

# Aspects should not die

*Frank Matthijs, Wouter Joosen, Bart Vanhaute,  
Bert Robben, Pierre Verbaeten*

*Report CW 266, April 1997.*



Katholieke Universiteit Leuven

Department of Computer Science

Celestijnenlaan 200A - B-3001 Heverlee (Belgium)

# Aspects should not die

*Frank Matthijs, Wouter Joosen, Bart Vanhaute,  
Bert Robben, Pierre Verbaeten*

*Report CW 266, April 1997.*

Department of Computer Science, K.U.Leuven

## **Abstract**

*In this report we relate current state-of-the-art aspect-oriented programming (aop) to meta-object programming and to more traditional decomposition approaches. This comparison is based on our own experience in the Correlate project. We also discuss the way aspects may appear during the software development process. We then give our view on why in tomorrow's aop aspects should not always die at weave-time but remain run-time entities during execution of the program.*

**Keywords :** aspect oriented programming, Correlate, run-time aspects

# Aspects should not die

Frank Matthijs, Wouter Joosen<sup>+</sup>, Bart Vanhaute, Bert Robben<sup>\*</sup>, Pierre Verbaeten  
 Dept. of Computer Science - K.U.Leuven  
 Celestijnenlaan 200A B-3001 LEUVEN BELGIUM

<sup>\*</sup> Research Assistant for the Belgian National Fund for Scientific Research

<sup>+</sup> Researcher for the Flemish I.W.T.

## Abstract

*In this report we relate current state-of-the-art aspect-oriented programming (aop) to meta-object programming and to more traditional decomposition approaches. This comparison is based on our own experience in the Correlate project. We also discuss the way aspects may appear during the software development process. We then give our view on why in tomorrow's aop aspects should not always die at weave-time but remain run-time entities during execution of the program.*

## 1. Introduction

A well known engineering discipline is to abstract and to decompose a complex system/application into manageable chunks. Functional decomposition does this by identifying chunks that have a well-defined function in the problem domain. Object-oriented programming has been quite successful for building complex systems this way. A major drawback of state-of-the-art technology however, is the fact that such a decomposition leads to inefficient execution, because the units of decomposition don't necessarily align with aspects such as memory management, coordination, networking, real-time constraints, etc.

One way of attacking this problem could be based on identifying deliverables associated with the development project, and describing them as the responsibilities for the development team. Such responsibilities could be related to performance of an application, security aspects, etc. Responsibilities for the developers will not necessarily map onto individual entities (objects) in the system. We believe that this is the point where aspects come in.

Informally speaking, aspects can be defined as ingredients of an application that describe key issues that are related either to the essential semantics of the application, or to the acceptable (read: performant) execution of the application. A fundamental difference between functional decomposition and aspectual decomposition is that the former tries to partition the problem into more manageable parts, whereas the latter is trying to describe the same problem from different, yet possibly mutually influential perspectives. Ultimately, an aspect-based program simply becomes a collection of aspects.

In an attempt to further refine this definition, we must answer questions regarding the nature of aspects, the way they are used, and how they should be specified.

1. Are all aspects equally important within the context of a single application (major/major view), or do they modify the behaviour of base entities (e.g. groups of objects) in the traditional sense (major/minor view)? When all aspects are viewed as equal, the programmer is encouraged to think in terms of aspectual decomposition from the very beginning.
2. What is the domain of aspects? Do they typically deal with run-time properties such as performance enhancement, or also with elements of a problem domain? Which properties? The range of possible domains in aspect-oriented programming (aop) can certainly be much broader in comparison to the related paradigms.
3. What is the appropriate level of abstraction to describe aspects? Clearly, there is a need for some aspect description language. This could be the same the language as the one for computations (similar to the concept of reflective languages), or it could be separate languages, better suited for aspect descriptions.
4. How do aspects show up at different stages of the development cycle? Are aspects identified as separate entities

during modelling in the analysis/design stage? Are they described separately in the program at high-level implementation and are aspects still visible in the generated code? On which levels are aspects orthogonal, on which levels are they manifest? In a traditional modelling process, aspect-based thinking has always been present implicitly. Early efforts manually wove the aspects together; there was no explicit notion of aspects at the programming level, and of course neither in the resulting executable. An advantage of the manual approach is that, given enough manpower, even heavily interfering aspects can be woven together. We believe that the focus of current research is at making the aspects manifest in every stage of the development.

We will use the above mentioned issues as a starting point not only for a tentative comparison of aop with current and previous related work, but also as a basis for identifying possible evolutions and properties that are missing in current (first generation) aop systems: for example, we feel that aspects (or at least some of them) must survive in the executable (at run-time) if dynamic behaviour is to be supported.

