Separation of Coordination in a Dynamic Aspect Oriented Framework

M. Pinto¹, L. Fuentes¹, M. E. Fayad², and J. M. Troya¹

¹Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga
Campus de Teatinos, s/n. cp. 29071 Málaga (SPAIN)
email:{pinto,lff,troya}@lcc.uma.es

²Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE, U.S.A
email:fayad@cse.unl.edu

Abstract

Aspect-Oriented Programming (AOP) separates in a new dimension, named aspect, those features that are spread over different components in a system. In this paper we present a Dynamic AO Framework where components and aspects are first-order entities composed dynamically at runtime according to the architectural information stored in a middleware layer. As an example we describe the coordination aspect, one of the most relevant and useful aspects of our approach, very useful to develop distributed open systems. The main functionality of this aspect is to encapsulate the interaction protocol among a set of components.

1. Introduction

Software Engineering moves towards the development of a system as a composition of components off-the-shelf (COTS). This approach claims the resultant software is more reusable, extensible and adaptable. However, achieving these goals requires an appropriate separation of the system concerns in independent modules what it is not a straightforward task. Commonly, the same concern happens to be spread over different modules creating dependencies among them.

Advanced Separation of Concerns (ASoC) is an attempt to solve this problem. Aspect-Oriented Programming (AOP) [1] [2], HyperSpaces [3] and Composition Filters [4] among others, are some approaches extending Object-Oriented Programming [5] [6] and Component-Oriented Programming [7] [8] with new dimensions of concerns, beyond ”objects” or ”components”. Our work focuses in AOP, where a new dimension, the aspect, is introduced to model those features present along multiple components in a system, that can change or evolve independently from them.

We can identify important differences in current AO technologies, depending mainly on three factors: the aspect definition language, the weaving process and the kind of concerns modeled as aspects. AO languages can be aspect specific, defined explicitly to implement, for instance, the synchronization aspect, or extensions of general purpose languages providing special constructions used to implement any kind of aspects. One drawback of AO languages is that the weaving process is static, mixing the code at compile-time. Although the resultant code is highly optimized, the separation of concerns and its benefits is lost at runtime.

Currently, AO frameworks are considered a promising alternative to AO languages. In AO frameworks both components and aspects are modeled as first-order entities and are implemented in the same general purpose language. One of the main features of AO frameworks is that the composition is performed more or less dynamically at runtime. Despite the fact that static weaving offers better performance, dynamic weaving is much more flexible because of late binding of components.

---

*This research was funded in part by the CICYT under grant TIC99-1083-C02-01, and also by the telecommunication organization ”Fundación Retevisión”*
and aspects involved. In AO frameworks aspects and components entities remain separated in all the software lifecycle, including system execution. This means software more reusable and adaptable where aspects and components can evolve independently and the number and kind of aspects applied to a component can change at runtime.

Another important issue is the kind of concerns that is worthy to separate. Most approaches focus in the separation of system concerns such as concurrency, synchronization, distribution and security. There is no doubt about the benefits of applying AOP to these concerns, but some application domains have particular features that cut across the basis functionality of domain components and should be modeled as aspects to take the advantages of AOP. Actually, much more research is needed in the scope of the definition and implementation of domain-specific aspects.

Our AO approach is a Dynamic Aspect-Oriented Framework (DAOF) [9], where components and aspects are first-order entities dynamically composed at runtime using the composition information stored in a middleware layer [10]. Though our approach is generic enough to separate any kind of aspects, we are interested in domain-specific aspects, where our application domain is Collaborative Virtual Environments (CVEs) [11]. In CVEs geographically dispersed users join a shared space that integrates all the resources they need to collaborate with other users as if they were co-located. We model the main features of CVEs, awareness, persistence, authentication, access control and multiples views, as aspects that are composed at runtime with components modeling rooms, users, collaborative tools and documents. Component and aspect interfaces are detached from their implementation classes, so that the framework pick at runtime the implementation module that is adequate to the current context, and replace it as context conditions vary. For instance, a CVE can be adapted to user preferences, by using different implementations of the multiple view aspect for each user.

The coordination aspect is one of the most interesting and useful aspects offered by our proposal. The main functionality of this aspect is to encapsulate the interaction protocol among a set of components, and keeping the information required to address the target components. The main advantage is that components do not need to know each other during interaction, increasing their reusability in different contexts. Additionally, the coordination aspect may change dynamically the component instances involved in a dialogue, making the system more adaptable and configurable. Currently we are experimenting with a working prototype that is being applied successfully to CVEs.

