

Coordination and Composition: *The Two Paradigms Underlying AOP?*

Robb D. Nebbe

Software Composition Group
Universität Bern, Institut für Informatik
Neubrückstrasse 10
CH 3012 Bern, Switzerland

<http://www.iam.unibe.ch/~nebbe/coord.ps.gz>

Introduction

This position paper is based on work recovering architectures from object-oriented systems in the context of the FAMOOS Esprit project. Our experience corroborates the existence of aspects that cross-cut the functionality of a software system. However, when examining how the problems arising from such a situation are dealt with in Ada where the language has built-in support for concurrency and C++ where no such support exists suggests the possibility of a more general approach to aspect-oriented programming based on the following hypothesis about software structure that so far has proven to be correct.

A software system can be structured as a set of independent semantic domains consisting of a core problem domain and a set of coordinated supporting domains.

I will use the term semantics to refer to an axiomatic or denotational notions of semantics where only the result is considered as semantically relevant as opposed to an operational notion of semantics where how the result was obtained is equally important. I will also use the term coordination to mean the linking of actions or instances from different semantic domains. This is a very general notion of coordination of which the more traditional use of coordination in relation to concurrency is an example.

This position paper starts by presenting my hypothesis about software structure in greater detail followed by a resume of the problems observed in our legacy systems. We then briefly discuss the possibility of extending a language to solve these problems and introduce the principles of orthogonality and transparency. Finally, this leads to a view of AOP as encompassing two paradigms, composition and coordination, that underlie the distinction between objects and aspects.

Software System Structure: A Hypothesis

This hypothesis was constructed to explain numerous observations about several FAMOOS¹ case studies consisting of object-oriented legacy systems provided by the industrial partners of the FAMOOS consortium.

A software system can be structured as a set of independent semantic domains consisting of a core problem domain and a set of coordination supporting domains.

A model of the core problem domain (or core domain model) captures the basic functionality of the software system. The supporting domain models help by providing services such as

¹FAMOOS is an Esprit project. <http://www.iam.unibe.ch/~famoos/>

synchronization, threads, persistency, and RMI that are needed to implement this functionality. Typically the relationship with these supporting domain models is hidden in the implementation of the core domain model where it is coded into the various classes.

We identified two different situations as being problematic:

- The code relating the core problem domain to the supporting problem domains is distributed across the classes of the core problem domain. Furthermore, this code is often fundamentally similar and is in effect duplicating a single policy in different contexts. For example the synchronization policy is often the same but adapted specifically to each class; changing the policy requires changing each class.
- Distinctions are lifted into the core domain model, where they are totally irrelevant, from supporting domain models in order to facilitate coordinating the two models. For example classes are split into synchronized and non-synchronized variants or persistent and non-persistent variants thus increasing the chance that the underlying similarity will go unnoticed.

Both situations complicate understanding the software system. The duplication in the first situation greatly increases the chances of human error in adapting and implementing a policy in a particular class. The second situation creates a risk that if the model of the problem domain is adapted then not all of the variants will be recognized as capturing a single concept from the problem domain and they will thus become inconsistent.

Both problems are aggravated by the fact that classes provide the only means of organizing abstractions. One means of getting around this limitation is to extend the programming language to support what is in our terminology a supporting domain model directly. In the next section we look at two such cases.

Language Extensions: Orthogonality and Transparency

In [Moss96] Moss and Hosking suggest that an extension to Java supporting persistence should be both orthogonal and transparent; principles they attribute to [Atkinson95]. What they mean by orthogonal is independent from the type system. This implies that persistency can be applied independently to a single object or perhaps a set of objects independent of their respective types. Transparency relates to how many decisions a programmer must make in the source code relating to a particular extension. Moss and Hosking point out that this relates to the amount of control the programmer has over the extension. If an extension is completely transparent then its use is entirely a consequence of the semantics of the core domain model. In AOP transparency does not translate into a total lack of control but it does imply a reliance on the semantics of the core domain model to provide the appropriate join points.

If we look at our two situations identified as problems we see that the second relates to a lack of orthogonality; however, the first is more complex. The distribution throughout the source code is a problem of transparency but the fact that fundamentally similar code is duplicated relates to the fact that in order to formulate a general policy capturing the underlying similarity one must have fairly complete reflective facilities and the distinctions upon which the policy is based must be discernible in the source code.

Another example of extending a language to encompass supporting domains is the Illinois Concert C++ system (ICC++). Even though they do not use the principles of orthogonality and transparency to explain their work they are quite appropriate. Using a simplified version

of C++² as a base ICC++ handles locality, communication, thread creation and synchronization both transparently and orthogonally as well as efficiently [Chien97].

The underlying argument in ICC++ (coming from my understanding of [Plevyak96]) is that the transparency is critical to obtaining a respectable level of performance. If the approach were not transparent then the programmer would be required to program to many assumptions about the context in which an application was to execute thus crippling the attempts of the compiler to produce efficient code. Orthogonality is also a critical aspect since many optimizations involve locking sets of objects with different types as a single collection.

What is particularly interesting about ICC++ is that the language provides very high level guarantee that the intermediate states of an object are not observable. This facilitates reasoning locally about the semantics of the source code since these semantics will be preserved by the compiler. I suggest that the goal of having the semantics defined in the core domain model independently of any aspects is similarly important if we are to be able to reason locally. It also helps to formulate a clear separation of responsibility between objects and aspects.

Two Paradigms: Composition and Coordination?

