# Separation of Coordination in a Dynamic Aspect Oriented Framework [*] [†]

M. Pinto[1], L. Fuentes[1], M.E. Fayad[2], J.M. Troya[1]

[1]Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga
Málaga (Spain)
{pinto,lff,troya}@lcc.uma.es

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

## ABSTRACT

Aspect-Oriented Programming 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 software 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, essential to develop open distributed systems. The main functionality of this aspect is to encapsulate the interaction protocol among a set of components.

## Keywords

Aspect-Oriented Programming, Component-Oriented Programming, Dynamic Composition, Coordination Aspect

## 1. INTRODUCTION

Software Engineering moves towards the development of systems in terms of a set of interactions among more or less detached components. This component-oriented approach results in more *reusable*, *extensible* and *adaptable* software. However, achieving these features requires an appropriate separation of the system concerns in independent modules which is not a straightforward task. Commonly, the same concern happens to be spread over different modules creating undesirable dependencies among them.

Advanced Separation of Concerns (ASoC) is an attempt to solve this problem, extending Object-Oriented Programming and in some cases also Component-Oriented Programming [2] with new dimensions of concerns, beyond "objects" or "components". Aspect-Oriented Programming (AOP) [6] [8] is a promising ASoC methodology that introduces a new dimension, the *aspect*. *Aspects* model those features presented along multiple components in a system that may change or evolve independently from them.

Current AOP technologies offer different alternatives to tackle the separation of concerns issue, that differ mainly in three factors: the aspect definition language, the weaving process, and the kind of concerns modelled as aspects. AO languages can be *aspect specific*, defined explicitly to implement one type of aspect (for instance, the *synchronization* aspect) or extensions of general purpose languages providing special constructions used to implement any kind of aspects [7]. One drawback of AO languages is that the weaving process is static, mixing component and aspect code at compile-time. Although the resultant code is highly optimized, the separation of concerns and its benefits are lost at runtime.

One promising alternative to AO languages are AO frameworks [3] [12], specially those that incorporate the component option. Usually, those AO frameworks model components and aspects as separated entities which 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 between components and aspects. In AO frameworks aspects and components entities remain separated in all the software lifecycle, including system execution. This means that the resulting software is more *reusable* and *adaptable*, where aspects and components can evolve independently.

Another important issue is the kind of concerns that is worthy to separate. Most approaches focus in the separation of distributed 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 basic functionality of domain components and should be modelled as aspects to take the advantages of AOP. Currently, much more research is needed on the definition and implementation of what we call domain-specific aspects.

Our AO approach is a Dynamic Aspect-Oriented middle-

[*]This research was funded in part by the CICYT under grant TIC99-1083-C02-01, and also by the telecommunication organization "*Fundación Retevisión*"

[†]Proc. of the First International Conference on AOSD, April 2002, pp. 134-140, ACM Press, The Netherlands


