Towards an Aspect-Oriented Framework in the design of Collaborative Virtual Environments *

M. Pinto, L. Fuentes, J.M. Troya
Dpto. 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

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

Abstract

The increasing complexity in the development of distributed system has promoted the appearance of new software technologies that complements compositional framework technology providing a high degree of separation of concerns. One of these approaches is aspect-oriented programming that introduces a new entity, the aspect, to model those features that cut across different components in the system, increasing its extensibility and configurability. Our goal is the application of this new technology to develop an aspect-oriented application framework to construct highly reusable collaborative virtual environments in short time.

1. Introduction

The increasing complexity in the development of distributed system has promoted the appearance of new software technologies that complements compositional framework technology providing a high degree of separation of concerns. One of these approaches is Aspect-Oriented Programming (AOP) [1] [2] [3] that introduces a new entity, the *aspect*, to model those features that cut across different components in the system, increasing its extensibility and configurability.

Our main goal is the application of this new technology in the design of an aspect-oriented application framework to construct highly reusable collaborative virtual environments (CVE) in short time. With this goal in mind we began studying other collaborative environments [4] [5] [6] to extract CVEs' properties, mainly *collaboration*, *awareness*, *authentication*, *access control* and *persistence*; and CVEs' components as *user sites*, *rooms* or *places*, *documents* or *shared tools*. In order to be useful, our CVE should be *configurable*, *extensible*, *scalable*, *adaptable* and *able* to reconfigure itself according to *user preferences* and necessities [7] [8] [9]. Due to the complexity of these systems, it is almost impossible to achieve all of these objectives if we construct each new service from scratch. This means that the use of the appropriate software technology will be critical.

During the last years, framework technology has consolidated as a suitable technology for the design and implementation of complex distributed systems [10] [11]. A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [12]. Using a component-oriented framework we must identify the main components in the application domain and provide the interfaces of these components and the way they have to be composed in order to derive a new instance of the virtual environment. With different combinations of these basic components or extensions of them it would be able to configure different CVEs with less effort.

Now, suppose that we use the framework to develop a virtual office service. For instance, I may say:

> *My virtual office will have a HomeRoom - the user's site, for each connected user; one SocialRoom - a rest place, and one MeetingRoom - a common place for meetings. It will be possible to use any kind of office software - text editors, database programs, data sheets. In addition,*

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

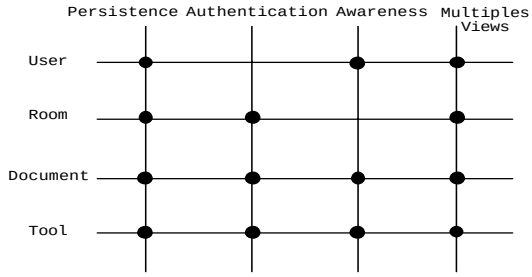

Figure 1. Functionality and Aspects cut crossing

in the SocialRoom there will be card games and video on demand applications and in the MeetingRoom there will be collaborative tools, like shared blackboard.

All the elements mentioned before would be modeled as components of the framework. But there are a lot of properties that are present in all of these components. Some of these properties are the *persistence* of the environment, the *access control* to the components or the *awareness* of users, documents and tools in the environment. We did not mention these issues before because using traditional framework technology these properties usually are hard coded in the framework basic units, the components. With the aim of reusing those properties in every component they must not be modeled as part of the functionality of the components in the framework, but as separate entities. Figure 1 shows different components (user, room, document, tool) and different properties that cut across most of them.

Framework technology by itself it is not enough to manage the complexity of these systems, because it does not provide the separation of concerns needed to implement the basic functionality of the components and the properties that cut across them as different entities. By one way, the developer has to provide different implementations of the same component's functionality because these properties are hard coded inside the components. By the other way, the same properties are implemented through different components which arise problems derived from code replication.

Nowadays there are different programming techniques that try to solve this problem through the separation of concerns. Some of these approaches are *Subject-Oriented Programming*, *Composition Filters*, *Generative Programming* or *Aspect-Oriented Programming*. The study and comparison of all these approaches is beyond the scope of this paper. A classification of these and other approaches can be found in [13].

