A component-based and aspect-oriented model for software evolution

Nicolas Pessemier, Lionel Seinturier*
and Laurence Duchien

INRIA-Futurs – LIFL, Projet Jacquard/GOAL,
Bâtiment M3, 59655 Villeneuve d'Ascq, France
E-mail: nicolaspessemier@gmail.com
E-mail: Lionel.Seinturier@lifl.fr E-mail: Laurence.Duchien@inria.fr
*Corresponding author

Thierry Coupaye

France Telecom R&D,
28 chemin du Vieux Chêne,
BP98, 38243 Meylan, France
E-mail: thierry.coupaye@orange-ftgroup.com

Abstract: Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) are solutions to support software evolution by decomposing a software system into concerns. In this paper, we propose Fractal Aspect Component (FAC), a general and symmetrical model for components and aspects. FAC decomposes a software system into regular components and aspect components which embody crosscutting concerns. We reify the relationship between an aspect component and a component, called an aspect binding, as a first-class runtime entity. The evolution of the system can be expressed by adding or removing components (aspect or regular) and by setting bindings (regular or crosscutting).

Keywords: aspect-oriented software development; AOSD; component-based software development; CBSD; crosscutting concern; aspect component; aspect binding; aspect domain.

Reference to this paper should be made as follows: Pessemier, N., Seinturier, L., Duchien, L., and Coupaye, T. (2008) 'A component-based and aspect-oriented model for software evolution', *Int. J. Computer Applications in Technology*, Vol. 31, Nos. 1/2, pp.94–105.

Biographical notes: Nicolas Pessemier is a PhD student in Computer Science at the University of Lille. His research interests include the merging of component-based software development and aspect-oriented software development.

Lionel Seinturier is a Research Associate at INRIA in Lille, France. He works on Software Engineering Techniques for designing and implementing middleware. From 1999 to 2003, he was an Assistant Professor in the Computer Science department of the University Pierre et Marie Curie, Paris, France. He received his PhD in computer science from CNAM, Paris, in December 1997. Before joining academia, he worked as a Research Engineer for France Telecom's R&D Department, on the integration of the ATM network technology with CORBA middleware. He is the co-author of one book and of 25 international publications. His current research interests include aspect-oriented and component-based software engineering.

Laurence Duchien is Professor at University of Lille in the INRIA Jacquard project-team. Until 2001, she was Associate Professor in the Computer Science Department at CNAM, Paris, France. She received her PhD from the University Pierre et Marie Curie, Paris, in 1988 and her Habilitation à Diriger des Recherches from University Joseph Fourier, Grenoble, France, in 1999. Her research interests include design methodologies, architecture description languages, and aspect-oriented and component-based software engineering.

Thierry Coupaye is senior research expert and head of the Distributed Software Architectures and Infrastructures research group at France Telecom R&D Division. He completed his PhD in Computer Science from the UJF Grenoble University, France, in 1996 in the area of active databases (Event-Condition-Action rules) and worked afterwards as a Teaching and Research Assistant at INPG Technological University. Then he worked as a researcher at the European Bioinformatics Institute (EMBL-EBI) in Cambridge, UK, in the area of semi-structured data management for genomics, and then in the Dassault Systems and University of Grenoble Joint

Laboratory on large-scale software deployment. His research interests include middleware architecture, reflexive component-based systems, aspect-oriented programming and autonomic computing.

1 Introduction

Software evolution is the subject of many studies both in academia and in industry. Indeed, a major part of current software development is devoted to software maintenance. Software systems need to evolve continuously to meet new software requirements (Parnas, 2002). Concern became a central concept to cope with software evolution by decomposing a software system into smaller and more comprehensible modules. CBSD and AOSD propose solutions for this decomposition. CBSD structures a program by separating concerns into reusable components. AOSD modularises crosscutting concerns into aspects, by identifying tangled and scattered code into systems.

However, it has been shown that the issues of code tangling and scattering arise at the level of CBSD as well (Duclos et al., 2002; Lieberherr et al., 1999). The integration of AOSD into CBSD is thus an important step in software development and has been proposed several times (Lagaisse and Joosen, 2005, Mezini and Ostermann, 2003, Suvée et al., 2003). However this one-way integration leads to drawbacks for software evolution. Indeed, most current AOSD approaches are asymmetric. This means that aspects and components are, structurally speaking, different entities: they are composed using different rules. Thus, the system becomes difficult to maintain and evolve because of the two dimension to consider.

In this paper, we propose a general and symmetrical model for components and aspects. The approach improves software evolution by taking advantage of CBSD and AOSD approaches. It provides a support for AOSD to the component approach, and applies CBSD concepts to AOSD. Our proposal relies on three main notions: aspect component, aspect domain, and aspect binding. An aspect component is a contractually specified component which embodies a crosscutting concern. An aspect domain is the reification of the components picked out by an aspect component. An aspect binding is a binding between a regular component and an aspect component.

Our unified model for components and aspects raises software evolution to a higher level of abstraction, which is a fundamental issue to tackle (Mens et al., 2005). We propose the co-evolution between design models (an architecture of components and aspects) and the source code (implementation of our model, providing strong reflection capabilities). Indeed, we validate our model by extending a reflective and general component model, named Fractal (Bruneton et al., 2004) and its ADL. In our extension, called Fractal Aspect Component (FAC for short), we introduce the notions of aspect component, aspect binding and aspect domain to the component model itself and to the ADL.

The remainder of this paper is organised as follows. In Section 2 we introduce our general model for components and aspects. Section 3 presents the mapping of our model to the Fractal component model. Section 4 illustrates FAC with an example. Section 5 presents related work around the merging of components and aspects and around software evolution. Section 6 concludes and gives some open issues.

