ANALYZING MULTI-DIMENSIONAL PROGRAMMING IN AOP AND COMPOSITION FILTERS

Lodewijk M.J. Bergmans (lbergmans@acm.org)
Mehmet Aksit (aksit@cs.utwente.nl)
http://trese.cs.utwente.nl
TRESE GROUP – CENTRE FOR TELEMATICS AND INFORMATION TECHNOLOGY (CTIT)
UNIVERSITY OF TWENTE, P.O. BOX 217, 7500 AE, ENSCHEDE, THE NETHERLANDS

Abstract--In this paper, we investigate the meaning and impact of multi-dimensional construction of software, as exemplified by Aspect-Oriented Programming and Composition Filters. A simple application is introduced to illustrate the discussion topics. We conclude with a number of requirements for multi-dimensional programming.

Lately, the software engineering field is increasingly becoming aware of the need for better modeling techniques to be able to design systems as compositions of independent concerns. Various models such as Adaptive Programming [Mezini 98], Aspect-Oriented Programming [Kiczales 97], Subject-Oriented Programming [Harrison 93] and Composition Filters [Aksit 88, 92, 94] have triggered and/or addressed this need. In particular Aspect-Oriented Programming (AOP) is gaining much interest recently.

Previously, we have argued for extending AOP towards – more generic– mechanisms for *multi-dimensional decomposition* [Bergmans 96a]. The motivation behind this idea was that the concept of cross-cuttings or systemic aspects is not only restricted to the specific technical/operational aspects, such as synchronization, real-time constraints or caching, as emphasized by the AOP community. Such systemic aspects can also be found in the application domains that a system deals with. This is illustrated for example in [D'Hondt 99]. The Visitor design pattern [Gamma 95] is an example of a technique to add an aspect to a group of classes, while keeping its specification modular and in a separate place.

However, in order to deal with domain-specific aspects, AOP languages that only support a limited set of predefined dedicated aspects are insufficient. Instead, a general aspect language, supporting user-defined aspects, is needed. As a result, if we introduce the ability to define new aspects, the software engineer will introduce new dimensions for decomposing the system under consideration. Assume that we introduce aspects that deal with insurance or tax calculations in an administrative or logistic system. Let us also assume that domain experts or analysts find a way to separate insurance and tax calculations completely from the rest of the system specification. Then for each piece of functionality, the software engineer has to make an additional decomposition decision: define the functionality either on the base level (in this paper, we will assume the base level to be object-oriented) or in one of the appropriate aspects. In other words, this situation calls for a model that supports an arbitrary number of decomposition dimensions.

In this paper, we investigate the meaning and impact of multi-dimensional construction of software with a few modeling approaches, notably aspect-oriented programming and composition filters.

1. MODELING SOFTWARE

From a software-engineering point of view, a key requirement of software development is to be able to construct a system by combining modules that have minimal dependencies between each other. When discussing components and modules, we will focus upon the source-code level, since many of the challenges in software development are related to the construction, adaptation and reuse of specifications (i.e. source code). In addition, there is a natural desire to have a direct mapping between source-code components and run-time components, as to avoid a large conceptual gap between specification and run-time entities.

In order to optimize the design, construction and maintenance process of software, we need to apply the divide-and-conquer principle by *decomposing* our software into smaller parts, that can be decomposed again, recursively. This decomposition process must continue until a level is reached where each unit (a) can be understood and constructed effectively, and (b) deals only with *a single concern*[1].

The decomposition process is a problem-solving technique, but it provides at the same time a basis for system construction and maintenance by providing the building blocks for putting together a system.

1 Actually, the notion of 'a single concern' is undefined, because we can always decompose until we reach the finest possible specification level. The importance of the notion of a single concern is to find the right abstractions in our problem and solution domains such that they can be decomposed into more fine-grained parts without violating the existing structure and interfaces (in other words: such that further decomposition is structure-preserving).