The organization of this paper is as follow. Section 2 presents our AO framework approach, highlighting the requirements in the definition of components and aspects and the dynamic composition at runtime. Section 3 introduces the separation among computation and composition and the use of the coordination aspect to achieve this separation. Finally, section 4 discusses our conclusions and future work.

2. A Dynamic AO Framework

We propose a Dynamic Aspect-Oriented Framework (DAOF) where components and aspects are first-order entities dynamically composed at runtime using the composition information stored in a middleware layer. We use Java as the general-purpose language to implement both components and aspects. The framework is generic enough to separate any kind of system or domain concerns. However, we focus our work in the separation of domain-specific concerns, since we think that little research is being done in this area. The aspects we define and implement are features of the CVE application domain, like awareness, persistence, authentication, access control and multiples views that crosscut the functionality of system components. Currently, Web application in general and CVEs in particular demand the instantiation of custom environments according to the user profile. Our framework advocates the dynamic configuration of distributed applications through the separation of the general architectural restrictions common to all the users from the user specific configuration [12].

The DAOF has two layers, the component-aspect layer with the components and aspects existing at runtime and the middleware layer in charge of composing dynamically these entities based on a set of dynamic composition rules. Figure 1 shows the UML profile of the DAOF design, and figure 2 shows the two layer structure with the coordination aspect as an example. As we can see in Fig. 1 the dynamic composition is performed through the components LocalUserSite and LocalEnvironmentSite that reside in the middleware layer. The LocalEnvironmentSite maintains the architectural constraints common to all users, defined by the components and aspects that may engage the system and composition rules describing which aspects must be applied to components and the application order. The LocalUserSite customize the system for each user according to his or her user profile. For instance, if different implementations of the same aspect are possible, the LocalEnvironmentSite has the list of all of them while the LocalUserSite holds the concrete implementation that has to be used for an specific user.

In order to facilitate the implementation of components and aspects, the framework provides two base classes, Component and Aspect with the attributes and methods that the middleware layer needs to create components and aspects, and to compose them at runtime. Since communication between components is performed through the middleware layer of the framework,
Component and Aspect classes maintain a reference to the `LocalUserSite`. DAOF components must extend the `Component` class and DAOF aspects must extend the `Aspect` class (Fig. 1).

Regarding to component communication, when a component propagates a message, the middleware layer intercepts it and evaluates the corresponding aspects. The `LocalUserSite` component offers primitives for component composition, definition of architectural constraints and aspect evaluation. The architectural constraints are encapsulated in the `LocalEnvironmentSite` component that then contains which aspects must be applied for each component. The framework defines methods to add this information to the middleware layer.

If the evaluation of output aspects fails the middleware layer throws an exception to the source components, otherwise the message is sent to the target components. The framework provides different ways of addressing the target components. The `components and aspects definition` describes different kinds of component addressing available and the message propagation primitives implemented by the middleware layer. It also defines a set of design patterns that may be used in the implementation of the framework entities. In the rest of this section we describe both the `component aspect definition` and the `composition rules`.

2.1. Component and Aspect Definition

The `components and aspects definition` describes how components and aspects are designed in our DAOF, what are the different kinds of component addressing available and the message propagation primitives implemented by the middleware layer.

The requirements in the design of DAOF components are mainly two. One, components must have a reference to the `LocalUserSite` in the middleware layer and two, they must use the message propagation primitives implemented in the middleware layer in their interaction with other components. As we mentioned before, the `Component` class is provided to accomplish these restrictions and simplify the developer task. As long as the components extend this basic class, the developer can follow any design pattern in their implementation. In [9] we propose but not impose the use of the `Role Object` pattern where a component can play different roles that can be added and removed dynamically and this issue promotes the reuse of the basic components in different contexts.

Due to the diversity of concerns that can be implemented as aspects and their different behaviour, we do not propose any
design pattern for DAOF aspects, that can be implemented at the user taste. The only restriction is the extension of the base class \textit{Aspect} that provides a reference to the \textit{LocalUserSite} and the interface called by the middleware layer in the aspect evaluation (method \textit{eval()} in 1).

The main purpose in the definition of components and aspects is to make the entities in the system as independent as possible, to reduce their dependencies and to increase their reusability in the development of different applications. However, components participate in interaction with other components and need to address them in order to accomplish their task. We think that the use of the traditional message passing in object-oriented programming, where a component maintains explicit references to other components, it is insufficient to achieve our goal of independence. So, we realize that new message delivery mechanisms are needed in open systems. Our framework offers four kinds of message delivery mechanisms that avoid direct references among components: role-based invocation, identifier-based (CID) invocation, instance-based invocation and interface-based invocation.