2 A general model

This section describes the three main concepts we introduce to support AOSD in a component model: *aspect component*, *aspect binding*, and *aspect domain*. Section 2.1 introduces AOSD main concepts and CBSD principles. Section 2.2 gives the motivations of our model. Section 2.3 describes our three main concepts. Then, we discuss in Section 2.4 the benefits we can derive from our symmetric approach.

2.1 Background

This section provides some general definitions on AOSD and CBSD.

Aspect-Oriented Software Development (AOSD) aims at modularising crosscutting concerns, which cuts across multiple objects, components or any other software entity. Even if no standardisation exists for AOSD concepts, most of AOSD approaches use the following terminology and concepts:

- *Aspect*. An aspect is the modularisation of a crosscutting concern. Approaches such as AspectJ (Kiczales et al., 2001), AspectWerkz (Vasseur, 2005), or JAsCo (Vanderperren and Suvée (2004)) consider aspects as different entities from those that compose the base system. These approaches are said to be asymmetric. Some other approaches such as FuseJ (Suvée, 2005), HyperJ (Lai et al., 2000), or our proposal adopt a symmetrical approach: aspects are represented as components. It increases the reusability of aspects because components are conceived as highly reusable entities. We elaborate on component and aspect symmetry in the following sections.

- *Advice code*. Advice codes implement the behaviour of an aspect. Similarly to classes and methods, the behaviour of an aspect can be split into several advice codes. Several types of advice code exist: before, around, after returning, or after throwing.
- *Join point*. A join point is a point in the execution of a program. Aspects are added on join points. Generally, join points are method invocations or calls, exception catch blocks, or field accesses.
- *Pointcut*. A pointcut designates a set of join points. It is used to specify where advice codes will apply. Some approaches such as JAsCo (Vanderperren and Suvée, 2004) JBoss AOP (Burke et al., 2005), or CaesarJ (Mezini and Ostermann, 2003) separate the pointcut definition from the advice code definition. It increases the reusability of advice codes. Pointcuts are frequently tied to the application because their definition is based on identifiers (class, method, field names) specific to applications.
- *Weaving*. The process of weaving an aspect to a set of base objects consists in assembling these entities together to produce the final application extended with the behaviours defined in the aspects. We distinguish static (compile time) weaver from dynamic (runtime) weaver.

Component-Based Software Development (CBSD) is a development paradigm that aims to improve software development and to reduce costs by assembling systems from software components. A software component is understood as

> "(...) a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Szyperski, 2002).

When describing a computer software system, software engineers often talk about the architecture of the system, where an architecture is generally considered as a component assembly. Thus, Architecture Description Languages (ADLs) are often used jointly with a component model. ADLs frequently use the notions of component, binding, and composite-component (Medvidovic and Taylor, 2000). A binding is a contractually specified relationship between two component interfaces or ports. Composite-components encapsulate other components defining a hierarchy of components to represent a system.

2.2 Motivations

When merging AOSD and CBSD two dimensions have to be considered: the integration of aspect-oriented principles into component-based systems, and the application of component-based principles to Aspect-Oriented Programming.

The integration of AOSD into CBSD is motivated by the code tangling issue inherent in CBSD (Duclos et al., 2002; Lieberherr et al., 1999). Indeed, whatever the choices made to design a component assembly are, some components mix different concerns, and a same concern is often spread over several components. Thus, the code tangling and the code scattering issues, which generally appear in object-oriented programs, arise in component-based systems, as well.

On the other hand the application of CBSD concepts to AOSD is less investigated. Most AOSD approaches such as AspectJ, use an asymmetric representation of an aspect. In other words, aspects and components are two different types of entities. It appears that the weaving of an aspect on a set of components uses different composition rules than the composition of components together. The weaving of an aspect uses pointcut declarations, which are tied to the structure of the base components. Consequently, when the system evolves, namely the structure of those components, the efforts to maintain the whole system are multiplied. In addition, the implicit relationships created between the piece of advice code (which compose an aspect) and the advised components, are never explicitly discernible and can surely not be individually manipulated at runtime. In brief, state of the art AOSD approaches fails in making the aspect-component composition evolvable (Tourwé et al., 2003).

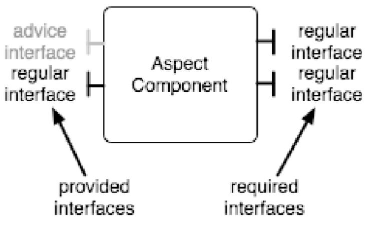
2.3 Overview

In our proposal we realise a twofold integration of CBSD and AOSD. We introduce three main concepts: *aspect component, aspect domain*, and *aspect binding*. These three notions are closely related to the three main concepts of the component approach: component, composite, and binding.

Aspect component

An aspect component (AC for short) embodies a crosscutting concern. It is a regular component providing as services pieces of advice code. Each advice code is encapsulated into a provided interface. Figure 1 represents a conceptual view of an aspect component which provides an advice interface. This interface represents the behaviour which will be woven around a set of regular components. A component is considered as an aspect component as soon as it provides at least an advice interface. The aspect component notion is similar to the notion of Aspectual Component proposed in 1999 by Lieberherr et al. (1999) to express each aspect separately in a modular structure.

Figure 1 An aspect component



Aspect binding

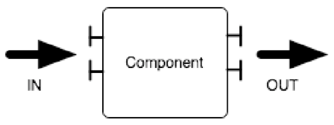
An aspect binding is the reification of the individual relationship between an aspect component and a regular component where the aspect component applies. A regular component exposes a set of join points on which aspect components can be woven. When weaving is performed, aspect bindings are created from each component where a join point is picked out by a pointcut to the aspect component associated to this pointcut. The notion of aspect binding is close to the pointcut interface notion proposed by Gudmundson and Kiczales (2001).

Basically, an aspect binding is set between a component and an aspect component. More precisely it is set between an intercepted interface of a component and the advice interface of an aspect component. This leads us to the definition of our join point model.

