Components, ADL & AOP: Towards a common approach

Nicolas Pessemier(1), Lionel Seinturier(1,2)
Laurence Duchien(1)

(1) INRIA Futurs, USTL-LIFL, Team GOAL/Jacquard, Villeneuve d'Ascq, France
(2) Univ. Paris 6, Lab. LIP6, Team SRC, Paris, France
first.last@lifl.fr

Abstract

Keywords: separation of concerns, ADL, component, AOP, Fractal

1 Introduction

Ever since the beginning of computer science, programming language designers have tried to find more and more abstract software artifacts. The goal is to provide programmers with powerful and safe structures to implement their solutions. Approaches such as architecture description languages (ADL) [1], component-based programming (CBP) [2, 3] and aspect-oriented programming (AOP) [4] all go towards that direction. By reifying software assemblies, ADLs provide a clear view of the "software map" of the application. CBP promotes modularity and composability by encapsulating units of code into a capsule that can be deployed, configured, and used through some clearly identified interfaces. Finally AOP provides a clear way for modularizing crosscutting concerns, i.e. functionalities that, with object-oriented programming (OOP) or CBP can not be cleanly localized in a single unit of code and crosscut several objects or components.

In this paper, we argue that each of the three approaches ADL, CBP and AOP, address the issue of software evolution. ADL clarifies the way software entities interact. CBP packages software entities in modules that can be more easily reused than objects. AOP removes from objects functionalities that are out of the scope of their primary concern. Our point of view is that ADL, CBP and AOP, extend object-oriented programming in different ways. Far from being conflicting, these ways are complementary. In this paper, we present our first experiment towards a framework that integrates concepts taken from these three domains. Basically, this framework is based on a component model and provides an ADL to describe software architectures with crosscutting concerns.

The paper is organized as follows. Section 2 reviews the existing project, Fractal, on which we based our approach. Sections 3 and 4 present the extensions we introduce in Fractal to obtain a framework that supports the concepts of ADL, CBP and AOP. Section 5 presents some related works. Section 6 concludes this paper and provides our directions for future works.

2 Background

Building a framework that merges the concepts of ADL, CBP and AOP, is a challenge where many choices are to be made. We can design and implement a whole framework from scratch, or we can rely on some existing works. Section 5 reviews some of them. In order to obtain a first working prototype, we decided to start from the Fractal framework [5] and to extend it. Fractal is in our opinion, the project that is closest to the requirements stated in the previous section.

The remaining of this section presents Fractal, and the next two sections introduce the way we extended it to support crosscutting concerns.

2.1 The Fractal Component Model & ADL

Many component models such as EJB, .Net or CCM exist and receive much interest from both the academia and the industry. These models are however mainly dedicated to coarse grained components for information system-like applications. Classes implementing these components must enforce programming rules, they must be bundled with XML descriptors, and they need to be executed by application servers. They can not thus be handled as easily as objects are handled by virtual machines. Thus, despite of their wide adoption by the community, there is a need for a lighter component model, closer to programming language concepts and that do not require the extra-machinery of the above-mentioned models. The Fractal component model [5] meets these needs. Further, it comes with an XML based ADL.

The Fractal component model [5] allows the definition, configuration, dynamic reconfiguration and clear separation of functional and non functional concerns. Built as a high level model, it put the stress on modularity and extensibility. The Fractal component model is recursive in the sense that a component may be primitive, or composite. In the latter case, this is an assembly that content other primitive or composite components. Components may also be shared between different composites.

Interfaces play a central role with Fractal. There are two categories of interfaces: business and control. Business interfaces are external access points to components. Fractal provides client and server interfaces; a server interface receive operation invocations and a client interface emits operation invocations. Thus, a Fractal binding represents a connection between two components (primitive binding) or more (composite binding). The Fractal model is a strongly typed model. As a consequence, the type of a server interface must be a sub type of the type of a client interface. As their name suggests it, control interfaces provide a level of control on the component they are attached to. These interfaces are in charge of some non-functional properties of the component, for instance its life cycle management, or the management of its bindings with other components.

2.2 Programming with Fractal

This section illustrates with an example the concepts of the Fractal component model. The example in figure 1 modelizes a gas station. Each rectangle is a component. Clients use a gun to fill their tank, and pay at a cash register that is connected to a bank. Each of these elements is a primitive Fractal component. There are two composite components: one for the station and one for the whole system. Composite components are assemblies of primitive and/or (other) composite components. The T-s attached to components are Fractal interfaces[1]. Arrows are bindings between components: they go from a client interface to a server interface.