I suggest that a composition language is good at defining individual semantic domains including both the core domain model and any supporting domain models. What a composition language is not good at is coordinating these different domain models. This is because in order to support the separation of concerns the coordination should be both orthogonal and transparent.

The traditional definition of an aspect makes the distinction between aspects and objects somewhat unclear [Lopes97]. I feel this is due to the fact that this notion encompasses a particular supporting domain model as well as the coordination of this model with the core domain model. For example, section 3.3 of her thesis [Lopes97] “Design Decisions and Alternatives” relates design decisions and alternatives essentially covering the domain models behind COOL and RIDL.

What is truly unique about an aspect is the ability to coordinate separate domain models. This consists of linking a set of objects from one domain model with a set of object from another domain model and then linking actions in one domain to actions in another domain.

Examples of what I consider to be coordination include: when a new instance is declared the domain model handling storage must allocate memory for the instance; when an object needs to be synchronized, it can be linked to a semaphore and before any of its methods are invoked the semaphore must be seized and afterwards it must be released; choosing how to communicate with an object based on whether it is local or remote.

In this view an object is written in a composition language and is based on the paradigm of composing larger objects from smaller objects. In contrast an aspect coordinates objects from a supporting domain model to implement and ensure the semantics of the core domain model. Semantics are defined within a domain model but many nonfunctional properties such as fairness and liveness are consequences of a supporting domain model and how it is coordinated with the core domain model.

²The restrictions are similar to those adopted by Java such as no casting between pointers and int's, no pointer arithmetic etc.

This suggests that a general aspect language would be open ended since any supporting domain model could be replaced or extended as needed rather than writing a new aspect language. However, this is dependent on the ability to define the semantics within a single domain model.

The independence of a supporting domain model from the core domain model is relatively straight forward since the coordination always appears in the classes of the core domain model. The inverse relationship, the independence of the semantics of the core domain model from the supporting domain models, is more subtle.

In COOL [Lopes97] the aspects are not allowed to change the state of an object. This is rationalized as preserving a clear separation of responsibilities. My view of coordination implies that an aspect does not change the state of any object; rather, it links actions in one domain model to those in another. This means that as a consequence of actions in one domain model, actions will take place in another but the semantics of these actions must be completely defined within their respective domain models. This is what is meant by semantically independent.

Of course there remains the case where there is a problem with the coordination. For example, a synchronization scheme that lets two writers modify an objects state at the same time. From the point of view of the core domain model this is somewhat like having an error in the compiler. The code in the core domain model is correct but its semantics are not correctly implemented in the executable.

Now assuming that the composition language has a full reflective interface and the necessary semantic distinctions are clear. In order to allow general policies to be formulated, of which an aspect is an instance we only need to allow access to this reflective interface from the coordination language in which aspects, or now policies are written. This solves some of the confusion linked with reflective solutions to coordination problems caused by the fact that metaobjects have no clearly defined role and in fact can do almost anything.

Discussion

If we assume that the theory set forth is true then we can characterize aspect-oriented programming as follows (somewhat paraphrasing [Kiczales97]):

Object-oriented technology provides a good fit for defining the semantics of individual problem domains. However, many important non-functional issues cross-cut any such domain. This situation arises because the relationship between a core domain model and its supporting domain models is not one of composition, upon which the object-oriented paradigm is based, but of coordination.

I feel that an important step is to automate much of work surrounding aspects. Just as many things can be built into a compiler today I think that it may be important to write general policies that automate the use of aspects in much the same way that memory management is automated in many programming languages. Most people will not be interested in writing aspects if there are already adequate solutions. However, if they feel the need to tune their specific application then they have the possibility to do it themselves rather than being locked in to a single domain model and its associated coordination as when it is implemented in the language.

Another advantage is the separation of the core domain model which is an important asset to the company who developed the software from the supporting domains that the company

needs but has no special interest in developing or maintaining. However, since supporting domain models can be shared across many different applications there should be a market for them.

The clear separation between the concepts of an object and an aspect and the underlying composition and coordination paradigms is advantageous. It is the result of a further separation of concerns where the coordination aspect is factored out of the specific domain model to which it is coupled.

Unfortunately this doesn't explain all the areas that have been represented as aspects. Memory allocation is easily handled by this theory, however, the loop unrolling found in [Kiczales97] is more difficult to characterize. It could be viewed as a relationship between the language domain model and the core domain model but I personally find this unsatisfactory.

Errors fit into this theory but not as their own aspect. Instead Errors must be separated in those that are within a core domain model and those which are external. Those within the core domain model often represent a programming error. However, since there is no semantic relationship with the supporting domain model an external failure represents the inability of supporting model to perform a coordinated action. This can result from physical failure such as the loss of a connection or exceeding the available resources such as memory or missing a deadline. Such an analysis is consistent with the way in which exceptions are used in embedded systems written in Ada.

Bibliography

[Atkinson95] M. P. Atkinson and R. Morrison, "Orthogonally Persistent Object Systems", *Int. J. Very Large Data Bases* 4, 3, 319-401, 1995

[Chien97] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, X. Zhang, *High Level Parallel Programming: The Illinois Concert System*, submitted for publication 1997

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997

[Lopes97] C. I. V. Lopes, *D: A Language Framework for Distributed Programming*, Ph.D. thesis, Northeastern University, Nov. 1997

[Moss96] J. E. B. Moss and A. L. Hosking, "Approaches to Adding Persistence to Java", in *First International Workshop on Persistence and Java*, Technical Report 96-58, Sun Microsystems Laboratories, Nov. 1996

[Plevyak96] J. B. Plevyak, *Optimization of Object-Oriented and Concurrent Programs*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996