We hope that feedback on these topics will lead to our better understanding of aspect oriented programming in general, and we think it can put a few questions into perspective about where we are and where we are going with aop.

This report is structured as follows: in section 2, we will try to relate our current and previous work to the issues raised above. Section 3 discusses a few possible evolutions and properties that we feel are missing in current aop systems. We summarize in section 4.

## 2. Aop and friends

Current and previous work in the areas of meta-object protocols, reflection and open implementation typically recognizes the need for a decomposition that is more aspect-related than functional [Aksit] [Kiczales] [Yokote]. A tentative comparison can help clarify the similarities and the differences of both approaches (the aop approach and the reflection, meta-object protocols and open implementation approach, which we call the aop-related approach). In this section, we present our view on current and previous work in these domains, taking our own Correlate project as an example. By doing so, we try to catalogue the aop-related efforts along the lines of the questions presented in the introduction. Each subsection also contains a brief comparison with aop.

### 2.1. Importance of aspects

In Correlate, we emphasize the distinction (a.o.) between what we call the *computational* view, where the base computation is described, the *physical* view, where aspects such as location control, security, routing etc. are expressed, and the *temporal* view, where scheduling and real-time aspects can be specified. Our separation into different views stems from a fundamental unhappiness with the infamous tangled code phenomenon, which probably drives all activities in this area. A characteristic of this and similar approaches is that a base aspect captures the main computation (“what to do”, computational view) and a number of meta-level components describe run-time properties such as memory management, load balancing, network communication, etc. (“how to do it”, physical & temporal view). This is a manifestation of what we call the major-minor view. See figure 1 for an illustration.

In contrast, aop tries to describe the problem from different perspectives, without attributing more importance to one of them. Some aspects are more application domain oriented, others cover run-time properties, but an aspect-based program is simply a collection of aspects, without one of them necessarily being prominent. We call this the major/major view.

### 2.2. Domain of aspects

In Correlate we address issues like location control, synchronization, communication and fault tolerance, separated from the main computation [vOeyen]. Given our background in distributed systems and networking, it is no surprise that we mainly focus on distribution and concurrency issues. While we have identified scheduling and real-time aspects as another point of attention, stepping back it is interesting to see that we use a separate view (the temporal view) to characterize and describe them, while we situate all other aspects in one view (the physical view). The trend here is an attempt to decompose the application into views, which are groups of related aspects. Like many

related activities in this area, however, we don't explicitly decompose the problem into all sorts of aspects, only the most immediately relevant aspects for the problem domains we target (i.e. distributed systems).

Aop explicitly encourages the programmer to see a problem as a collection of different issues, and to try to find all issues that are relevant to the problem, not just some of them.

### 2.3. Aspects languages

What is the appropriate level of abstraction to describe aspects? In Correlate, as in other reflective languages, aspects are described in the same language as the main computation. An advantage is that the programmer can use a familiar language to program at the meta-level. A disadvantage is that the base language may not be the most appropriate for the job at hand.

Aop starts from a different viewpoint, and stresses the importance of each and every aspect. Therefore, the choice of a specific language for each aspect comes as a natural consequence. However, we think that the current generation of aspect description languages involves programming at a fairly low abstraction level: it is some kind of medium-level programming language. In our opinion, a more natural fit for aop are higher level, more declarative descriptions of aspects.

### 2.4. Manifestation of aspects

A last interesting difference that deserves more attention, is the stages in the development process where aspects are manifest and orthogonal. Correlate (and meta-level programming in general) improves on the traditional hand-coding of aspects into an application by explicitly separating the main computation and the run-time properties of the application. To make this possible, we use a run-time whose architecture supports meta-objects and allows each class of meta-objects to handle a specific aspect. This mapping of compile-time descriptions of aspects (meta-programs) onto run-time entities (meta-objects) is typical for the aop-related approach: aspects are manifest in the model, in the program code and at run-time.

An advantage of this approach is that aspects are part of an overall architecture, which ensures that their behaviour is well-defined. This makes it easier to reason about the run-time behaviour of the system and the interference and interaction of the aspects. The problem of composing aspects at weave-time is thus transformed into a similar problem of composing meta-objects. In this view, meta-objects are not really necessary at run-time, but may more be a means to reason about composability at a higher level than source code, and to actually build such systems. Even then, we tend to isolate aspects that are orthogonal (and therefore can be modelled separately), so that the composability of meta-objects is feasible. Coping with the more general problem of non-orthogonal aspects still is subject to research [CIOO96].