As a basis for discussing software construction in general, and more specifically composition and decomposition of software, we introduce a number of definitions:

- The *software product* that is the result of the development process consists of a (usually large) collection of *code fragments*; we adopt the notion of code fragments as the most fine-grained independent specifications that can be merged into a running system. Each code fragment has its own identity and –most important– deals with a single concern only.
- A *use case* is the (description of) behavior for a typical usage of a system or product in a way that is non-specific for data values. A use case can be implemented by a number of code fragments. The execution (instantiation) of a use case consists of the ordered[1] execution of a set of code fragments.
- The final code of a software product consists of a set of code fragments that is organized in such a way that the complete set of use cases defined for that product can be executed, while satisfying the (operational) quality requirements of the use case/product.

According to the above definitions, it is possible to develop software systems by collecting all the code fragments that are needed to implement all the use cases that are required for a particular software product. In general there will be overlap between use cases, which allows for sharing the same code fragment between multiple use cases.

We can visually represent a software product as follows:

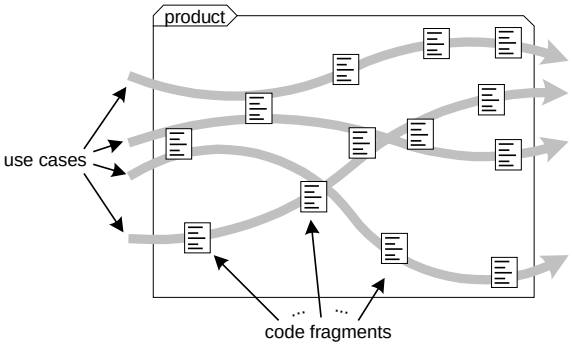

Figure 1. Modeling software as a collection of fragments.

From the run-time perspective, this model of software products is satisfactory. However, assuming a very large collection of code fragments, it becomes impossible to manage the complexity of such a system. As a result, we must establish additional structure upon the code fragments. This is achieved by decomposing the system into smaller parts, which can be decomposed again, repeatedly, until the level of code fragments is reached: a divide and conquer strategy in order to manage system complexity.

- A *decomposition* is a particular (usually hierarchical) ordering upon design entities, such as code fragments. If there are several orderings upon the same group of entities, we refer to each ordering as a *decomposition dimension*.

The goal of this paper is to investigate approaches that go beyond the traditional decomposition dimensions of the object-oriented model.

2. EXAMPLE APPLICATION

To understand and compare the various approaches, we will use one common example. The example is based on the following conceptual model, and should be thought of as a part of an application that manages real-time (audio or video) streams:

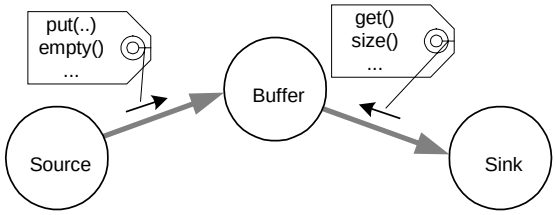

Figure 2. Conceptual model for our example application

The idea is that the *Source* represents an input device (anything from audio or video equipment to a communication channel to another application, as long as they repeatedly provide data that together form e.g. an audio or video stream). The *Sink* in this case is anything from an application to an output device that needs the data as a series of packets at regular intervals. The *Buffer* is used to deal with the timing differences between the input an output of this little system.

This application has to deal with the following concerns:

- *Synchronization*: the *Buffer* has only a limited size, which means that the *Source* should no longer add extra entries when the buffer is full (i.e. until the *Sink* has removed one or more entries).
- *Logging*: for testing purposes, all relevant information is to be logged, e.g. when a packet is handled by each part of the system, how it is synchronized, and how much memory is consumed by the various parts of the application. Logging is blocked when *Buffer* reaches its top capacity (e.g. at 95%), since this is a sign of system overload.
- *Memory Management*: for optimizing the usage of the limited memory available, each of the three parts in the system may dynamically claim extra memory. However, the following rules apply:
  1. The maximum amount of memory claimed by these three parts together is limited (say, to M_{max}). This amount may vary dynamically as well.
  2. The maximum amount of memory available for each part should be limited according to the following scheme: $M_{Source}:M_{Buffer}:M_{Sink}:M_{max} = 4 : 8 : 1 : 13$.