Join point model

Our join point model is composed of two different types of join points: incoming calls on provided interface operations, and outgoing calls on required interface operations (see Figure 2). This choice is motivated by the fact that we consider AOSD in a component world. Since components are black boxes, it is rather natural to consider only join points on externally visible elements, i.e., exported and imported interfaces. Taking into account other kinds of join points, such as the ones on implementations, would break component encapsulation. Yet, for cases where this would be necessary, we believe that a best practice is to use a combination of component-based and implementation (e.g., object) based aspect-oriented tools.

Figure 2 Join point model



The level of interception we define is very similar, at the component level, to the composition filters approach (Aksit et al., 1992), which defines IN and OUT filters on objects to intercept messages.

Pointcut language

The pointcut language we use to select join points is based on a pointcut expression, divided in two parts:

- A keyword that specifies if the incoming calls (keyword SERVER) or outgoing calls (keyword CLIENT) or both of them (no keyword) must be selected
- Three regular expressions separated by semicolons that specify which components, interfaces, and operations must be selected.

Figure 3 gives the grammar of the language and Table 1 gives some examples of PcDs. The regular expressions rely on the *java.util.regexp* package.

Figure 3 Pointcut language grammar

```
pcd:= jp_type component;interface;method
jp_type: CLIENT | SERVER | BOTH
component: regular expression on component name
interface: regular expression on interface name
method: regular expression on method signature
```

Table 1 Pointcut language: examples

Pointcut expressions	*Captured elements*
;;deposit*:void	Every incoming and outgoing methods returning void that start with deposit in any component and interface
CLIENT B;*;deposit*	Every outgoing methods named deposit in any interface of a component named B
SERVER B;ITransfert;*	Every incoming method in the interface ITransfert in a component named B

Pointcut expressions are used to weave an aspect component on a set of components. When an interface of a component matches the expression, the interface is bound to the aspect component. In order to keep clarity over crosscutting relationships defined within our system, we introduce the notion of aspect domain, which we detail next.

Aspect domain

An aspect domain is the reification of the components picked out by an aspect component. The goal of an aspect domain is to keep an overview on all the components affected by an aspect. It offers an abstraction on each aspect component woven on a set of components. It can be seen as a reification of the notion of pointcut.

Figure 4 illustrates the notion of aspect domain. In the first part of the figure an aspect component *AC1* is woven on the components *A* and *B*. In the second part, an aspect component *AC2* is woven on *B* and *C*. After this two weavings, two aspect components apply on *B*, and *B* is now shared by three composite-components: the two aspect domains *AC1* and *AC2* and the composite-component of the original architecture.

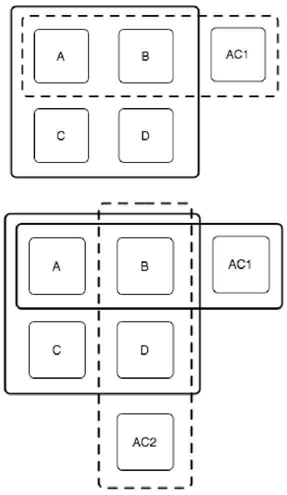
Composition of aspect components and components

Although an aspect component is a regular component, the composition of the two kinds of entities can be done using regular or aspect bindings. The composition rules between aspect components and regular components follows:

- The *component-to-component* interaction is the classical client-server interaction. The client component uses a service provided by a server component interface. This kind of interaction exists in every component model using the notion of binding.

- The *component-to-aspect component* interaction is managed by an aspect binding. Many technical services such as logging, persistence or transaction can be plugged in the system using aspect bindings towards aspect components. It is worth using aspect bindings when a concern is crosscutting and applies on several components (mostly technical services but not exclusively).
- The *aspect component-to-component* interaction, using a regular binding, is used as an AOSD best practice. In Figure 5 we can see this kind of interaction between the aspect component and various policy components. In this example, changing a transaction policy is performed through a reconfiguration between the aspect component and the components providing policies. This type of interaction illustrates the real role of an aspect component. An aspect component can be seen as a specific connector to integrate a crosscutting concern, which itself is represented as a regular component. It manages the interactions between base concerns implemented with base components and a crosscutting concern also implemented with base components.
- The *aspect component-to-aspect component* interaction can be managed with two types of bindings: a regular binding between two aspect components expresses a collaboration of the two aspects. An aspect binding between two aspect components expresses that the second aspect is woven on the first one. However the consequences of such operations are not debated in this paper.

Figure 4 Aspect domains

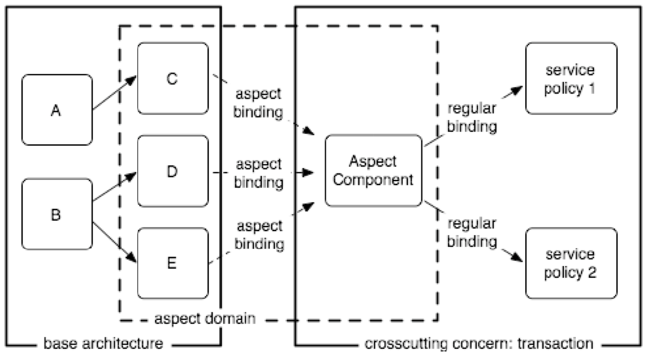


Integration model

Figure 5 shows how crosscutting concerns are integrated to a base application using our model. Two component assemblies are composed together in this figure. The components *A*, *B*, *C*, *D*, and *E* constitute the base application (left-hand side of the figure). The aspect component and the two service policy components represent the crosscutting concern (right-hand side of the figure) which has to be integrated into the base application.

In a full-fledged component approach, the amount of modifications to obtain the same result requires numerous and tricky modifications. For example, the components would have to be stopped, manually adapted and reconfigured, to finally be restarted. Moreover, once integrated to the system, it seems difficult to remove one of these concerns in an easy and proper way. With our approach, the removal of the crosscutting concern is achieved by unsetting the aspect bindings.

