ACP4IS Workshop at AOSD'04

Towards Integrating Aspects and Components

Houssam Fakih[1,2] fakih@ensm-douai.fr
Noury Bouraqadi[1] bouraqadi@ensm-douai.fr
Laurence Duchien[2] duchien@lifl.fr

March 22, 2004

Integrating aspects and components can be important for both AOSD and CBSD. On the one hand CBSD suffers from crosscutting and tangling code [7]. On the other hand, actual AOSD technologies are not mature enough to enable aspect reuse [6, 4]. So, each paradigm can resolve other's paradigm limitations.

Problem Statement

The integration of AOSD and CBSD is a complex task that can be subdivided into three facets.

1. Facet 1 consists in componentizing aspects. That means representing each aspect as a single reusable component. Basic characteristics of a component include attributes, provided and required services. The challenge is to map these characteristics on aspects. We have also to explore the applicability of related concept such as connector, composite and sub-component on aspects.

2. Facet 2 consists in aspectizing a component-based software. Nowadays, AOSD is used in conjunction with object oriented or procedural languages. Base code is expressed using either an object oriented language or a procedural one. The second facet allows extending this list with component based languages. Set differently, the second facet consists in defining aspects that act on base code expressed in terms of components and related concepts. In this context, we have to define weaving and join points on execution flow and structure of components and related concepts.

It is worth noting that there is a variety of component models. Thus, redefinition of AOSD concepts will certainly vary according to the model used to implement base code. A solution proposed for a flat component model (*i.e.* a model without composite concept) will probably not be applicable to a hierarchical model (*i.e.* a model with composite concept). However, we believe that some solutions could be transposed between some models. In the case of component model with explicit connectors there would be an extra relationship : the one between connectors and aspects. But, this relationship can be transposed to a component-aspect relationship. Indeed, we agree with Sacha Krakowiak that components and connectors are two entities of the same nature (i.e. structure/behavior) but with different roles [5] . So, connectors can be considered as components dedicated to connection.

3. Facet 3 is a merge of the two previous facets. It consists in unifying aspects and component-based software by defining a general enough component model to encompass not only "traditional" CBSD concepts, but also AOSD concepts. This unification should lead to a single definition that should apply for both aspects and components. In this context, weaving aspects with a base code consists of assembling components from a base code with components representing aspects. We identify two differences between assembly and weaving mechanisms.

- First in CBSD, all participating components in an assembly are aware of their assembly points and the provided or required services. On the contrary in AOSD only aspects are aware of assembly points (join points) and services they provide to change or adapt the base code normal execution.
- Second, the assembly mechanism is not intrusive like weaving. The former keeps components intact while the latter often changes base-code structure and behavior.

Note that as for facet 2, solutions will probably vary according to concepts provided by the chosen component model.

First steps towards the integration

1. Provided and required services are among components characteristics. In a componitizied aspect, provided services include advices. Indeed, advices should be triggered in order to execute. Introductions are also part of provided services of a componitizied aspect. This is because their execution can be

[1] Ecole des mines in Douai. France. *http://csl.ensm-douai.fr/research/*

[2] Lille University Of Sciences and Technology. France. *http://www.lifl.fr/GOAL/*

triggered. Introductions are performed when code elements to extend/change are given. One possible solution for facet 1 consists in using the concept of contract [1]. Contracts seem to be applicable for provided services corresponding to both advice and introductions. Syntactic contracts (types) for advice enforce the type of acceptable join points (e.g. message to be sent or to be received, field access, ...). While behavioral contracts for advices check their invariants and pre-post conditions (e.g. change of a log file for a logging aspect). For an introduction, a syntactic contract corresponds to the kind of constructs (class, method, ...) to which the introduction is applicable.

2. One possible solution for facet 2 consist in defining entry points on components [3] that allow to change its internal behavior. Thus, aspects are plugged on these entry points. The definition of join points depends on component model. We can identify some basic join point families common to all component models such as actions related to the component state (creation, initialization, ...) or provided or required component services (call of services, connecting or disconnecting components,...)

3. One viable solution for Facet 3 could be to define a reflective component model (figure 1). We distinguish two kinds of components : a base component defines application business features while a meta-component defines how to perform these features (aspects) [2].

Each component should have an extra-functional interface (an extra-functional interface corresponds to an entry point) allowing it to be connected to a meta component. A meta-component controls one or several base components. A componentizied aspect can be made up of one or several meta-components. So it can be considered as a single (Aspect 2 and Aspect 3) or a composite component (Aspect 1) as we show in figure 1. Composite components assemble more than one meta-component in order to represent a single aspect. It manages also the visibility of provided and required meta-component interfaces. Representing aspect as a set of meta-components has the advantage to make no difference between assembly components and weaving componentizing aspects mechanisms. In cases of one component controlled by several meta-components. We have to manage potential conflicts among meta-components. One solution consists of either using a chain of responsibility. The ideal solution is to give users the possibility to change or manage conflicts. We could do it by using a meta-component dedicated to conflict resolution. The same strategy can be used in case of conflicts among componentizied aspects. Conflicts can be addressed defining an adapter component that link componentizied aspects to base components.

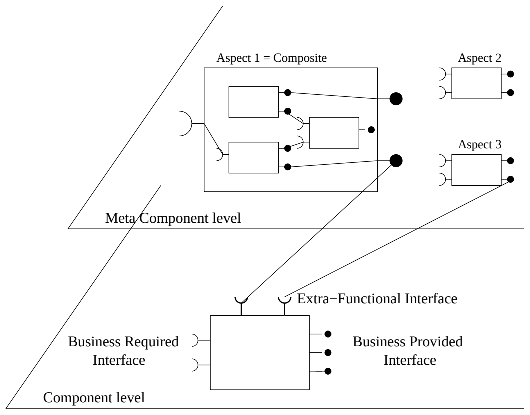
Aspect 1 = Composite
Aspect 2
Aspect 3
Meta Component level
Extra−Functional Interface
Business Required Interface
Business Provided Interface
Component level

Figure 1: a reflective component model

References

[1] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, jul 1999.

[2] N. Bouraqadi and T. Ledoux. *Aspect-Oriented Software Development*, chapter 11 – Supporting AOP using Reflection. Addison-Wesley, 2003.

[3] Patrice Gahide, Noury Bouraqadi, and Laurence Duchien. Promoting component reuse by integrating aspects and contracts in an architecture model. In Yvonne Coady, editor, *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 51–55, Enschede, The Netherlands, April 2002. University of British Columbia.

[4] Stefan Hanenberg and Rainer Unland. Using and reusing in aspectj. *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, October 2001.

[5] Sacha Krakowiak. Patrons et canevas pour les intergiciels. Talk given at 4th Summer school on distributed systems in Autrans, France, August25 2003.

[6] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, March 1999.

[7] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software Practice and Experience*, 33:957–974, 2003.