  The effect of these rules is that the possibility to claim more memory in each part depends on the variable M_{max} and the memory usage of the other parts at that time.

We show how these concerns and the concepts are related in a single table:

[1] Not necessarily sequential, and possibly interleaved!

associated: (maps to)	Source	Buffer	Sink	Sync.	Logging	Mem.Man.
Source	-	Y				
Buffer		-				
Sink		Y	-			
Synchr.	i	Y	i	-	Y	
Logging	Y	Y	Y	Y	-	Y
Mem.Man.	Y	Y	Y		Y	-

Legend: the direction of mapping onto is from row to column. "Y" is applied to, "i" indirectly applied to, "-" irrelevant

We can summarize that logging and memory management are cross-cutting concerns, whereas synchronization can be concentrated mostly at the buffer. It is also important to note the dependency between concerns: logging requires synchronization, and depends on the memory management concern.

It is natural to map the concepts in the above problem description to classes in various object-based approaches, and we will assume so unless denoted otherwise.

3. DECOMPOSITION IN THE OO MODEL

In the design of an object-oriented system, we can distinguish the following decomposition activities (not necessarily performed in the order presented here):

1. The problem domain is decomposed into independent concepts represented by classes.
2. The behavior of classes is decomposed into a set of methods.
   a) Each method consists of one or more code fragments.
   b) The state of –instances from– classes is represented by a set of instance variables.
3. In order to deal with the large amount of classes, the class-space is decomposed again ([Booch 94] discusses this 'canonical model of complex systems'):
   a) With *is-a* or *inheritance* relations.
   b) With *part-of* relations, that designate aggregation at the instance level.

This leads to a three-level decomposition structure as illustrated by the following figure:

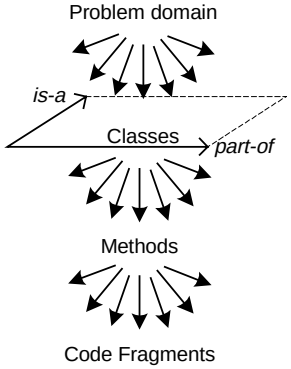

Figure 3. Decomposition from problem to code fragments

The most interesting decomposition level is the first, which decomposes from problem domain concepts to classes: this decomposition step is refined through *is-a* and *part-of* decompositions.

When designing the example application in an OO manner one would like to retain all concerns fully separated (as first-class entities), for instance for the purpose of maintainability or reuse. The following class diagram explicitly shows all specific concerns as independent entities, without optimizing the design for code reuse:

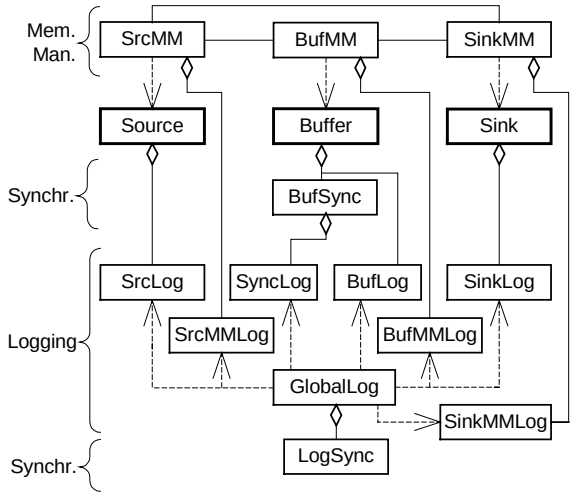

Figure 4. This class diagram shows the object-oriented relations between the concerns that are involved[1].