Figure 5 Integration model



Compared to an aspect-oriented approach, with our approach, the relationships between aspects and components are reified as aspect domains and aspect bindings which are manipulable entities. In addition, these notions are expressed in terms of component-based notions (components, bindings and composite-components), instead of being implicit in the code of aspects.

2.4 Discussion on software evolution

The main contribution of our model is to bring AOSD notions at the level of CBSD. This is done by a twofold integration of the AOSD and CBSD approaches. Thus, benefits are derived from both approaches. The issue of code tangling and code scattering stated in Section 2.2 is addressed by the composition of aspect components using aspect bindings and aspect domains (comparable to the use of advices and pointcuts in state of the art AOSD approaches). The issue of reification over crosscutting relationships (also stated in Section 2.2) in AOSD approaches is addressed by the use of explicit aspect domains and aspect bindings to represent pointcuts. The remainder of this section elaborates on the solutions we propose to these issues.

Revisiting concern tangling and concern scattering with aspect components

Our approach follows the classical pointcut/advice model of AOSD. The novelty of our approach is to represent these concepts as component-based concepts. Thus, crosscutting concerns are well modularised into aspect components as it is usually addressed by AOSD approaches (Kiczales and Mezini, 2005).

Reflection over crosscutting relationships

In existing AOSD languages, the relationship between an aspect and the objects containing join points picked out by the aspect is explicit in the source code but is implicit at runtime. Indeed, this relationship is structurally defined by a pointcut in the source code, but is lost when the woven code is executed. In our model, we propose to reify crosscutting relationships of a system with aspect bindings and aspect domains.

An aspect domain structurally represents a pointcut. In the context of a reflective component model, aspect domains can be introspected and reconfigured as components using an API (concrete details are given in Section 3). Compared to aspect domains, aspect bindings are more fine-grained entities to show a concrete relationship between an aspect component and a component. There is no real equivalent to this notion in current AOSD terminology. An aspect binding also offers strong reflection capabilities to specify which operations and interfaces of a component have to be aspectised by the aspect component behaviour.

In Ebraert et al. (2005), the authors identify several pitfalls in unanticipated software evolution. One of them is concerned by the finding of dependencies. We have seen that AOSD approaches fail in clearly representing dependencies between aspects and components, especially in asymmetrical approaches. In our model dependencies are fully identified (component dependencies, and aspect dependencies), and are represented as first class entities of the model. Authors also state that static and dynamic information about the architecture of the system is a key feature for software evolution. They argue that reflection or meta-object protocol manipulations are indispensable in the process of software evolution.

Towards a co-evolution between a component architecture and source code

In Kwon et al. (1998), authors survey the state of the art on software maintenance, in particular tools that need to be investigated to cope with legacy systems. The objective of our proposal is not to provide a full support for system maintenance, nor to provide tools for complex software evolution, such as the migration of legacy code. We propose to take into account the process of evolution by providing the *co-evolution* between design models (an architecture of components and aspects) and the source code. This co-evolution becomes possible as soon as strong reflection capabilities are provided by the model to manipulate and reconfigure components and aspect components.

The next section presents the mapping of our model to a general and reflective component model named Fractal. Our extension of Fractal is called Fractal Aspect Component and has been validated with two implementations. These implementations provide runtime weaving of aspect components onto Fractal components by using our concepts of aspect domain and aspect binding, which offer runtime information about the aspect component woven on base components.

3 Mapping onto fractal

This section presents the mapping of the main notions presented in the previous section onto the Fractal component model, which is a general and extensible component model supporting regular dynamic bindings. Our extension of Fractal is called FAC for Fractal Aspect Component. Section 3.1 presents the Fractal component model, and Section 3.2 proposes our extension FAC.

3.1 Fractal: A general and reflective component model supporting dynamic bindings

Fractal is an ObjectWeb[1] consortium project that proposes an extensible and modular component model (Bruneton et al., 2004). This section describes Fractal main features. Note that Fractal is independent of any programming language. Several implementations exist in different languages such as Java, SmallTalk, C, C++, and the languages supported by the .NET platform.

Contrary to component models for application servers such as EJB or .NET, Fractal is a general and reflective component model for developing complex software systems, such as operating systems and middleware. Besides the notion of component, Fractal uses other notions: composite-component (offering different views and abstractions on a system), shared component (a component directly nested by several composite component), dynamic binding (between components). Fractal is a reflective component model and offers introspection (system monitoring), and reconfiguration capabilities (modification of the system architecture).

A Fractal component has two parts: a *content* and a *membrane*. The content of a composite component is built as a set of sub-components, and the content of a primitive component implements its provided services.

A component *membrane* can offer a level of control and a level of interception. The level of control is a set of interfaces to manage the non-functional properties of a component such as life cycle, bindings, content, name, or attributes management. This set can be extended by the addition of new control interfaces to a component membrane. The interception mechanism reifies messages sent by and received on component interfaces. These messages can be modified, discarded or delivered to the component.

An interface is an access point to a component comparable to the notion of a port in several component models, like ArchJava (Aldrich et al., 2002) or CCM (OMG, 2002). A Fractal component offers external and internal interfaces. External interfaces are accessed from the outside of the component, while internal interfaces are only accessible from the composite's sub-components.

A binding is a communication channel between a client interface and a server interface. A client interface uses operations provided by a server interface.

Component assemblies (see Figure 6) can be described with the Fractal Architecture Description Language (ADL), which is XML-based. Figure 7 presents the syntax of the ADL. It presents the architecture description of the example of Figure 3. Lines 2–4, and 9 show the definition of server interfaces (*role* = *"server"*). Lines 4–10s define the component *A* and Lines 11–15 the component *B*. Lines 16–17 are binding declarations of the binding between the server interface *r* of the composite and the server interface *r* of component *A*, and the binding between the client interface *s* of the component *A* and the server interface *s* of the component *B*.