The main contribution of our work is the design of an aspect-oriented application framework (AOAF) for the development of CVEs. In the rest of the paper we will define what is an *aspect* and will present the main characteristics of AOP and our approach to apply it to the development of CVE systems.

2. Aspect-Oriented Programming

The use of object-oriented application frameworks (OOAF) [11] is an approach to build more robust and correct applications in a shorter development time that those built from scratch. The primary benefits of OOAF stem from the *modularity*, *reusability*, *extensibility*, and *inversion of control* they provide to developers [10]. However, the development of a framework and the subsequent instantiation or extension to build custom domain specific applications is not a straightforward task and some times, the framework fails to provide the modularity needed to localize the impact of design and implementation changes. This issue decreases frameworks reusability and extensibility. In addition, very often a minor change in the functionality of the system can mean mayor changes in many components of the framework.

The main complexity in the development of a framework is the decomposition of the system functionality in components. Many systems have properties that do not necessarily align with the functional components of the system. These properties normally are present across different components of the system and are called *aspects* in the AOP. Aspects were introduced by [3] and are defined as system properties that tend to cut across functional components, increasing their interdependencies, and resulting in what was coined as the *code tangling* problem. This code tangling makes the source code difficult to develop, understand and evolve by destroying modularity and reducing software quality [1].

Current AOP approaches model components and aspects as two separate entities where the aspects are automatically weaved into the functional behaviour of the system in order to produce the overall system [2] [14]. These approaches differ in the languages used to define the aspects and in the way in which aspects and components are weaved.

Some authors [14] argue that current languages are not suitable for the straightforward description of aspects. They think that it is necessary to have languages designed specifically for the description of one aspect. These languages, called *aspect languages*, can be completely new languages or extensions of general-purpose languages. But the main drawback of this approach is that we need an aspect language for every type of aspect and also an automatic weaver tool that must implement one or more aspect languages [1]. The other alternative is the use of a general-purpose language as is, without any extensions to describe both components and aspects, which is more feasible.

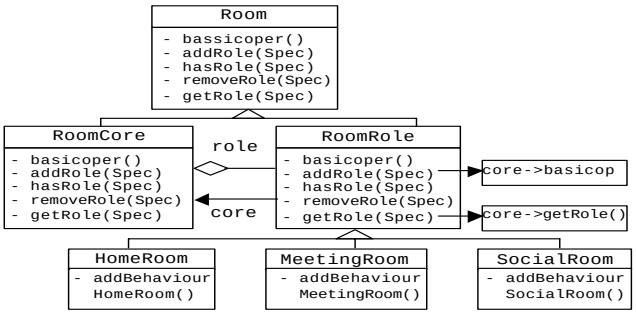

Figure 2. Room Component with Pattern Role

The other difference between AOP approaches is the way in which aspects and components are weaved. We can distinguish between the *static* and the *dynamic* weaving. In the static weaving *components* and *aspects* are implemented as different entities, but then during the compilation both are intermingled in the same binary code. The resultant code is highly optimized, but there are also several disadvantages, for instance, it is required to implement a compiler to weave the code which is a costly task and the modification of the aspects at runtime it is not possible. By the other way, in the dynamic weaving both the aspects and components are independent entities at runtime that are composed dynamically.

Among all these approaches in our aspect-oriented framework, presented in the next section, we choose to define the aspects and the components in the same general-purpose language without any extension or modification. We have detected different aspects in our application domain - *awareness*, *persistence*, *authentication*, *coordination*, *multiple views*; although the software developer can add new ones depending on their needs. Daring with such a changing environment we found that it is not very practical to define a new aspect language for each aspect.

In addition, we use dynamic weaving. In order to do the weaving, we define a middleware platform that maintains information about the components and aspects in the virtual environment and composes them at runtime. This makes the environment more *configurable*, *extensible*, *scalable* and *adaptable* taking into account the different users preferences and necessities. As the users preferences change during the execution of the service, it can be adapted to these changes modifying the aspects that are applied dynamically.