The point of this diagram is to illustrate that even with so few concerns involved, a combinatorial explosion will occur if a designer wants to keep these concerns fully separated: As shown in Figure 4, for the *Source*, *Buffer* and *Sink* concepts it is necessary to define a corresponding Log concept, resulting in the declaration of multiple Log classes, such as *SrcLog*, *BufLog*, etc. In combination with logging of the status of synchronization of the *Buffer* class also a class *SyncLog* is defined. Class *LogSync* contains the synchronization of the global log class. Further remarks about this diagram:

- Note that separating these concerns in independent classes –or even separate methods– may not be possible to implement at all (e.g. synchronization).
- If there is a lot of overlap between all logging code or all memory management code, it could be separated into a single class. Reusing this code can be done through inheritance or part-of relations.
- Further optimization of the design will consist of replacing some of the classes with methods or even code fragments within other methods; but such steps compromise the modularity of the design!

We can conclude that the object-oriented model cannot cope very well with the modular specification of independent concerns.

4. DECOMPOSITION WITH ASPECTS

Aspect Oriented Programming (AOP) in the general sense is based on –separately– specifying aspects to enhance a design

[1] Note that this diagram does not intend to show the best possible design for this application; a specifically optimized design would probably involve use of inheritance and design patterns such as Decorator and Observer.

or software system (existing or under development). Aspects are defined as the features that crosscut other components in a systemic way. This means that other components (these are typically objects, although not necessarily) are –possibly– affected throughout the entire system. In AOP, this impact is restricted to the so-called 'base level'; this is best compared to the functional part of the system. Typical examples of aspects include memory management, synchronization, tracing and logging, and so on.

The main idea behind AOP is to be able to specify dedicated semantics, which can be seen as enhancements to the basic functionality of a program, in a separate module. A dedicated aspect language that is optimally suited to expressing the specific aspect can be adopted for this purpose. The aspect specifications are to be merged ('woven') with the base level program through a so-called AspectWeaver™.

Figure 5 illustrates how the example application would be modeled in an AOP-way. A clear distinction is made between the (base level) classes *Source*, *Buffer* and *Sink*, and the aspects *Synchronization*, *Logging* and *Memory Management*. The resulting diagram shows that in this example, with the limited number of classes involved, indeed most aspects crosscut all components. Only for the synchronization aspect, this is less obvious: all synchronization is centered on the *Buffer* class, but this may involve those classes that are interacting with the *Buffer* class as well (indicated by open bullets in the diagram).

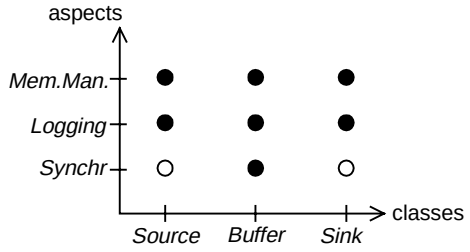

Figure 5. Mapping between classes and aspects

The figure clearly illustrates that the mapping between classes and aspects is of *n*:*m* nature; each class may have multiple aspects, and each aspect may apply to multiple classes.

Currently, the *AspectJ* language [Lopes 98, AspectJ 99] is the trend-setting approach to AOP. AspectJ brings the principle of AOP to Java. It adopts the following design decisions:

- AspectJ adopts no dedicated aspect languages, but uses plain Java code (method and variable declarations) for specifying aspect semantics. Dedicated syntax is introduced for specifying join points (i.e. those points in the base level code where the aspect code must be inserted) and weaving semantics.
- Rather than intricate code weaving of aspect specifications with base level code at the statement level, in AspectJ the aspect code is woven as method pre- and postfixes and new declarations of methods and variables within classes.

In other words, the key contribution of AspectJ is to add a new kind of composition-semantics to Java; based on the concept of aspect composition, AspectJ enables the merging of method bodies in a sequential manner.