*AOSD* 2002, Enschede, The Netherlands

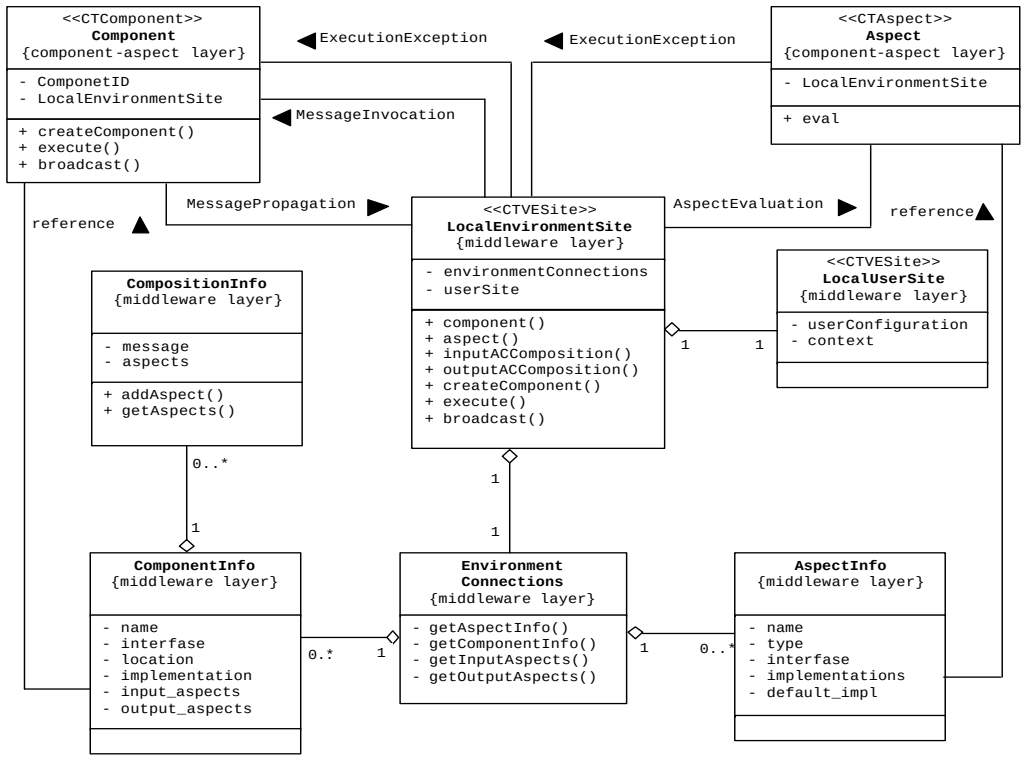


**Figure 1: UML design of the Dynamic Aspect-Oriented Framework**

ware Framework (DAOF). In DAOF components and aspects are first-order entities dynamically composed at runtime using the composition information stored in a middleware layer [11]. We use Java as the general-purpose language to implement both components and aspects. Our approach provides the basis to separate any kind of aspects, however we are interested in domain-specific aspects. Our application domain is Collaborative Virtual Environments (CVEs). In CVEs, geographically dispersed users join a shared space that integrates all the resources they use to collaborate [1] [13]. We model the main features of CVEs, *persistence*, *authentication*, *access control*, *awareness* and *multiples views*, as aspects that are composed at runtime with components modelling *rooms*, *users*, *collaborative tools* and *documents*. An extended description of all these aspects and components can be found in [9].

Features like *persistence*, *authentication* or *access control* are in the scope of any distributed application, while others like *awareness* and *multiple views* are more specific of collaborative domains. In order to make possible the collaboration of dispersed group members in collaborative environments, it is fundamental to provide *awareness* about users location, the activities they are engaged and the documents they work with. Users should also perceive *multiple views* of the environment depending on their preferences or resource availability. This results in different views of the same component that should be changed dynamically at runtime. The separation of these domain-specific features is sometimes obviated and in consequence component crosscutting is not completely avoided.

An important contribution of our approach is that component and aspect interfaces are detached from their implementation classes. That is, as context conditions vary, the framework picks at runtime the implementation module that is adequate to the current context. For instance, a CVE can be adapted to *users preferences* using different implementations of the *multiple view* aspect. The class name corresponding to a certain aspect is stored in the framework middleware layer, in particular inside an object that refers the *architecture of the application* (AA).

After describing the main characteristics of our approach, we will focus in the *coordination aspect*. The *coordination* aspect is one of the most interesting and useful aspects offered by our proposal. This aspect encapsulates the interaction protocol among a set of components. The main advantage of this aspect is that components do not need to know how to interact with external components, increasing their *reusability* in different contexts. 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 DAOF approach, a component-aspect model with dynamic composition. 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

Currently, Web applications in general, and CVEs in particular, need the instantiation of custom environments adapted to *user profiles*. In a component-aspect based development as we offer, this customization implies applying different number and type of aspects for each user. In addition, the complexity of nowadays systems imposes the challenge of