Figure 6 A component assembly

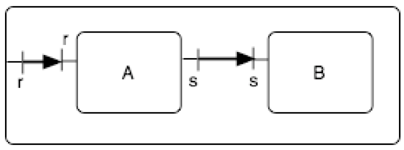


Figure 7 A component assembly (XML code)

```
01 <definition name="HelloWorld">
02 <interface name="r" role="server"
03              signature="java.lang.Runnable"/>
04 <component name="A">
05 <interface name="r" role="server"
06              signature="java.lang.Runnable"/>
07 <interface name="s" role="client"
08              signature="Service"/>
09 <content class="AImpl"/>
10 </component>
11 <component name="B">
12 <interface name="s" role="server"
13              signature="Service"/>
14 <content class="BImpl"/>
15 </component>
16 <binding client="this.r" server="A.r"/>
17 <binding client="A.s" server="B.s"/>
18 </definition>
```

The use of XML files to describe a Fractal architecture and its weaving tasks is not necessary. In Fractal, the XML-based Fractal ADL uses the Fractal API to instantiate components. The Fractal ADL can be seen as a front-end for the API.

In brief, Fractal is a general component model that is implemented in several programming languages. The model is reflective and open and provides a clear separation between the functional properties of a component and its level of control. Given these properties, the Fractal component model appears to be perfectly well suited to the integration of our three main concepts: *aspect component*, *aspect binding*, and *aspect domain*. The next section described how we have performed this integration.

3.2 Fractal Aspect Component (FAC)

FAC is our mapping of the general model exposed in Section 2 on the Fractal component model. As said previously, our notions of aspect component, aspect binding, and aspect domain are represented as component notions. Thus, their mapping to Fractal is quite straightforward. However, some particular elements still need to be defined, such as the advice interface of an aspect component, the interception mechanism used to capture join points.

The remainder of this section describes the *advice component interface* and introduce the *weaving interface*, before discussing implementation issues.

The Aspect Component Interface (ACI)

The Aspect Component Interface (ACI) follows the AOP Alliance API,[2] an open source initiative to define a common API for AOP frameworks. Figure 8 presents the *Advice* Java interface and an implementation example of this interface.

Figure 8 The *Advice* interface

```
/**
 * Interface provided by an Aspect Component
 * to define an advice.
 */
public interface Advice extends
        org.aopalliance.intercept.Interceptor {
/**
 * Define an advice executed around incoming
 * and/or outgoing method invocations reified by m
 * @param m the reification of the method invocation
 * @return the result of the advice
 */
 Object invoke(FcMethodInvocation m) throws
                                             Throwable;
}

/**
 * An example of an AC with before and after code.
 */
public class GenericAC implements Advice {
  Object invoke(FcMethodInvocation m)
                                  throws Throwable {
    System.out.println("before "+m.getMethod());
    Object ret = m.proceed();
    System.out.println("after "+m.getMethod()+
                                 " invoked");
    return ret;
  }
}
```

We have already seen, while defining our join point model that aspect components apply on component methods exposed by client and server interfaces. The context captured at a join point is then related to the context of an invocation of a Fractal interface. Thus, the parameter of the *invoke* method is a reification of a Fractal interface invocation. It provides a set of methods to introspect the join point, to get for instance the name of the component.

The argument of the invocation can also be modified, the intercepted method can be called (*proceed*), and the reference of the intercepted component can be retrieved.

The *proceed* call denotes the original method call. The code written before and after *proceed*() represents the before and after advices of AOSD. If more than one aspect applies on a given join point, the *proceed* call will trigger the next aspect, till the original method code is reached. If *proceed* is omitted the original method call will not apply. This can be useful to prevent, for example, the execution of the intercepted method.

Weaving Interfaces

The *Weaving Interface (WI)* of a component plays a key role in FAC. It manages the weaving of aspect components around the interfaces of the component it controls. In the context of Fractal, we chose to represent the *WI* as a control interface in the component membrane. The *WI* uses the interception mechanism, which is provided by the membrane of components to intercept incoming and outgoing calls on its functional interfaces, and then, delegates the calls to the aspect components bound to (with an aspect binding) these operations. The *weaving interface* in FAC has three main objectives:

- Set/unset aspect bindings to aspect components

```
void setAspectBinding(Component comp,
                      ItfPointcutExp regExp,
                      AspectComponent ac);
void unsetAspectBinding(AspectComponent ac);
```

- Automatically weave an aspect component around a set of components following a pointcut declaration (this weaving task will automatically create an aspect domain, add the components which match the pointcut declaration into this aspect domain, and bind with aspect bindings the aspect component and the impacted components)

```
void weave(Component rootComp,
           AspectComponent ac,
           ItfPointcutExp pExp,
           String aspectDomain);
void unweave(Component rootComp,
             Component ac);
```

- Provide a set of operation to order/re-order aspect components which apply on an interface operation.

```
String[] changeACorder(String acName,
                       int newPosition);
```

In FAC, a component supporting the *weaving interface* is called an aspectisable component. Otherwise, no aspects can be woven to this component. Since the weaving of an aspect component using the *weaving interface* is recursive and traverse the component hierarchy, if the component controlled by the WI is a composite component the weaving is also performed by its sub-components. A weaving operation can be initiated on the system as a whole (top-level composite) or on any sub system (intermediate composite).

All the operations provided by the interface can be invoked either with the Fractal ADL (extended with FAC notions) or directly at runtime.

The following piece of XML code presents the architecture of a Fractal assembly where a directive (tag <weave>) weaves a *traceAC* component (defined lines 2–4) to each component of the composite C (*rootComp=*"*this*" line 12), which has an interface operation starting with "s" and returning "void".