The AOP approach applied to the object-oriented model can be visualized by the following decomposition dimensions:

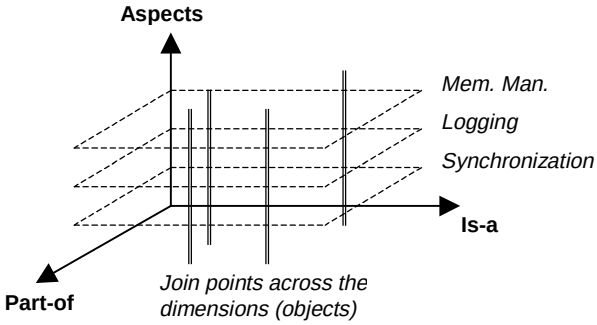

Figure 6. 'Standard' AOP applied to the OO model.

The figure shows that the aspects form one additional dimension: in this dimension each aspect can be mapped onto a set of join points that can be located anywhere within the base level program[1]. This also illustrates that aspects can not apply to the specifications of other aspects; they are confined to be enhancements of the base level program. This has at least the following consequences:

- Composability of aspects with each other is undefined; they are completely independent, and cannot interface with each other. However, this does not mean that they cannot *interfere* with each other. Depending on the semantics of the aspect specification language, independently specified aspects may or may not interfere. In the case of AspectJ, where the aspect language has the full expression power of Java, this means that the introduction or modification of aspect specifications may –or may not– cause a program to break or behave in unforeseeable ways.
- The fact that aspect specifications cannot apply to other aspects, may be a substantial limitation. This is visible for instance in our example application: one of the requirements is that logging may show the status of other aspects as well, and logging is blocked (i.e. has a synchronization constraint itself) when the buffer is full[2].

We can conclude that AOP can solve a range of problems in (object-oriented) programming by introducing an additional decomposition dimension. However, if multiple concerns are addressed within this single dimension, they cannot be applied to each other. Unless aspects are restricted to well-defined and orthogonal operational characteristics of the base level program, this is insufficient.

5. DECOMPOSITION WITH FILTERS

Composition Filters (see e.g. [Aksit 88, 92, 94] & [Bergmans 96b]) offer a form of aspect programming coming from a different perspective; the notion of independent, self-sufficient objects. To address this, composition filters allow for specifying and composing aspect specifications such as object behavior, synchronization constraints, multiple views and

[1] This figure does not show any decomposition dimensions within classes (since the visualization of more than 3 dimensions is problematic).

[2] It is not entirely clear whether this is fundamental to AOP or not, this seems to be open for discussion.

object interaction abstractions. Each filter specifies one aspect, and is associated with a class. Important characteristics of this approach are:

- One single, consistent, specification technique that is shared by all types of aspect specification (i.e. 'filters').
- An open-ended set of aspect types (filter types).
- All aspect specifications are composable with each other.
- Straightforward aspect specification reuse between classes.

Due to lack of space, we refer to e.g. [Bergmans 94] or [Koopmans 95] for detailed explanations of composition filters.

In this approach, crosscutting aspects are dealt with by importing the relevant class and/or filter specifications wherever necessary, e.g. through inheritance (as specified in a Dispatch filter). Figure 7 shows how aspects can be mapped to objects in the composition-filters approach:

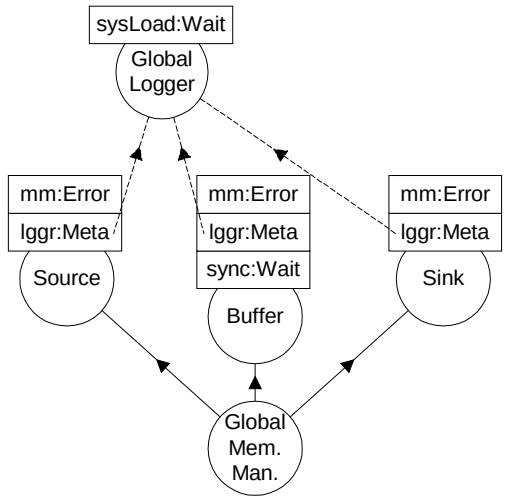

Figure 7. A CF-based approach to the example application.