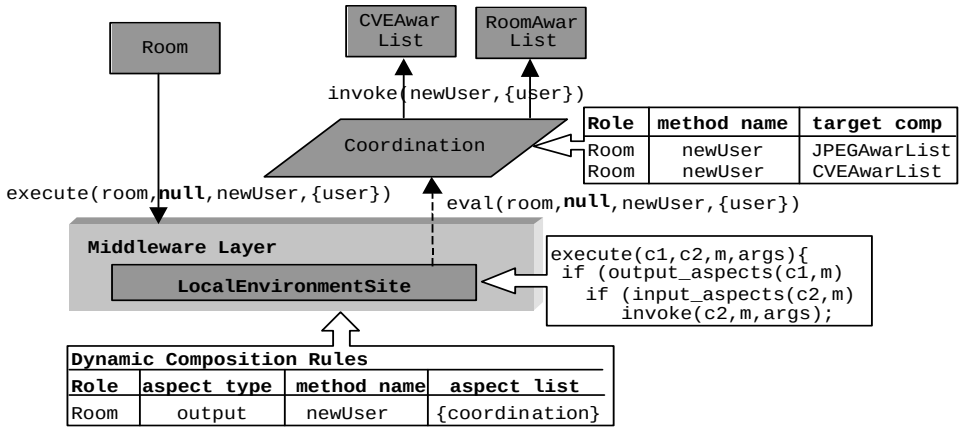


**Figure 2: Component computation and coordination detachment**

making highly *reusable* and *adaptable* components and aspects to avoid their development from scratch. Components-aspects interaction information is usually hard-coded inside components creating dependencies among them, which reduce their reusability. In order to avoid these dependencies we put the AA, that is, components, aspects and weaving information, inside the framework middleware layer.

The *middleware layer* is in charge of composing dynamically components and aspects based on a set of dynamic composition rules and the AA information. In the rest of this section we present how components and aspects must be developed in our DAOF and the mechanisms offered by the middleware layer to define and store the AA used to perform the dynamic composition of components and aspects at runtime. Figure 1 shows a UML class diagram with the *CTComponent*, *CTAspect* and *CTVESite* (part of the middleware layer) stereotypes, the three main entities of our model.

## 2.1 Application Architecture

The architecture of an application describes which components set up the system and how they interact to accomplish the required functionality. Normally this architecture is spread over the system because components have direct references among them for their communication. With the aim of decoupling components and aspects the information about when and how to apply aspects to components it is not hard coded, but it is explicitly set aside inside the AA.

In order to take out direct references from components and aspects code we assign a unique and universal *role name* to name them inside a particular context. Components and aspects with the same role are supposed to provide the same behavior, so we are able to replace them by equivalent components or aspects. Using the role concept we do not enforce components and aspects to implement exactly an interface, we only require them to offer a set of required methods. The benefits of this approach is that a component or aspect role can be provided by different interfaces and implementations, letting developers configure an application customizing the generic AA at design or adapting it dynamically at runtime.

The AA of a system is defined by the list of components and aspects that can be instantiated in the system and a set of *Architectural Restrictions* (ARs) which are explained explained further on. Each component and aspect inside the AA of a particular application configuration is defined by a *role name*, an *interface* and an *implementation* class name. Interfaces are detached from their implementation classes, which may evolve independently at runtime. Aspects are different from components in that the AA holds several implementations, each one corresponding to a user preference. Regarding the number of aspect instances that must be created at runtime, framework users can choose one of the following alternatives:

- *environment-oriented*. There is only one instance of an aspect inside the system. An example can be the *persistence* aspect in charge of registering all the logins and logouts in the environment.

- *user-oriented*. Each user has his/her own instance of the aspect, which is shared among all components collocated at the user site. As an example, the framework creates a unique instance of the *multiple view* aspect for each user, according to his/her visual preferences (2D or 3D representation, for instance).

- *type-oriented*. One instance of an aspect will be shared by the set of components that play the same role. An example is having different *access control* mechanisms depending on the component type. For instance, the access to *documents* is controlled by a LDAP Directory Service while the access to *rooms* is decided asking permission to the room owner.

- *component-oriented*. There is an aspect instance per component instance. For example, different components instances with the same role *room* could have different *access control*, depending again on the room owner preferences.

It is important to remark that this information is part of the aspect description inside the AA, because the middleware layer will create aspect instances according to this information. Completing the AA information, the ARs state which aspects apply to each component, if they must be applied before and/or after the execution of a component method, dependencies among aspects and the application order. An outstanding feature is that the number and type of aspects that are applicable to a component can change dynamically.