The aspect domain of this weaving will be automatically created and the composite representing this domain will be named "D" (*adomain* = "*D*" line 12}).

```
01 <definition name = "C">
02 <component name = "traceAC"/>
03 <interface name = "ACI" role="server"
04         signature = "AspectComponent"/>
05 </component>
06 <component name = "A"/>
07 <interface name="itf1" role = "client"
08         signature = "Itf1"/>
09 </component>
10 <component name = "B"/>
11 <interface name = "itf1" role = "server"
12         signature = "Itf1"/>
13 </component>
14 <binding client = "A.itf1" server="B.itf1"/>
15 <weaving ac = "traceAC" pcd = "*;*;s*:void"
16         rootComp = "this" adomain = "D"/>
17 </definition>
```

Every reconfiguration operations including the ones of our extension: setting/unsetting of aspect binding, weaving of an aspect component are dynamic operations.

Implementation issues

The mapping of our general model for component and aspect on the Fractal component model has been validated with two different implementations in Java. Our first implementation extends the reference implementation of the Fractal component model in Java called Julia (Bruneton et al., 2004). Julia uses a mixin (Bracha and Cook, 1990) mechanism to program the level of control of components. The second implementation extends another implementation of the Fractal component model in Java, called AOKell (Seinturier et al., 2005), which uses AspectJ (Kiczales et al., 2001) aspects to implement control membranes.

4 Illustration: the Comanche example

This section illustrates the evolution of a component based application to a new crosscutting concern using aspect components, aspect bindings and aspect domains. The application is a minimal HTTP server, called Comanche, which is included in the Fractal distribution.

Section 4.1 shows the decomposition of the application into components. Then, we study in Section 4.2 the evolution of this application to a new requirement, which appears to be a crosscutting concern: a monitoring aspect to manage resources (files and threads).

4.1 The Comanche architecture

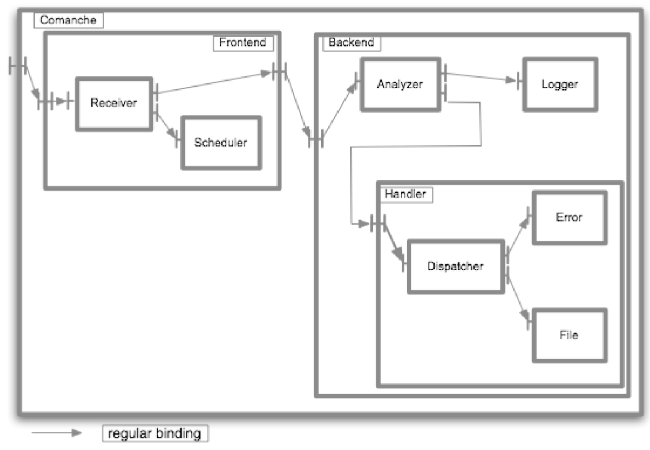
To implement a component-based version of the Comanche web server, we first identify the services of the application, then we associate them to components, and finally we distinguish the relationships and dependencies between components.

We start by identifying the services of the application. Two main services are immediately identified: a request receiver service and a request processor service. Likewise, we can see that the request processor uses a request analyser service, and a logger service, before effectively responding to a request. This response is itself constructed by using a file server service, or an error manager service. This can be generalised into a request dispatcher service that dispatches requests to several request handlers sequentially, until one handler can manage the request.

After the services have been specified, one must assign them to components. In the case of Comanche, we will use one component per service. We therefore have the seven following components: request receiver, request analyser, request dispatcher, file request handler, error request handler, scheduler and logger.

Now that the components have been identified, we determine the dependencies between them, and we organise them into composite components. The topside of Figure 9 shows the resulting architecture.

Figure 9 The Comanche architecture



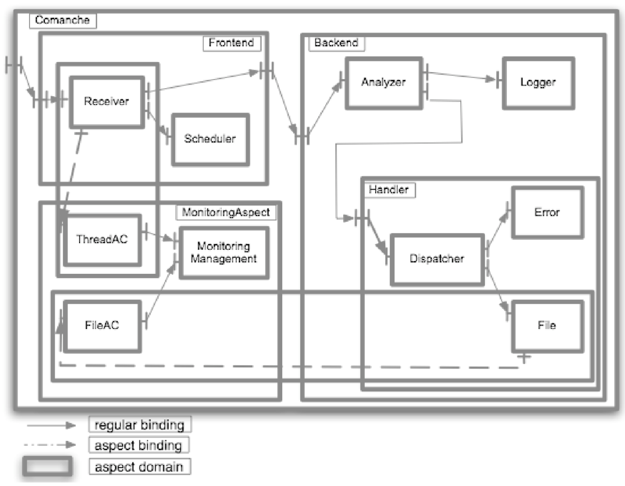
In Fractal the reconfiguration of a component assembly can be achieved at runtime. However if a crosscutting concern has to be plugged to this component architecture, the transformations can rapidly become too heavy to manage. In the case of our particular example, we can see that a crosscutting concern that would impact two or more components would incur too many changes (stop the components, replace the components, and restart the components, and finally reconnect them), and consequently the application would have to be unavailable for a moment.

With aspect components, aspect bindings, and aspect domains, FAC offers an alternative solution to cleanly modularise this crosscutting concern. It brings the crosscutting support that is missing to the component model to manage crosscutting concerns, and it keeps the advantages of Fractal, which offers strong reflection and dynamicity properties.

4.2 A monitoring aspect

In this section we show how to extend the Comanche architecture with a monitoring concern. The monitor controls the number of resources used within the system: the number of running threads, and the number of created files. This concern influences several components, namely *Scheduler*, and *File*. The overall architecture is presented in Figure 10.

