997] Mendhekar, A., Kiczales, G., and Lamping, J. (1997). RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center.

[Ossher and Tarr, 1998] Ossher, H. and Tarr, P. (1998). Operation-level composition: A case in (join) point. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*.

[Piveta and Zancanella, 2001] Piveta, E. K. and Zancanella, L. C. (2001). Aurélia: Aspect oriented programming using a reflective approach. In *Workshop on Advanced Separation of Concerns (ECOOP 2001)*.

[Pryor and Bastán, 1999] Pryor, J. and Bastán, N. (1999). A reflective architecture for the support of aspect-oriented programming in Smalltalk. In *Workshop on Aspect Oriented Programming (ECOOP 1999)*.

[Seinturier, 1999] Seinturier, L. (1999). JST: An object synchronization aspect for Java. In *Workshop on Aspect Oriented Programming (ECOOP 1999)*.

[Tekinerdoğan and Akşit, 1998] Tekinerdoğan, B. and Akşit, M. (1998). Deriving design aspects from canonical models. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*.