The figure shows that each class is adorned with one or more filters. In this example three commonly used filter types are used: a *Wait* filter defines synchronization constraints upon messages, an *Error* filter triggers exceptions, and a *Meta* filter performs message reflection and lets each message be manipulated by a user-defined *abstract communication type* [Aksit 93] (in this case *GlobalLogger*). Filter reuse between classes is possible but is not shown here. Class *GlobalLogger* is of particular interest since it (partially) implements the logging aspect, but to do so requires another aspect; synchronization, which is defined by the *sysLoad* filter of the *Wait* type. All filter processing may involve information about the messages and the state of the objects.

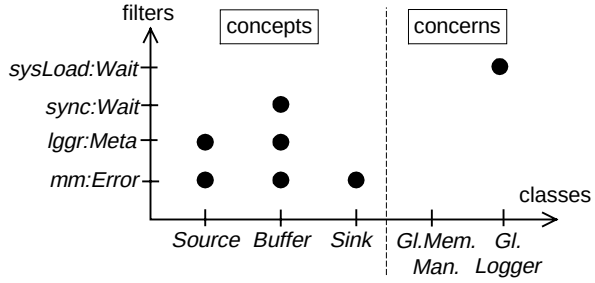

Figure 8. Mapping between classes and filters

Figure 8 above shows the mapping between filters and classes in this application. In the composition filters approach, concerns can be represented either by classes or by filters. If a concern is represented by a class, such as the *GlobalLogger* class, it is possible to define 'a concern of a concern', as is the case here with the synchronization by the *sysLoad* filter.

The disadvantage of expressing concerns in classes (apart from losing many of the benefits that usage of filters has) is that the full expressiveness that is available for defining this concern reduces the guarantees for composability of different concerns. This problem is partially addressed by enforcing certain type constraints (i.e. *abstract communication types*), and partially because each object explicitly specifies all its concerns, making it easier to verify potential composability problems.

One important design decision for the composition-filters model was –following the basic object-oriented philosophy– to make each class self-contained; only explicitly imported features extend the behavior of the class. This has many advantages from a software engineering perspective, but for systemic aspects, this requires either explicitly importing them in all classes in a system, or incorporating them through inheritance of a common ancestor.

Apart from the lack of a mechanism for defining systemic properties, the composition filters approach can address all requirements of the example application. It must be observed, however, that it is not fully multi-dimensional in the sense that concerns can only be defined upon classes, but not upon filter specifications.

6. CONCLUSION

From the previous discussion and example, we can conclude that multi-dimensional construction of systems is indeed needed, and not trivial to achieve. For our example application this can be graphically illustrated as follows:

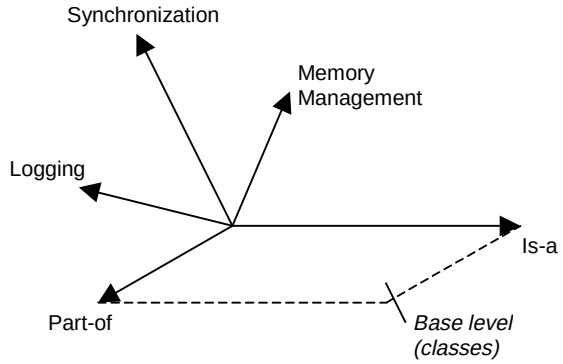

Figure 9. A base level (class) space plus additional decomposition dimensions for three concerns

One of the main themes of this paper is that the entities in one dimension may cross-cut with the entities in other dimensions (e.g. synchronization is needed during logging). It is open to discussion whether we still can –and should– talk about a single base level in such a system, but unless this proves to be problematic, we feel this is at least needed for practical reasons.