3. Aspect-Oriented Framework in CVE

In this section we present an AOF, our approach for the development of CVEs. The framework has two main entities, *components* and *aspects*. Since we pursue the dynamic composition of components and aspects in a way that both types of entities are bounded, as the CVE has to be tailored to users profiles, components and aspects do not know nothing about each other. Composition between components and aspects is performed dynamically at runtime depending on the information stored in a third entity, a middleware layer (ML) that we present below. In order to construct a new CVE, the framework user will have to provide the description of components and aspects and the composition information between them. That is, the architectural constraints that control what type of aspects must intercept a message sent or received by a certain type of component. The most outstanding feature of this approach is that the ML will be able to modify this information at runtime, making the environment more *extensible*, *scalable* and *adaptable*.

3.1. Framework Components

For the design of components, we have found that there are many design patterns that can be used. Characterizing the types of components in CVEs, we encountered that the most recurrent ones are the *User* component, that models any user inside the environment; the *Document* component that models any document shared by users or the *Room* component, that models a common place where users collaborate.

The aim of this work is to provide a framework ready to be configured as any kind of CVE. Examples of CVEs are *virtual offices*, *collaborative design of artifacts* in different domains or *distributed team games*. The matter is that although all the components presented above, the *user*, *room* or *document*, appear in any of the mentioned environments, they play different roles.

The previous reasoning follow us to include the role concept inside the components. The *Role Pattern* [15] seems to be suitable to components design. Figure 2 shows the classes of this design pattern modeling the *Room* component. The context-specific views of the component are modeled as separate role objects which are dynamically attached to and removed from the core object. The base class in this pattern (the room component in the example) defines a minimal protocol for managing roles.

The main advantage of this pattern is that roles can be added and removed dynamically and this issue promotes the reuse of the basic components in different contexts. The main disadvantage is that the components design, implementation and management is more complex. However, to skip this difficult the developer can follow any other pattern. The use of the *Role Pattern* is only a suggestion that it is not imposed by the framework.

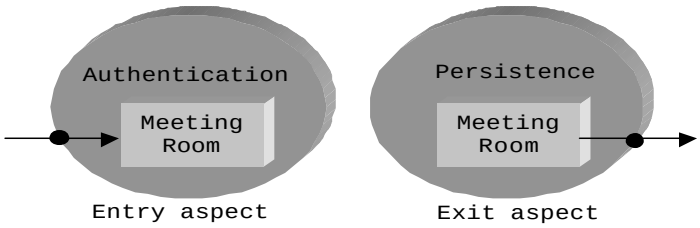

Figure 3. Representation of the entry and exit aspects

3.2. Framework Aspects

All the properties that intersect several components in the framework, will be modeled as aspects. Studying our application domain we appreciate that in addition to synchronization, communication, coordination or security *aspects*, typical of distributed systems, there are distinct aspects like *awareness*, *persistence*, *authentication* and *multiple views*. In [7] you can find a complete description of them and the reason why we consider these properties as aspects.

Due to the diversity of aspects and their different behaviour, we do not propose any design pattern for aspects, that can be implemented at the user taste. However, he/she decides the category of the aspect:

- An aspect can be *environment-oriented*, *user-oriented*, *type-oriented* or *component-oriented*. An aspect is *environment-oriented* if there is only one instance of this aspect in the environment. It is *user-oriented* if there is an instance of the aspect for each user in the system, it is *type-oriented* if there is an instance of the aspect for each type of component and it is *component-oriented* if there is an instance of the aspect for each component object. For instance, the *authentication* aspect will be an *environment-oriented* aspect if all the elements in the virtual office use a LDAP directory service to access control; a *user-oriented* aspect if the user can configure the access control to their resources, a *type-oriented* aspect if while the room components use a LDAP directory, the document components use access control lists and a *component-oriented* aspect if each room or document in execution have their own access policy.

- Regarding when aspect must be applied, it can be an *entry* aspect or an *exit* aspect. An *entry* aspect is evaluated before the execution of a method in a component and an *exit* aspect is evaluated after the execution of the method (Figure 3).

