ASPECTS & CROSSCUTTING IN LAYERED MIDDLEWARE SYSTEMS

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

1. INTRODUCTION

Today’s middleware systems aim at providing a uniform interface for the application designers by abstracting from the local operating system and network technology. For this purpose, middleware systems provide distribution transparencies and functions that support application development and delivery. Uniform interfaces to a distributed system facilitate faster development of applications and easier porting of applications to other computer platforms.

To manage the complexity, middleware systems are decomposed into the so-called horizontal and vertical dimensions. Horizontal decomposition is modeled as a set of cooperating objects. The vertical decomposition is modeled as a stack of logically organized services called layers.

If the interfaces of objects and layers are well-defined and stable, then the evolution of a system will only have a local effect. Unfortunately, not all the concerns of a distributed system can be defined and encapsulated by the interfaces of objects and layers. For example, as indicated by several authors (e.g. [Becker 98]), QoS concerns such as performance and reliability relate to multiple objects in multiple layers. This, the so-called crosscutting aspects hinders the adaptation of middleware systems since changes cannot be any longer restricted to the implementations of objects and layers.

Consider for example a client object that sends a message to a set of server objects using the TCP/IP protocol. Initially, only a few server objects receive the message. Now assume that the client object starts to send the same message to a large set of server objects. After a certain number of server objects, a multicast protocol would be more efficient than the TCP/IP protocol. Transparent QoS management requires the system to automatically switch from TCP/IP to a multicast protocol.

We have been experimenting with this scenario using the Orbacus CORBA implementation. This implements an interface (OCI) to separate the protocol layer from the middleware layers. We have realized, however, that it is not possible to dynamically substitute the protocol layer without making changes to the other layers. The origin of the problem was twofold. Firstly, due to the synchronization schemes of multiple threads flowing through the layers; the middleware layers make certain assumptions about the synchronization policies (e.g. related to buffering) of the protocol implementation. Since these assumptions are not specified explicitly, changing a protocol implementation may cause synchronization problems if the newly instantiated protocol behaves differently. Secondly, to implement an open-ended protocol switching mechanism, a conditional delegation mechanism is required to support the dynamic switch between objects with different interfaces. Using case statements and Strategy pattern-like implementations are limited in that they assume a fixed set of alternatives.

In the literature, various reflective techniques have been proposed to extend the functionality of the middleware systems. For example, portable interceptors have been introduced to extend the behavior of middleware by using message reflection. Although reflection is the right step for making systems extensible, it does not solve the crosscutting problem per se. For example, portable interceptors do not provide a direct solution to the thread synchronization and conditional delegation problem.

Aspect-oriented techniques have been introduced to solve the crosscutting problems. In this paper we describe this problem, define a set of requirements and propose an aspect-oriented approach to the construction of middleware using Composition-Filters.

2. ASPECTS & CONCERNS: BASIC CONCEPTS

In this section, we will very briefly describe the issues in constructing software that mixes and distributes concerns over several parts of the program, such as different layers.

The following shows the three layers of a (middleware) system that all consist of a number of objects. However, certain aspects (i.e. specific concerns of the system such as performance, reliability, synchronization) crosscut these layers.

An illustration of mixed & distributed concerns

The above figure illustrates an example of multi-dimensional separation of concerns [Bergmans 97, Tarr 99]: first a separation into layers, and a separation within the layers into different objects. Secondly, a ‘vertical’ separation into aspects, which crosses the boundaries set by the layers and objects. With today’s technology, a feasible solution is to distribute the code related to a one aspect over the code that implements the objects which are possibly located in different layers. The resulting mixture is sometimes referred to as ‘tangled’ code. In case of a middleware system, the tangled code may be implemented over the multiple layers. It exhibits bad maintenance and adaptability properties. For example, tailoring thread synchronization scheme necessary for instan-
tiating different protocols may not be easily realized by dynamic switching of protocols.

Recent work on Aspect-Oriented Programming (AOP) has highlighted such design considerations [Kiczales 97, Bergmans 99]. We distinguish the following issues in composing multiple dimensions of concerns:

1. **Composition direction**
   - **import**: a module specifies how its behavior is composed from imported behavior from other modules. This is the object-oriented approach.
   - **export**: a module specifies how its behavior is superimposed [Bougé 88] upon other modules. This is the aspect-oriented approach.