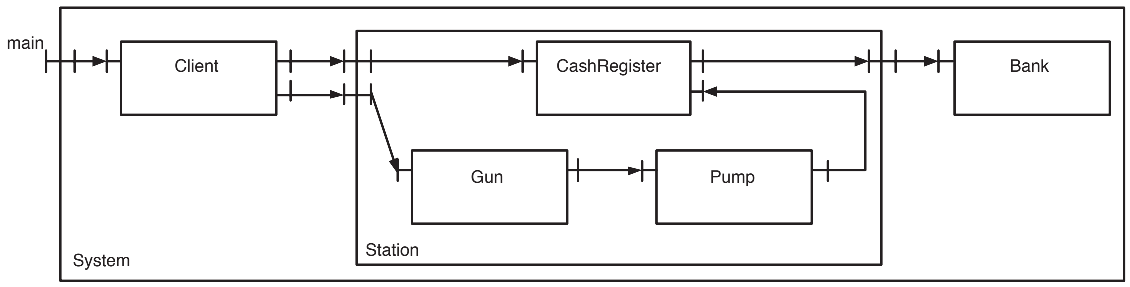

Figure 1: A gas station with Fractal components.

[1]Only business interfaces are specified in the example. Fractal provides the concept of control interfaces that provide a level of control on the component there are attached to.

Fractal provides a Java API to create, introspect and manage the components, their interfaces and their bindings. For instance, components can be started and stopped, and bindings can be created and removed dynamically. The structural definition of components is provided with an XML ADL. Figure 2 provides a piece of the architecture definition for the gas station.

```
<!-- A composite component definition -->
<component name="station">

   <!-- The Station component provides (server role) or
        requires (client role) interfaces -->
   <interface name="gunProvideGas" role="server" signature="station.GunProvideGas"/>
   <interface name="bankAuth" role="client" signature="station.BankAuth"/>
   <!-- other interfaces -->

   <!-- The station component contains the pump, gun & cashRegister components -->
   <component name="pump">
      <!-- ... -->
   </component>

   <!-- The Station component interfaces are bound to other interfaces
   <binding client="this.cashRegisterUserInterface"
            server="cashRegister.cashRegisterUserInterface" />
   <binding client="this.gunProvideGas" server="gun.gunProvideGas" />
   <!-- other bindings -->
</component>
<!-- .... -->
```

Figure 2: The software architecture of the gas station with the Fractal ADL.

3 Rationale for our Project

The software architecture presented in the previous section reifies business dependencies between components. The bindings that have been identified come from the analysis of the business logic of the application. Based on this logic, some crosscutting domains may arise when the application evolves, either because some new unforeseen requirements emerge, or because all the concerns could not have been addressed at a first stage. For instance, a security domain may be needed between the CashRegister and the Bank components. This domain crosscuts the functional domains that have been identified by the business analysis (figure 3 illustrates this).

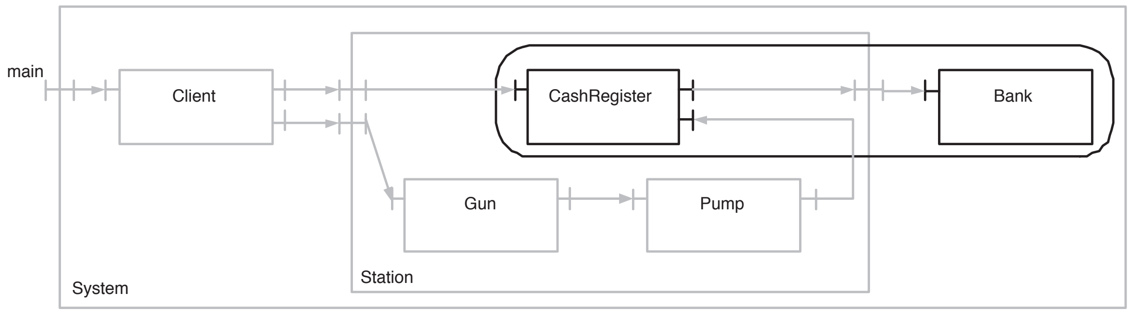

Figure 3: Crosscutting security concern in the gas station example.

Whereas business assemblies and domains are clearly reified in the architecture description of the Fractal ADL, this is not the case with the crosscutting domains: no artifact exists in the Fractal model and ADL to express them. This is true for Fractal, but this is also true as far as we know, for any other ADL or component model.

Hence, the purpose of our project is to extend the existing Fractal model and ADL to take into account crosscutting domains.