In section 4 we observed that AOP introduces one extra decomposition dimension for defining aspects upon the base

level. We discussed under which circumstances this restriction becomes apparent, and observed the lack of composability for generic aspects. In the previous section we evaluated the Composition Filters approach with respect to the example application and the ability to perform multi-dimensional decomposition.

We can make the following observations about the requirements for multi-dimensional decomposition:

- We need a generic language to express aspects so that application specific aspects can be defined as well (this is exemplified by AspectJ).
- A general aspect composition mechanism is required that supports the composition of aspects from multiple dimensions. This probably requires a certain composition expression (such as join point specifications in AspectJ and filter expressions).
- Since some dimensions may need to influence each other, it is necessary to provide general abstractions as a means to share information among multiple aspects. For example, the composition filters model supports the notion of *conditions* (abstractions of the internal state of an object) and an *object manager* (which deals with issues such as message queues and scheduling).

Our future work comprises the investigation of other modeling approaches such as Hyperspaces [Tarr 99] and Adaptive Programming [Mezini 98], in particular with respect to their composability properties. We will also look into extending the composition filters model to be able to express systemic aspects and possibly other support for the construction of multi-dimensional systems. Finally, the integration with our current research on balancing among the many dimensions of design criteria is an important item on our research agenda.

REFERENCES

[Aksit 88] M. Aksit & A. Tripathi. *Data Abstraction Mechanisms in Sina/ST*, Proc. of the OOPSLA '88 Conference, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 265-275

[Aksit 92] M. Aksit, L. Bergmans and S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, Proc. of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395

[Aksit 93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa. *Abstracting Object-Interactions Using Composition-Filters*, In *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS 791, Springer-Verlag, 1993, pp 152-184

[Aksit 94] M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans. *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, Proc. of ECOOP '94, LNCS 821, Springer Verlag, July 1994, pp. 386-407

[AspectJ 99] *AspectJ Language Specification*, XEROX Corporation, URL: http://www.aspectj.org, 1999

[Bergmans 94] L. Bergmans. Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands, 1994

[Bergmans 96a] L. Bergmans, *Aspects of AOP: Scalability and application to domain modelling*, position paper for the first 'Friends of AOP' workshop, XEROX PARC, Palo Alto, 1996

[Bergmans 96b] L. Bergmans & M. Aksit, *Composing Synchronization and Real-Time Constraints*, Journal of Parallel and Distributed Programming, September 1996

[Booch 94] G. Booch, *Object-Oriented Analysis & Design-with Applications*, 2nd edition, Benjamin/Cummings Publishing Company, 1994

[D'Hondt 99] M. D'Hondt & Th. D'Hondt, *Is Domain-Knowledge an Aspect?*, position paper for the ECOOP'99 Workshop on Aspect-Oriented Programming, to be published in Springer-Verlag ECOOP workshop proceedings, 1999

[Gamma 95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Harrison 93] W. Harrison & H. Ossher. *Subject-oriented programming (a critique of pure objects).* In proceedings of OOPSLA '93, September 1993.

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997.

[Koopmans 95] P. Koopmans. *On the Definition and Implementation of the Sina/st Language*, M.Sc. Thesis, University of Twente, The Netherlands, July 1995

[Lopes 98] C.V. Lopes & G. Kiczales, *Recent Developments in AspectJ*, in *Object-Oriented Technology-ECOOP'98 Workshop Reader*, position paper for the Workshop on Aspect-Oriented Programming, pp. 398-401, LNCS 1543, 1998

[Mezini 98] M. Mezini & K. Lieberherr, *Adaptive Plug-and-Play Components for Evolutionary Software Development*. OOPSLA '98, October 1998.

[Tarr 99] P. Tarr, H. Ossher, W. Harrison & S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In proceedings of ICSE 21, May, 1999.