The role-based invocation addresses a component based on the role the component plays in the system. For instance, in a CVE all the components modeling the place where users join to collaborate have the role room. Different components with the same role can be instantiated simultaneously in the system. The identifier-based invocation addresses a component by its unique identifier CID. Each component in the system has its own and unique CID, however the CID of a component is only known if a previous message has been receive from that component. The instance-based invocation allows to address a set of components representing the same resource. For instance, in a CVE each component with the role ”room” has an instance identifier to differentiate them. The interface-based invocation addresses a component based on its interface, that determines in which interactions the component can be involved.

Using the above message delivery mechanisms we detach components from other components. It is also important to detach components from aspects and aspects from other aspects. Components are detached from aspects since they have no knowledge about the number and type of aspects they are affected by. This provides enough flexibility to apply different aspects depending on the context the component is used. Even more important, the aspects applied to a component can change dynamically at runtime adapting the system to new requirements or user preferences. In addition, there is no explicit joint points in the aspect definition. So, the aspects are also independent from the components they affect, being able to apply them to different components in different time.

Finally, the aspects do not have information about the rest of aspects applied in the system. There can be dependencies in the order of application, but the aspects are not aware of them. For instance, in a CVE the \textit{authentication} aspect must be always applied before the \textit{multiple views} aspect because it is needed to identify the user before configure the component according to his or her user profile, however nor the \textit{authentication} or the \textit{multiple views} aspects have any reference to this relationship that it is stored in the middleware layer.

The message delivery mechanisms described above are used to address the target component in the set of primitives the middleware layer implements to enable the programming of distributed application. Concretely, three primitives are defined: \textit{execute}, \textit{broadcast} and \textit{notify} with four parameters: the source component CID that identify uniquely the origin of the message, the target components, the message and its arguments.

1. \textit{execute(source.CID, target.component, message, args)}. This mechanism allows the delivering of a message to a local component or a unique instance of a remote component. The invocation mechanism can be role-based, CID-based, instance-based or interface-based. If a local component is found with the specified identifier the message is sent to it. In other case, it is propagated through the middleware layer to other nodes looking for a remote component.

2. \textit{broadcast(source.CID, target.component, message, args)}. This mechanism is used to broadcast the same message to different components. The target components are addressed using the role-based, instance-based or interface-based invocation mechanism.

3. \textit{notify(source.CID, target.component, message, args)}. This mechanism is used to propagate a message to all the instances representing the same resource in the system. The invocation mechanism will be instance-based. This mechanism is very useful in CVEs where a component is replicated among different nodes representing the same resource. For instance, all the users connecting to the same “room” will have a local representation of the room sharing the same instance identifier.

2.2. Composition Rules

The composition rules are maintained in the middleware layer by a third entity that dynamically composes components and aspects at runtime. As we mentioned before, this component is modeled using two classes, the \textit{LocalUserSite} and
Components are computational entities that encapsulate data and computation. However, components are not isolated entities, they usually interact with each other to accomplish a certain task. The separation of data processing from coordination patterns is proved to be a good approach [13]. With this separation, components are passive entities characterized by the complete ignorance of how propagated messages influence the execution of an application. This component can be reused in different context participating in different interaction. The component interaction and coordination protocols are encapsulated in another entity. In our system this entity is the coordination aspect.

Using the coordination aspect, a component do not need to address the target component. The component can send a message using one of the middleware layer primitives described above - execute, broadcast or notify, setting to null the target component parameter. The coordination aspect has the information to decide which component or components must receive the message.

For instance, in a CVE a room contains the set of documents the users in that room are working on. There is also a set of individual and collaborative tools used to work over that documents. When a user clicks in one of the documents a tool able to manage it must be opened. However, the kind of documents a room can contain it is unpredictable, and the room component should not explicitly address the tool used to open the document. This can be achieved applying a coordination aspect that contains the information needed to know which tool need to be open according to the document extension. In Fig. 2, when the component room use the execute primitive in the middleware layer it set to null the target component parameter. The composition rules stored in the middleware layer state that the coordination aspect has to be evaluated when
the component room sends the open message. During the evaluation the coordination aspect determines that the component with role JPEGEditor is used to open files with extension .jpg and the open message is sent to this component.