2. **Crosscuttings**
   - **uniform**: a single concept implementation is composed repeatedly within various contexts. This form of repeated functionality is typical for systemic concerns: behavior that applies to all entities in a (sub)system.
   - **multiform**: a single concept is divided into sub-concepts, each of which is composed in a different context. Typical examples occur in a layered system where functionality such as thread synchronization is addressed (at distinct levels of abstraction) by several layers.

3. **Granularity**
   We can consider composition at several levels of granularity; we discuss the two prevalent granularities:
   - **class-level**: this is best understood, since it considers only first-class abstractions that are composed as a whole. Examples are object-oriented inheritance, delegation and aggregation.
   - **method-level**: to achieve compositions that are better tailored to the target context, more fine-grained composition is necessary. In recent years, many approaches such as Adaptive Programming [Lieberherr 95], Subject-Oriented programming [Harrison 93] and Composition Filters [Aksit 92] have provided composition at the method-level (or even more fine-grained).

**3. REQUIREMENTS**

We now propose a set of requirements on a composition mechanism that is needed to support middleware that consists of a multitude of aspects and that can cope with both evolutionary and dynamic adaptability requirements. These requirements derive from the analysis of composition anomalies in e.g. [Bergmans 94b, 96b]. We use the less overloaded term **element** for the individual modules/grains that the composition scheme can deal with.

In a nutshell, these requirements are about the following issues, starting with three requirements related to splitting a program into elements that deal with separate concerns:

1. Be able to separate a program into distinguishable concerns that can be defined and addressed separately.
2. Sufficient expressiveness of the computation model to express the various concerns in a program.
3. Be able to combine/compose given concerns—with some sensible limitations—into a working program.

The following requirements deal with the specification of the mapping between the elements:

4. Distribute a concern specified in one element over one or more other concerns.
5. Be able to do that either by importing concerns to compose them into a complex element, or by exporting concerns, super-imposing them upon other elements.
6. The various elements have dependencies between each other, the resulting coupling should be minimized.
7. Different binding times of the mappings between the elements should be possible.

**4. THE ROLE OF COMPOSITION FILTERS**

As a solution strategy to address the issues we presented in the previous section, we propose the application of composition filters. Since space does not permit an explanation of this enhancement of the object model, we refer to e.g. [Aksit 92, Bergmans 94, 96b]. We will here just point out the major contributions of the composition filters (CF) model with respect to the requirements we just presented in the previous section:

1. The CF model supports different filter types for different aspects (both predefined and user-defined filters are possible). Examples of aspects that predefined types deal with are: synchronization, conditional delegation, real-time constraints, assertions, inter-object communications. The independent specification of each of these aspects supports a more fine-grained separation of concerns.
2. The filter types as mentioned above can also be used to enhance the expressiveness of a system, e.g. by introducing new types for synchronization or message reflection.
3. A key characteristic of the CF model is the ability to compose objects from several aspects and from (aspects of) other objects, for example by specifying a series of filters.
4. The expressions that are used to instantiate a filter type allow polymorphic mapping of a concern upon a group of methods from one or more objects.
5. The CF computation is especially suited for specifying compositions of imported objects. Achieving the export of behavior can be achieved through code-weaving techniques.
6. Coupling between objects is minimized through strict interface dependencies only, and adopting a black-box (aggregation) composition strategy, rather than the more sensitive white-box (inheritance) approach.
7. The mappings specified in filters assume run-time bindings. However, the declarative specification of filters allows a compiler to detect static (unconditional) bindings.

**5. CONCLUSIONS**

In this paper we illustrated the crosscutting problem in realizing composable middleware systems. Section 2 summarized the current AOP techniques that deal with the crosscutting problem. This led to a set of requirements. Finally, the CF model was presented as a possible solution for designing composable middleware systems.

**ACKNOWLEDGEMENTS**

The work presented here is being carried out within the AMIDST project, which is funded by the Telematics Institute. We want to thank all members of this project, in particular Ali Noutash for his work on aspects related to transport protocol plug-ins in CORBA.
REFERENCES


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