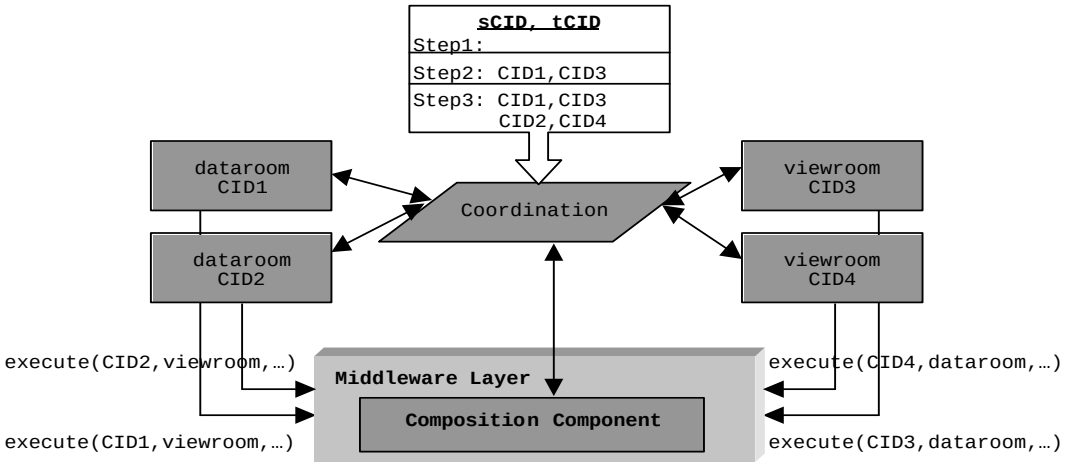


**Figure 3: Component matching using the coordination aspect**

## 2.2 DAOF Components and Aspects

The aim of our proposal is to make the entities of the system as independent as possible, to reduce their dependencies and to increase their reusability across different applications. However, components interact with other components, so they need the receiving component address (reference) to put it in an output message or method invocation. But the use of the traditional message passing in object-oriented programming, where a component maintains explicit references to other components, is pernicious 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 the use of direct references among components: *role-based invocation*, *identifier-based invocation*, *instance-based invocation* and *interface-based invocation*.

The *role-based invocation* addresses a component by the role name that the component plays in the system. For instance in a CVE, components that model a place that users join to collaborate have the role *room*. So, when a user requests to enter into a room, a message with *room* as the destination address is delivered. The *identifier-based invocation* addresses a component by its unique identifier named CID (Component IDentifier). Each component in the system has its own CID. However, the CID of a component can only be known if a message from that component has been received before. Therefore, its use is restricted to this case. On the other hand, the middleware layer assigns the same identifier to those components representing a resource that is replicated through different nodes. The *instance-based invocation* is used when every instance must react to the same events. This mechanism is very useful in CVEs. For instance, all users connected to the same "room" have a local representation of that room, all sharing the same *instance identifier* (e.g. *demoRoom*). Updates to a *demoRoom* instance (e.g. the addition of a new document) are notified to the rest of them by sending a message with *demoRoom* as the target address. Finally, the *interface-based invocation* addresses a component by its interface, which determines in which interactions the component can be involved.

Using the above message delivery mechanisms we detach the component that sends a message from the receiver component. Likewise it is important to detach components from aspects and aspects from other aspects. Components are not aware of aspects since they have no knowledge about the number and type of aspects they are affected by, and even if they are affected by any aspect. This provides enough flexibility to apply different aspects depending on the context the component is being 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, as we said above they are defined in the AA stored in the middleware layer. So, aspects are also independent from the components they affect, being able to apply them to different components in different times.

Finally, aspects do not have any information about the rest of aspects applied at the same time to a component. There could be dependencies in the order of application or due to non-orthogonal aspects, but the aspects are not aware of them. For instance, in a CVE the *authentication* aspect must always be applied the first, but it does not have any references to the following aspect. This aspect interconnection is specified in the AA and stored in the middleware layer. In addition, if an aspect needs any information generated previously by other aspect it will get this information from the middleware layer also. That is, aspects output information is stored inside the middleware layer, ready to be retrieved by any other aspect that needs it. So, in our approach aspects never interact directly.

In order to facilitate the implementation of components and aspects, the framework provides two basic classes, *Component* and *Aspect*. DAOF components must extend the *Component* class and DAOF aspects must extend the *Aspect* class (figure 1). Since communication between components is performed through the middleware layer, these classes maintain a reference to the middleware layer, to interact with other components and to create new components, simplifying the developer task.