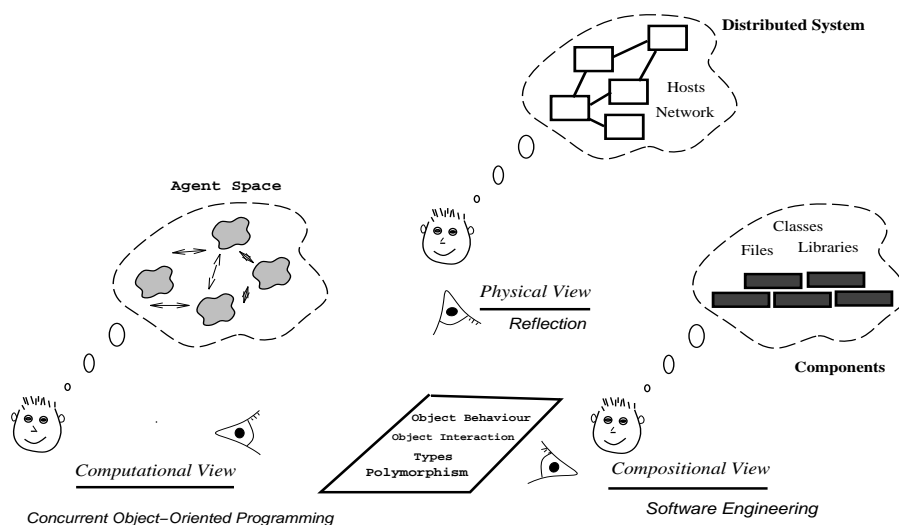


Figure 1: The Correlate views

Aop can directly produce source code without the need for run-time entities, resulting in faster code. This is done by an aspect weaver. The result is that aspects are manifest in the model, in the program code, but not in the final executable. A new element is that aop explicitly tries to weave non-orthogonal aspects together. These weavers can however only be produced for aspects where the interactions and interference between different aspects are well understood: they are relatively difficult to build and they can for the moment only be built for specialized purposes.

Table 1 summarizes the findings we presented in this section, contrasting both aop and aop-related approaches to their predecessors.

Development stage	Property	The early days	Traditional	MOP, OI	AOP
<b>Modelling</b>	<b>Manifestation</b>	not present or implicit	implicit	explicit	explicit
	<b>Equality</b>	N/A	N/A	major/minor	major/major
	<b>Domain</b>	N/A	N/A	some aspects, orthogonal	all aspects, non-orthogonal
<b>Description</b>	<b>Manifestation</b>	not present	hand coded	some aspects	explicit
<b>Run-time</b>	<b>Manifestation</b>	not present	not present	some aspects	weaved

**Table 1: Properties of aspects in the development cycle**

### 3. Evolution of aop

As stated in the previous section, aop can be seen as one stage in an ongoing evolution process. There are two dimensions along which we think aop may evolve. The first is the abstraction level of the languages used to describe the aspects, and the second is the support level for dynamic environments. The evolutions discussed here have one element in common: they both lead to a system where some aspects are explicitly manifest at run-time.

Regarding the aspect description languages, we believe that aspects become truly useful when programmers can describe the aspects they want to use at a high level of abstraction. This contrasts many current examples in the AOP papers where e.g. synchronisation is expressed using low level primitives, such as mutexes. We feel that aop encourages more declarative descriptions of aspects. Aspect definition languages may therefore evolve more and more into interpreted languages, where the weaver contains a specific interpreter for each ADL. This evolution logically introduces the discussion about a run-time dimension of aspects.

Another situation where run-time entities can be desirable or in fact needed, is when the aspects themselves get a dynamic property. The dynamic nature of many applications and environments cannot always be captured in a static description of an aspect. This suggests that a weaver will sometimes need to make some decisions at run-time. Therefore, aspects can have a dynamic (run-time) dimension as well.

As aop is positioned today, it is not well suited to cope with situations such as the ones sketched above. Therefore, we think aop will evolve into allowing at least some aspects to have a run-time presence. Table 2 expands the previous table to include “next generation” aop.

Development stage	MOP, OI	AOP now	AOP tomorrow
<b>modelling</b>	explicit	explicit	explicit
<b>description</b>	some aspects	explicit	explicit
<b>run-time</b>	some aspects	weaved	some explicit, others weaved

**Table 2: Manifestation of aspects in AOP**

However similar the aop-related approaches and aop with run-time aspects may seem, they differ in at least two essential ways: firstly, the former only identify a limited set of aspects whereas aop emphasizes a description in

terms of all sorts of aspects, and secondly, most aop-related approaches make *all* identified aspects explicit at run-time, while we think that aop will eventually be able to let the programmer decide which aspects are to be woven and which ones are to survive at run-time. In this way, the advantages of aop and of current aop-related approaches can be combined: the flexibility of aop-related approaches and the performance of weaved code.