Figure 10 The Comanche architecture: a monitoring aspect



We first create an empty composite *MonitoringAspect* that is added to the *Comanche* composite (see Figure 9). This composite represents the monitoring concern we want to add to the system. This composite is composed of several sub-components: an aspect component to intercept the creation of a thread, an other one for the creation of a file, and a regular component, which is notified by both aspect components and manages the resources.

This schema follows the integration model presented in Figure 5. If different thread or file policies are required, the regular bindings between the two aspect components and the *MonitoringManagement* component can be connected to an other component representing another policy.

Once dynamically added to the system, the two aspect components are then woven to the architecture, using the weaving interface of the *Comanche* composite for example. The weaving interface (which is a Fractal control interface) can be called directly at runtime, or by using an XML file for Fractal ADL. In the latter case the XML file will also call the API.

In the following piece of code we can see that two aspect domains are created *threadAD* and *fileAD*.

```
<weave root = “comanche” aspectComponent =
“threadAC”
        pointcutExp = “s;*;*schedule*”
        aspectDomain = “threadAD”/>
<weave root = “comanche” aspectComponent = “fileAC”
        pointcutExp = “frh;*;*"
        aspectDomain = “fileAD”/>
```

Once woven to the application, the aspect bindings are automatically created by the weaving operation. These bindings can be manipulated and unset, dynamically afterwards. The introspection capabilities provided by the aspect control interface of components allow to keep an overview of the domain of application of the two aspect components.

For instance, the weaving interface of the *File* component can give a representation of aspect components that applies on its *filerequest* interface and specifically on its *handleRequest* method. If we weave another AC on the *File* component, we can re-order afterwards the aspect components on this particular method.

The main difference with usual AOSD approaches working with components is that in FAC, pointcut declarations and advices are seen as components and bindings. Indeed, the process of weaving an aspect in FAC consists in the setting of bindings and the adding of a composite-component (called an aspect domain).

5 Related work

In this section, we compare FAC with different kinds of approaches. Firstly, we focus on approaches using a symmetric representation of components and aspects. Then, we study some component models providing support for software evolution.

5.1 Unification of the notions of component and aspect

FuseJ (Suvée, 2005), which is the follow-up project by the JAsCo (Suvée et al., 2003) team, mainly focuses on the nature of an aspect that is represented as a regular bean component. The approach is symmetric: all concerns are implemented as plain components. Components in FuseJ are equipped with gates. A gate is a kind of interface to specify component services: aspect-oriented and regular services. The connectors (extension of JAsCo connectors) specify the types of interaction between gates. FuseJ defines regular and aspect-oriented connectors.

Regular connectors are in charge of functional connections between gates, and aspect-oriented connectors are in charge of weaving a component behaviour to an other component. All the connections defined by a component can be consulted locally. However, FuseJ does not yet propose a global description of a component architecture with its connections. The component is quite limited for the moment. Moreover, so far, only before and after advices are supported.

DAOP-ADL (Pinto et al., 2003) is a component and aspect-based language to specify the architecture of an application in terms of components, aspects and a set of rules between them. As Fractal ADL, DAOP-ADL is a XML-based language. This language is interpreted by the DAOP platform, a dynamic component and aspect platform. Component interactions, with DAOP, are performed through the platform. With Fractal, interactions are fully decentralised and are under the responsibility of each component (more precisely the binding control of each component). In our opinion, this leads to a solution which is more efficient and more scalable.

5.2 Software evolution of components

K-Component (Dowling and Cahill, 2001) is a component model for building context-adaptive applications. The model reifies the structure of the application and describes adaptation contracts written with an Adaptation Contract Description Language (ACDL) to dynamically reconfigure the application. The representation of the architecture is defined with a typed graph. Thus, the reconfiguration of the architecture is performed through a graph transformation. The K-Components are defined using the OMG-IDL3 language and C++ idioms. The main drawback of this approach is that adaptation is always realised through reconfiguration of the component architecture. We have seen in Section 2 that the evolution of an application through reconfiguration is hard. Some crosscutting concerns may not be captured using this process.

SAFRAN (P-C. David, 2006) is an extension of the Fractal component model to support dynamic adaptation of components. Adaptation policies can be dynamically set to individual components introducing a new control interface to manage the setting/unsetting of these policies. This new interface is comparable to the weaving interface in FAC, which manages the weaving of aspect component to the base application. The weaving interface in FAC is more general purpose than the adaptation interface of SAFRAN. Moreover aspect components in FAC can be bound to several components, whereas local adaptation policies are set to each individual components in SAFRAN.

6 Conclusion

In this paper we have presented a general model for components and aspects and its mapping on the Fractal component model called FAC (short for Fractal aspect component). This model improves software evolution by taking benefits from AOSD and CBSD. We realise a twofold integration of both approaches to address the code tangling and code scattering issues inherent in CBSD, which limit evolution, and we leverage the AOSD approach by giving a support for the evolution of aspect.

Our model introduces three main notions: aspect component, aspect domain, aspect binding, which are related to the notions of component, binding, and composite component in component-based models. A crosscutting concern is embodied by a regular Fractal component called an aspect component. We have shown that an aspect component is an encapsulation of an advice code. An aspect domain is the reification of the components picked out by an aspect component. The implicit relationship between a woven aspect component and the component where the aspect component applies is a first-class entity called an aspect binding.

The long-term objective of FAC is to work with aspects at three different levels (Pessemier et al., 2005). The first level is what we have shown with the implementation of the Fractal component model with aspects. The second level is FAC itself with the notions of aspect component, aspect binding, and aspect domain that can be mapped to each implementation of the Fractal component model. Join points at this level are interface invocations on components. And finally, we want to consider a third level, an architectural level, where join points are architectural operations and transformations.

Acknowledgement

This work was partially funded by France Telecom under the external research contract number 46 131 097.