The *Component* class in Figure 1 has a reference to the *LocalEnvironmentSite* component in the middleware layer (see next section), and its most relevant methods are *createComponent*, *execute* and *broadcast*. The *createComponent* is used to create new components with a *role name* and an *instance name*. The *execute* and *broadcast* methods are used

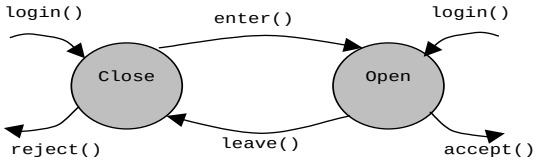


**Figure 4: State Diagram Interaction Protocol for a coordination aspect**

for message delivery and will be explained in the next section.

Likewise, the *Aspect* class in Figure 1 has a reference to the *LocalEnvironmentSite* component of the middleware layer. The main method in the *Aspect* interface is *eval*, that will be invoked at runtime by the middleware layer to perform the aspect evaluation.

## 2.3 Middleware Layer

The dynamic composition of components and aspects is performed through the *LocalEnvironmentSite* component of the middleware layer (figure 1). Each user is represented inside the system by an instance of this component that maintains a local copy of the AA. As we can see in Figure 1, the *LocalEnvironmentSite* is a composition of two main classes, the *LocalUserSite* class and the *EnvironmentConnections* class.

The *LocalEnvironmentSite* stores the AA with the ARs that are common to all users. This general AA can be customized for each user according to his or her *user profile* and this information is stored inside the *LocalUserSite* class [10]. For instance, if different implementations of the same aspect are available, the *LocalEnvironmentSite* has the list of all of them while the *LocalUserSite* holds the concrete implementation that was chosen for a specific user.

The *EnvironmentConnections* class stores the AA. The information about components and aspects is maintained in the *ComponentInfo* and *AspectInfo* classes respectively. For each component the *CompositionInfo* class stores the list of aspects applied to each method in that component and the application order. We define two different kinds of rules: *input-aspects composition rules* and *output-aspects composition rules*. In the former, aspects are applied before the execution of the method, and in the latter aspects are applied after the execution of the method. The interpretation of these rules is as follow: "Aspects will be applied to a component before or after the execution of the indicated method and in the specified order".

Regarding to component communication, when a component sends a message, the middleware layer intercepts it and evaluates the corresponding aspects. If the aspect evaluation fails the middleware layer throws an exception to the source component, otherwise the message is sent to the target component. In order to maintain again the independence among components and aspects, components catch always a general exception, so the aspect who elevates the exception is unknown for the component.

Components send messages by invoking *execute* and *broadcast* methods of the middleware layer with four parameters: the source component CID that identifies uniquely the source of the message, the destination component, the message name and its arguments. The destination component can be indicated according to the already mentioned mechanisms for addressing target components, (they were *role name*, *CID*, *instance name* or *interface*).

1. $execute(sourceCID, targetComponent, message, args)$. This method performs the delivering of messages to a local component or a unique instance of a remote component. If the destination component is local is sent directly to it. In other case, it is propagated through the middleware layer to other nodes, looking for a remote component.

2. $broadcast(sourceCID, targetcomponent, message, args)$. This mechanism is used to broadcast the same message to all the components addressed as target components.

# 3. SEPARATION OF THE COORDINATION CONCERN

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 according to a coordination protocol. Coordination protocols usually crosscut all participant components with the result of a high coupling between coordination and computation code. The separation of data processing from coordination patterns is proved to be a good approach in component-based frameworks [4] and other areas like agent environments [5]. With this approach, components are considered passive entities characterized by the complete ignorance of how output messages influence the application execution. Thus, components can be reused in different contexts and engaged in different interactions. Component coordination protocols has been moved to another entity, that in our system is the *coordination aspect*.

The *coordination aspect* is used in our framework to solve component incompatibilities and integration problems. We are going to illustrate different applications of the *coordination aspect* for the CVEs domain.

Usually open systems require that unknown components can be added to a system at runtime. DAOF copes with the difficulties of coordinating off-the-shelf components (COTS) by providing communication primitives with no destination address, like events. These are the *execute* and *broadcast* methods where the *target_component* parameter is set to null. Now, output messages do not contain any information about destiny, they are intercepted by a *coordination aspect* that decides which component or components must finally receive the event.