### **An illustration: load balancing.**

As an illustration of the above, we take the example of load balancing as available in the Correlate language and run-time [Joosen]. The Correlate programmer has access to a library of load balancers with different selection, initiation and location policies. When dynamic load balancing is desired, the Correlate programmer instantiates a load balancer with properties that are beneficial for the application at hand. The programmer is guided by a property-based tool [Aksit].

Both aop and meta-object programming can provide a solution to this problem. The main difference at run-time being that the aop solution will weave the loadbalancing components throughout the application code whereas in the other case, the load balancer remains alive as a run-time entity. Both have their good sides: the aop solution will be more efficient while the mop guarantees a greater flexibility.

Now consider an adaptive loadbalancer. In some cases, the selection of an appropriate loadbalancer is difficult. When the application executes in an unpredictable environment, it is hard to choose a loadbalancer that adapts to all changes. For instance, studies on loadbalancing show that receiver-initiated policies work well in a heavily loaded environment, but perform very poorly in lightly loaded system. The reverse can be said for sender- initiated policies. In an environment like a cluster of personal workstations where the load of the system may change drastically due to external effects (like a user logging in and starting some applications) it then becomes very hard to statically make a good overall choice of loadbalancer. The solution here is to create an adaptive load balancer that during execution evaluates the current situation and, if necessary, replaces the current loadbalancer with a more appropriate one. This adaptive loadbalancer can be seen as an interpreter of a high-level declarative language specifying which loadbalancer to use in a certain environment.

Present-day aop faces a problem here. Due to the weaving of application and loadbalancing code, it is difficult to change the loadbalancing policy as this code is dispersed over the entire application code. In tomorrow's aop on the other hand, it should be possible to explicitly specify which aspects (like loadbalancing in the example) are to remain alive during execution. These aspects are represented at runtime as components and can thus be replaced when necessary. On the other hand, aspects that can be statically dealt with (like synchronization) can still be woven as before. This way, tomorrow's aop can be sketched by taking the best of both worlds from meta-object programming and state-of-the-art aop.

## **4. Summary**

In this report we gave our view on current state-of-the-art aop and related it to meta-object programming and more traditional decomposition approaches. An important issue we noted is that in current aop, aspects are only explicit until weave-time. Indeed, the weaver takes the aspect descriptions and tightly interconnects them with the application's functionality.

A key issue we see in tomorrow's aop is the survival of (at least some) aspects at run-time. Illustrated through an example of adaptive loadbalancing, we argue that survival of aspects at run-time is a necessary precondition to ensure maximal flexibility and to allow an aspect to adapt itself based on execution-time information.

## **5. References**

- Aksit M. Aksit et al. "Abstracting Object Interactions Using Composition Filters." In *European Conference on Object-Oriented Programming, Workshop on Object-Based Distributed Programming*. 1993.
- Berghm G. Berghmans. *Een intelligent werktuig voor de samenstelling van belastingspreiders in XENOOPS*. Dept. of Computer Science, K.U.Leuven, Belgium. Master's thesis.
- CIOO96 ECOOP '96 Workshop on Composability Issues in Object-Orientation.  
<http://www.trese.cs.utwente.nl/cioo96>

- Kiczales G. Kiczales. "Towards a New Model of Abstraction in Software Engineering." In *Proceedings of the International Workshop on New Models in Software Architecture '92; Reflection and Meta-Level Architecture*. 1992.
- Joosen W. Joosen. *Load Balancing in Distributed and Parallel Systems*. Dept. of Computer Science, K.U.Leuven, Belgium. PhD. thesis.
- vOeyen J. Van Oeyen, S. Bijmens, W. Joosen B. Robben F. Matthijs, P. Verbaeten. "A Flexible Object Support System as Runtime for Concurrent Object-Oriented Languages." In *Chris Zimmermann, editor, Metaobject Protocols, Proceedings of the workshop on Advances in Metaobject Protocols and Reflection (ECOOP '95)*. 1995.
- Yokote Y. Yokote. "The Apertos Reflective Operating System: The Concept and Its Implementation." In *Proceedings of 7<sup>th</sup> Object-Oriented Programming Systems, Languages and Applications*. 1992.