References

Aksit, M., Bergmans, L. and Vural, S. (1992) 'An object-oriented language-database integration model: The composition-filters approach', in Lehrmann Madsen, O. (Ed.): *ECOOP'92: Proc. European Conference on Object-Oriented Programming*, Springer, Berlin, Heidelberg, pp.372–395.

Aldrich, J., Chambers, C. and Notkin, D. (2002) 'ArchJava: Connecting software architecture to implementation', *ICSE'02: Proc. of the International Conference on Software Engineering*, Orlando FL, USA.

Bracha, G. and Cook, W. (1990) 'Mixin-based inheritance', in Meyrowitz, N. (Ed.): *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications Proceedings of the European Conference on Object-Oriented Programming*, ACM Press, Ottawa, Canada, pp.303–311.

Bruneton, E., Coupaye, T., Leclercq, M., Quema, V. and Stefani, J-B. (2004) 'An open component model and its support in Java', *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland.

Burke, B., Khan, K., Rainone, F., Pedersen, S., Fleury, M., Brock, A., Hussenet, C. and Culpepper, M. (2005) *JBoss-AOP*, www.jboss.org/developers/projects/jboss/aop

Dowling, J. and Cahill, V. (2001) 'The k-component architecture meta-model for self-adaptive software', in Yonezawa, A. and Matsuoka, S., (Eds.): *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. LNCS 2192*, Springer-Verlag. Kyoto, Japan, pp.81–88.

Duclos, F., Estublier, J. and Morat, P. (2002) 'Describing and using non functional aspects in component based applications', *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, ACM Press, New York, NY, USA, pp.65–75.

Ebraert, P., D'Hondt, T., Vandewoude, Y. and Berbers, Y. (2005). 'Pitfalls in unanticipated dynamic software evolution', *RAM-SE'05-ECOOP'05 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings*, Fakultat fur Informatik, Universitat Magdeburg, Glasgow UK, 15 July, pp.41–50.

Gudmundson, S. and Kiczales, G. (2001) 'Addressing practical software development issues in AspectJ with a pointcut interface', in Bergmans, L., Glandrup, M., Brichau, J. and Clarke, S. (Eds.): *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, Budapest, Hungary.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. (2001) 'Getting started with AspectJ', *Communications of the ACM*, Vol. 44, No. 10, pp.59–65.

Kiczales, G. and Mezini, M. (2005) 'Aspect-oriented programming and modular reasoning', *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, ACM Press, New York NY, USA, pp.49–58.

Kwon, O., Boldyreff, C. and Munro, M. (1998) *Survey on a Software Maintenance Support Environment,* Technical Report 2/98, Centre for Software Maintenance, School of Engineering and Applied Science, University of Durham, UK.

Lagaisse, B. and Joosen, W. (2005) 'Component-based open middleware supporting aspect-oriented software composition', *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, pp.139–154.

Lai, A., Murphy, G~C. and Walker, R~J. (2000) 'Separating concerns with HyperJ: an experience report', *ICSE'00: Proc. of the International Conference on Software Engineering*, Limerick, Ireland.

Lieberherr, K., Lorenz, D. and Mezini, M. (1999) *Programming with Aspectual Components*, Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA.

Medvidovic, N. and Taylor, R~N. (2000) 'A classification and comparison framework for software architecture description languages', *IEEE Transaction on Software Engineering*, Vol. 26, No. 1, January, pp.70–93.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R. and Jazayeri, M. (2005) 'Challenges in software evolution', *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE 2005)*, Lisbon, Portugal.

Mezini, M. and Ostermann, K. (2003) 'Conquering aspects with Caesar', *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, ACM Press, Boston, USA, pp.90–100.

OMG (2002) *CORBA Components*, v3.0 (full specification), Document formal/02-06-65, June.

Parnas, D.L. (2002) 'On the criteria to be used in decomposing systems into modules', *Software Pioneers: Contributions to Software Engineering*, Springer-Verlag New York, Inc., New York, NY, USA, pp.411–427.

David, P-C. (2006) 'An approach for developing self-adapting fractal components', *SC 06: Software Composition*, LNCS, Vienna, Austria.

Pessemier, N., Barais, O., Seinturier, L., Coupaye, T. and Duchien, L. (2005) 'A three level framework for adapting component-based systems', *Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT05)*, Glasgow, Scotland.

Pinto, M., Fuentes, L. and Troya, J~M. (2003) 'DAOP-ADL: an architecture description language for dynamic component and aspect-based development', *Proc. Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany.

Seinturier, L., Pessemier, N., Duchien, L. and Coupaye, T. (2006) 'A component model engineered with components and aspects', *CBSE '06: Proceedings of the 9h International SIGSOFT Symposium on Component-based Software Engineering*, Springer-Verlag, Vasteras, Sweden, LNCS 4063, pp.139–156, http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html

Suvée, D. (2005) FuseJ website, http://ssel.vub.ac.be/fusej/

Suvée, D., Vanderperren, W. and Jonckers, V. (2003) 'JAsCo: an aspect-oriented approach tailored for component based software development', *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, ACM Press, pp.21–29.

Szyperski, C. (2002) *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Longman Publishing Co., Inc.

Tourwé, T., Brichau, J. and Gybels, K. (2003) 'On the existence of the aosd-evolution paradox', in Bergmans, L., Brichau, J., Tarr, P. and Ernst, E. (Eds.): *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*, Boston, USA, pp.1–5.

Vanderperren, W. and Suvee, D. (2004) 'JAsCoAP: adaptive programming for component-based software engineering', in Lieberherr, K. (Ed.): *3rd International Conference on Aspect-Oriented Software Development (AOSD-2004)*, ACM Press, Lancaster, UK.

Vasseur, A. (2005) Aspectwerkz website, http://aspectwerkz.codehaus.org

Notes

[1]http://objectweb.org

[2]http://aopalliance.sourceforge.net

Websites

http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html

http://fractal.objectweb.org/aokell