For instance, users that want to collaborate should be easily found inside a CVE. So, a CVE usually contains components modelling awareness lists that show information about user locations which are updated as users move inside the environment. For example an awareness list may show all users connected to the environment including their actual work *room* or the connected users inside a room and their actual state. Whenever a user logs into the environment or moves between *rooms*, the information in both sort of lists should be updated. However, the component (source component) knowing that a user state has changed do not need to be aware of the sort of components (target components) that need to receive the new state for that user. The source component sends the *newUser* event with the destiny address set to null. Then, the *coordination aspect* intercepts this event and notifies it to those components noted in the

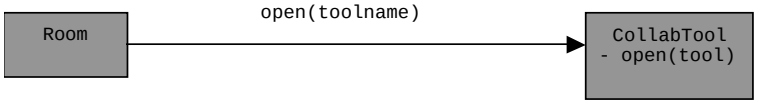


**Figure 5.a**

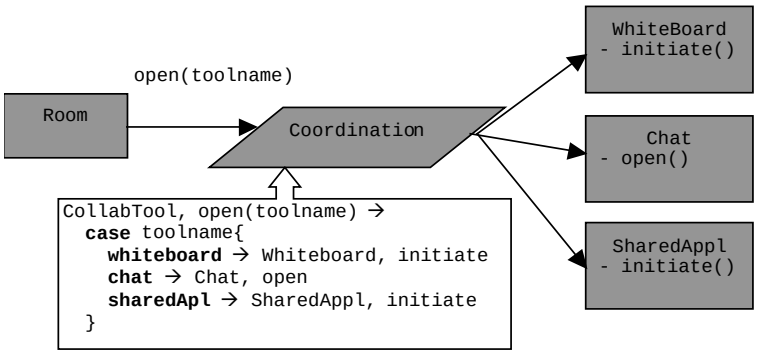


**Figure 5.b**

**Figure 5: Coordination aspect as an Adapter**

aspect internal data. In figure 2, when the component *room* use the *execute* primitive of the middleware layer it sets the *target_component* parameter to null. The composition rules stored in the middleware layer states that the *coordination aspect* is applied before the invocation of the *newUser* event in the *room* component. The evaluation of the *coordination aspect* determines that components with roles *CVEAwarList* and *RoomAwarList* are listeners of the *newUser* event, so the even is sent to these components.

Even in normal case, that is output messages containing the destination address, there are situations where the *coordination aspect* can be very useful. Though a component can use the CID to uniquely address other component, the CID is only known if a message from that component was received previously, so it is not possible to use it in all situations. In addition, components *reusability* increases if the system is designed having in mind the *roles* each component plays in the interaction and not concrete instances. As we already explained in section 2.3, this can be achieved using the alternative invocation mechanisms offered by the middleware layer: *role-based*, *instance-based* or *interface-based*. However, the inconvenience of using these mechanisms is that more than one component can be candidate to receive the message. Depending on the application logic, all of them or only a subset of them should actually receive the message, but components must not be aware of that. So, another functionality of the *coordination aspect* is to solve this problem, by linking source and target components transparently.

For example, in the development of CVEs we detach the component behavior from component view (component graphical representation) implementing them in two separate components. This means that a *room* is actually modelled with two components: the *dataroom* and the *viewroom* components. According to the component-aspect definition described above, these components do not have explicit references among them and do not know each other CIDs. Take a CVEs with different rooms instantiated simultaneously for a user, this means that there are different *dataroom* components and different *viewroom* components and it is not possible to differentiate them. When, for instance, a user interacts with a *viewroom* and a message is sent to a *dataroom*, there is no way to know which *dataroom* component must receive the message unless we use a *coordination aspect* that matches pairs *dataroom* and *viewroom* components.