Even addressing the target components there are situations where the coordination aspect can be very useful. Though a component can use the CID as a way to uniquely address other component, the CID is only known if a previous message has been received from that component. In addition, the components are more reusable if the programmer design the system having in mind the roles each component plays in the interaction and not concrete instances. This can be achieved using the role-based, instance-based or interface-based invocation mechanisms. The inconvenience of using these mechanisms is that more than one component can be candidates to receive the message. Depending on the application logic all of them or only part of them should actually receive the message and the component might not be aware of that. So, the coordination aspect solves the problem linking source and target components transparently.

An example of the use of the coordination aspect in this situation can be the following. In the development of CVEs, we detach the component model from the component view implementing them in two separate components. This means that in order to model a room we have the modelroom component and the viewroom component. According to the component-aspect model described above these components cannot have explicit references among them. Both of them have an unique CID but they must initiate the interaction using another invocation mechanism because the CID for a component is only known if a previous message has been received from that component. Supposing a CVEs with different rooms instantiated simultaneously for a user, this means that there are different modelroom components and different viewroom components. When, for instance, a user interacts with the viewroom and a message is sent to the modelroom, there is no way to know which modelroom component must receive the message unless we use a coordination aspect that matches pair of modelroom and viewroom.

In Fig. 3 we have two modelroom and two viewroom components, each one with its own CID. They interact with each other using the execute primitive addressing the target component by its role. Initially (step 1 in Fig. 3) the coordination aspect do not have any information about the components. Lets suppose that the modelroom component with CID1 invokes the execute primitive. The coordination aspect gets in the middleware layer the CIDs of all the components with role viewroom, takes one of them, for instance CID3, and matches CID1 and CID3 (step2 in Fig. 3). From now on, when the component modelroom with CID1 sends a message to the viewroom component, the coordination aspect change transparently the invocation mechanism from role-based to CID-based sending the message to the component with CID3. The same occurs for the other pair of components (step3 in Fig. 3).

The interaction protocols implemented in the coordination aspect can be as simple as organize the communication among a set of components. For instance, in a CVE when a user enters in a room, the coordination aspect can decide to send the newUser(user) message to all the components whose interface implements this method. However, more complicated interactions are possible. The coordination aspect can encapsulate the state transition diagram of a protocol defining different
states in the interaction. Suppose a CVE where each user has its own office and to enter in one of these rooms the owner must be there. We assure that using a coordination aspect with two states open and close, initially in the close state (Fig. 4). If the owner is not in the office the coordination aspect is in the close state and it sends the reject message if a user tries to login in the room. Otherwise, the coordination aspect is in the open state and it sends the accept message.

The coordination aspect also provides a solution to the problems derived from the integration of COTS. It can play the role of an adapter to compose components that initially are incompatible due to differences in their interfaces. The variations in the interfaces can be due to different message names, arguments or types. In addition, the coordination aspect can hide the fact that the service used by a source component is actually implemented from a combination of components and not only one.

For instance, in Fig. 5.a the application has a component CollabTool that integrates all the collaborative tools in the environment - whiteboard, chat and application sharing. When the component room wants to open one of them sends the message open(toolname). Suppose now that we want to develop a new application reusing the component room, but where the collaborative tools are implemented in independent components: WhiteBoard, Chat and SharedAppl. In this case, the interfaces do not match and the coordination aspect is needed to adapt the target component name and component message, that will be different depending on the value of the toolname parameter.
4. Conclusions and Future Work

We consider AO Frameworks a better approach than AO languages, especially in the development of AO applications which need to be highly customizable, because the separation of components and aspects in all the software lifecycle and the dynamic composition at runtime allow the construction of software more flexible, reusable and extensible.

In this paper we have presented the main characteristics of our DAOF approach. The component-aspect model describes how components and aspects must be designed and implemented in our framework, the message delivery mechanisms and the primitives offered by the middleware layer to facilitate the programming of distributed applications. The composition model describes the information a developer needs to provide to deploy an application in our framework and how this information is stored in the middleware layer.

As an example we have presented the coordination aspect, that provides separation of data processing from coordination patterns. The coordination aspect encapsulates interaction protocols of different complexity, detach components avoiding explicit references among them, and acts as an adapter to solve the problems derived from COTS integration.

Our future goals is to complete the implementation of our DAOF framework and the definition of an application framework for the development of CVEs. We plan to complete the design and implementation of the coordination aspect and use it in different contexts as part of the working prototype we are building. Concretely we are developing a virtual office as part of a funded research project.

References