4 Extending Fractal to support crosscutting concerns

The extension that we have implemented let us superimpose on an initial business assembly, a level of assembly that corresponds to a crosscutting domain. Hence some new bindings need to be set up. For instance, we want to redirect all outgoing calls from the CashRegister component to perform some encryption functions, and we want to redirect all incoming calls to the Bank component to perform a symmetric decryption function (figure 4 illustrates this).

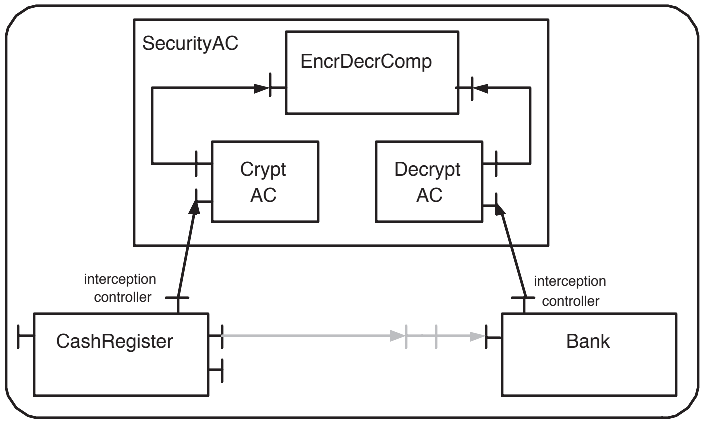

Figure 4: Aspect component for the security concern.

At this point, we need two elements to implement our extension: we need the notion of a component that localizes the definition of the crosscutting concern, and we need a mechanism that weaves the crosscutting concern on a software architecture. Both elements are described below.

4.1 Aspect Components

An Aspect Components (AC)[2] localizes the definition of a crosscutting concern. This is the case of SecurityAC in figure 4. This AC is composed of 3 sub-components: 2 of them, CryptAC and DecryptAC, implement the interception logic associated with the security concern, and the 3rd one, EncrDecrComp, provides the mathematical functions that are required to crypt and decrypt messages. In AspectJ terms, we could say that CryptAC and DecryptAC implement pointcut descriptors, whereas EncrDecrComp implements two pieces of advice.

The CryptAC and DecryptAC components are method interceptors. CryptAC intercepts method calls and DecryptAC intercepts method executions. Their server interface conforms to the AOP Alliance API[3]. They implement the `MethodInterceptor` interface that defines the following method.

[2]The term Aspect Component comes from our previous project, JAC [6], which is a framework for dynamic aspect-oriented programming in Java.

[3]AOP Alliance `<http://sourceforge.net/projects/aopalliance>` is an open-source initiative to define a common API for AOP framework. The API is implemented by Spring and JAC, and soon by DynAOP.

```
public Object invoke( MethodInvocation mi ) throws Throwable;
```

The `invoke` method provides the code that must be run before and after the joinpoint. `MethodInvocation` provides a `proceed` method to execute the joinpoint. To perform the interception, both CryptAC and DecryptAC rely on an interception controller provided by the CashRegister and Bank components. This controller ensures that the outgoing and incoming calls can be reified.

To sum up our approach, a crosscutting concern is implemented with a Fractal component (composite like in our example, or simply primitive) called an aspect component (AC), that provides at least one `MethodInterceptor` interface. No other requirement is needed. The remainder of this section focuses on the way ACs can be woven on top of a software architecture.

4.2 Crosscut bindings

In our approach, weaving an AC is very similar to binding two components together (except that one of them is an AC, and that the other must provide an interceptor controller). There are two ways to establish such a binding, that we call a crosscut binding: either directly, or declaratively with a pointcut expression.

Direct crosscut binding. Starting from the references of the component to be aspectized and of the AC, a crosscut binding is directly created between the two. All methods of the component will then be intercepted by the AC. This binding is similar to establishing a meta-link relation between a base component and a meta-component.

Crosscut binding with a pointcut expression. In this case, a pointcut expression is required. It is composed of three regular expressions that operate on component names, interface names, and method names. All methods that match the three expressions are aspectized by the AC. This weaving is performed by calling a `weaveAC` method on a root business component. This method recursively traverses the hierarchy of sub-components of this root component, and finds all the methods that match the pointcut expression. The technique is similar to the weaving performed by aspect compilers and frameworks.

Note that in both cases, bindings established either directly or with a pointcut expression, can later on be manipulated dynamically: they can be unbound or rebound to modify the crosscut policy of the AC. Hence, they are quite similar to bindings between business components.

5 Related Works

Some recent approaches tried to combine CBP and AOP. This section provides a quick review of some of them.