In figure 3 we have two *dataroom* 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 figure 3) the *coordination aspect* do not have any information about the components. Lets suppose that the *dataroom* component with CID1 creates the *viewroom* component with CID3. The *coordination aspect* is applied before the creation of the *viewroom* component and registers the pair CID1 and CID3 (step2 in figure 3). From now on, when the component *dataroom* with CID1 sends a message to a *viewroom* component, the *coordination aspect* change transparently the invocation mechanism from role-based to identifier-based sending the message to the component with CID3. The same occurs for the other pair of components (step3 in figure 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 previous example when a user enters in a *room*, the *coordination aspect* sends the *newUser(user)* message to all the components whose interface implements this method. However, more complicated interactions are possible. A *coordination aspect* can encapsulate a state transition diagram as it is showed in figure 4. Suppose a CVE where each user has his/her own office and guest users can enter private rooms only if the owner is there. In this case the *coordination aspect* has two states, *open* and *close*. If the owner is absent the *coordination aspect* is in the *close* state sending *reject* messages to users that try to *log* into the room. Otherwise, the *coordination aspect* is in the *open* state and it sends the *accept* message when somebody knocks the door.

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 because of different message names, arguments or types. In addition, the *coordination aspect* might hide the fact that the service used by a

source component is actually implemented with a combination of components and not with only one.

For instance, in figure 5.a a CVE 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 it 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*, as shown in 5.b. In this case, new components offer different entry methods, so the *coordination aspect* is needed as an adapter. The *coordination aspect* will replace the name of the output message (*open*) with that offered by new components, depending on the *toolname* parameter value.

# 4. CONCLUSIONS AND FUTURE WORK

From our experience we consider AO Frameworks a better approach than AO languages, specially for applications with high customization requirements, due to the separation of components and aspects in all the software lifecycle, and the dynamic composition of them at runtime. Both features impact on the *flexibility*, *reusability* and *extensibility* of the resultant software.

In this paper we have presented the main characteristics of our component-aspect model: 1) components and aspects are first order entities that exist at runtime; 2) the model detaches components and aspects interfaces from the final implementation classes identifying them by role names; 3) the AA is explicitly stated and stored in the platform; 4) components and aspects implementation can be modified at runtime without client code recompilation and without changing the abstract definition of the software architecture; 5) components do not have direct references among them; 6) components have no knowledge about the aspects they are affected by; 7) the number and type of aspects that are applicable to a component might change dynamically.

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, detaches 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. Concretely we are developing a virtual office as part of a funded research project.

# 5. REFERENCES


[1] S. Benford, C. Greenhalgh, T. Rodden, J. Pycock. To what extend is cyberspace really a space? collaborative virtual environments. *Communications of the ACM*, 44(7), July 2001.

[2] A.W. Brown, K.C. Wallnau. The current state of CBSE. *IEEE Software*, September/October 1999.

[3] C.A. Constantinides, A. Bader, T.H. Elrad, M.E. Fayad and P. Netinant. Designing an Aspect-Oriented Framework in an Object-Oriented environment. *ACM Computing Surveys*, March 2000.

[4] L. Fuentes, J.M. Troya. Coordinating distributed components on the web: an integrated development environment. *Software-Practice and Experience*, 31, 2001.

[5] W.C. Jamison, D. Lea. Truce: Agent coordination through concurrent interpretation of role-based protocols. In *Coordination'99*, November 1999.

[6] G. Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*. LNCS 1241. Springer-Verlag, 1997.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold. An overview of AspectJ. In *ECOOP01*, June 2001.

[8] C. Lopes, E. Hilsdale, J. Hugunin, M. Kersten, G. Kiczales. Illustrations of crosscutting. In *ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, June 11-12 2000.

[9] M. Pinto, M. Amor, L. Fuentes, J.M. Troya. Collaborative virtual environment development: An aspect-oriented approach. In *Proceedings of DDMA Workshop. In conjunction with the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001.

[10] M. Pinto, M. Amor, L. Fuentes, J.M. Troya. Heterogeneous users in collaborative virtual environments using aop. In *Proceedings of the CoopIS'01 workshop*, September 2001.

[11] M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya. Towards an aspect-oriented framework in the design of collaborative virtual environments. In *Proceedings of FTDCS'01 workshop*, November 2001.

[12] R. Pawlack, L. Seinturier, L. Duchien, G. Florin. Jac: A flexible and efficient framework for aop in java. In *Reflection'01*, September 2001.

[13] H. Shinkuro, T. Tomioka, T. Ohsawa, K. Okada, Y. Matsushita. A virtual office environment based on a shared room realizing awareness space and transmitting awareness information. In *Proceedings of the 10th annual ACM symposium on user interface software and technology*, 1997.