A first approach that combines CBP and AOP is JAsCo [7] that extends the Java bean model by introducing the notions of *aspect beans* and *connectors*. The *Aspect bean* describes what and where (the notions of *advice* and *pointcut* in AOP) to apply a context independant behavior, using a kind of inner class called the *hook*. The *connectors* are in charge of deploying *hooks* in a specific context of application. A great contribution of JAsCo is provided by its connector language that allows to define a more fine-grained control than AspectJ, on the order the aspects are executed. Finally, we can notice that the management of hooks and connectors is completely centralized and handled by a *connector registry* that have been recently enhanced through HotSwap and Jutta [8] in order to reduce the cost that every dynamic AOP approaches suffers from.

DAOP [9] is a dynamic distributed platform where aspects and components are first-order entities that are composed at runtime by a *middleware layer*. As JAsCo approach, the aspect and component management is centralized and all the information about the architecture and its entities is stored in the *middleware layer*. In a first phase, aspects and components are described with a specific *aspect-component language* that allows interfaces, roles and binding definitions. Then, at runtime components and aspects are concretely bound following the *middleware layer* specifications. The original contribution of the DAOP approach is to give a unique role name to every component and aspect. By this way, communications are done by giving these role names and not by object references.

Jiazzy [10] is an enhancement of the Java language for large scale binary components that are separately compiled and externally linked (*units*). Units are kind of pre-compiled Java classes container that are of two different types: *atoms* (construct from Java classes) or *compounds* (construct from other *atoms* or *compounds*). New behaviors can be added to methods or fields without editing the source code, thanks to a mechanism of *open classes* and *open signature* that are based on mixins. To sum up, Jiazzy separate concerns at the granularity of classes and offers some behavior enhancement with a mechanism of mixins that is less powerful than the AspectJ approach.

JBoss AOP [11] is a project that provides AOP capabilities to EJB applications. JBoss AOP allows to modify an application with aspects, to introduce new features in an application with a mixin mechanism, and to manage some metadata. Advices are programmed as Java classes implementing an interception API. Pointcuts are defined in XML and associate interceptors with the application. The weaving is dynamic with JBoss AOP.

6 Conclusion

ADL, component models and AOP are all concerned about software evolution. Their goal is to empower developers to give them the possibility to build complex systems. Our position in this paper is that the three approaches are complementary. However, as far as we know, they have never been put together into one unified framework. This paper proposes a first step into that direction. We have extended an existing component model and ADL, Fractal (see section 2) with some AOP support for crosscutting concerns. We are then able to describe complex component-based software architecture with aspects.

Aspects in our proposal are components called aspect components (AC), that implement a special meta-interface. Weaving an aspect is then a matter of establishing bindings between business components and ACs. There are two ways for establishing such a binding: either directly between a business component and an AC, or by recursively traversing a hierarchy of composite components and finding components that match a given pointcut expression. These two ways unify

This study served as a proof of concept that merging concepts from ADL, CBP and AOP can lead to a working prototype. However, many features remain to be implemented to obtain a more complete development environment: the pointcut definition language must be enhanced, some API must be defined to support crosscut introspection and some GUI tools are needed to assist developers in specifying their crosscutting assemblies. The integration between ADL, CBP and AOP has been conducted by mapping the concepts of AOP onto the ones of ADL and CBP. It remains to be seen whether the reverse is also true. Finally, as with other aspect languages or frameworks, the weaving of an aspect onto a business application is based on the pointcut expressions. The information contained in these expressions is rather light, and in all cases only concerns the structure on the underlying application. Behavioral pointcut expressions could certainly lead to a more powerful aspect programming environment.

References

[1] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26, May 2000.

[2] C. Szyperski and C. Pfister. Why objects are not enough. In *Proceedings of the International Component Users Conference*, 1996.

[3] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[5] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Workshop on Component-Oriented Programming (WCOP) at ECOOP'02*, June 2002. `http://fractal.objectweb.org/current/fractalWCOP02.pdf`.

[6] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, September 2001.

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

[8] D. Suvée and W. Vanderperren. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of the Dynamic Aspects Workshop (DAW) at AOSD'04*, March 2004.

[9] M. Pinto, L. Fuentes, M.E. Fayad, and J.M. Troya. *Separation of Coordination in a Dynamic Aspect Oriented Framework.* April 2002.

[10] S. McDirmid and W. Hsieh. Aspect-oriented programming with jiazzi. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79. ACM Press, 2003.

[11] B. Burke and al. JBoss-AOP. `http://www.jboss.org/developers/projects/jboss/aop.`