Figure 3 represents the entrance and exit of a *room*. When a user wants to enter to the *room*, he/she must authenticate himself, and when he goes out all his actions are stored in the environment. As a consequence, depending on who is the user, his or her preferences may imply a different visual representation (*multiple views*) of the *room* and the elements contained in it.

3.3. Dynamic Composition of Components and Aspects

In this framework, functionality and aspects are modeled as separate entities, so we define a composition mechanism that will operate at runtime. The dynamic connections between components and aspects are established by the *middleware layer*.

In order to do this, it stores information about:

- The components that comprise the final application.
- The aspects that comprise the final application.
- The definition of a component-aspect oriented architecture that is expressed in terms of which aspects must be applied to each component, the order in which the aspects must be applied and the type of the aspects.

3.3.1 Component Description in AO-CVE Middleware Layer. When a software developer wants to construct a new environment using our framework he or she must register all the components in the middleware layer. In order to register the components he or she uses the syntax showed below. For every component in the application it must be provided an identifier, that will be a string, the interface that implement, and the implementation class.

```
component ::=
  def component
    <ident> <interface> <implementation>
  enddef
```

As an example, you can see the declaration of a *Room* component.

```
room_component ::=
  def component
    Room roomint.class roomimpl.class
  enddef
```

As we can see in the declaration of the components, we only provide the *interface* and *implementation* of components and there is no mention of the roles. By this way the middleware layer is not aware of how components were implemented. Although we recommend the use of the *Role Pattern*, any developer will be able to implement the components as their own preferences.

In addition the separation of the *interface* and *implementation* decouples the services from the components implementation. This means that maintaining the same interface, it is possible to change the component behaviour dynamically changing the information about its implementation in the middleware layer, and the rest of the application will not be affected (i.e. the class name).

3.3.2 Aspect Description in AO-CVE middleware layer. Likewise components, when a software developer wants to construct a new environment using our framework he or she must also register all the aspects in the middleware layer. In order to register the aspects he or she uses the syntax showed below. This notation shows that each aspect has a unique interface but different possible implementations. This means that for a given type of component the same aspect will be applied, but in a different way. Each aspect is defined by an identifier, that is a string, an interface and the list of available implementations classes identified by a string.

```
aspect ::=
  def aspect
   <identifier> <interface> <type> <impl-list>
  enddef

impl-list ::= <impl> | <impl> impl-list

impl ::=
  def impl
    <identifier> <implementation>
  enddef

type ::= env | user | type | component
```

As an example, you can see the declaration of the *Authentication* aspect with three possible implementations - *BDAuthent*, where the user login/password is consulted in a database, *LDAPAuthent*, that uses a LDAP Directory Service and *OwnerAuthent*, where the resource's owner decides who can access to it.

```
authentication_aspect ::=
  def aspect
   Authentication authint.class env
   def impl BDAuth bdauthimp.class enddef
   def impl LDAPAuth LDAPauthimp.class enddef
   def impl OwnerAuth ownerAuth.class enddef
  enddef
```

3.3.3 Component and Aspect dynamic interconnection. The interconnection information that the middleware layer stores it is defined in terms of the *components identifiers*, the *aspects identifiers*, the *messages* sent between the types of components and the *entry* and *exit* aspects applied for each message.

```
comp-aspect composition ::= <comp-list>

comp-list ::= <comp> | <comp> <comp-list>

comp :: =
  def comp <aspect-identifier> <aspect-type>
   (apply-comp <component-list> | all)|
   (notapply-comp <component-list>)
   (apply-mes <messages-list> | all)|
   (notapply-mes <messages-list>)
  enddef

aspect-type ::= input | output
```

As it can be seen above, for each aspect identifier the software developer must indicate if it is an input (entry) or an output (exit) aspect. The list of components and messages that are affected for this aspect can be specified as an inclusion list or an exclusion list. If you use an inclusion list it is also possible to use the wild-card *all* to indicate that all the components or all the messages in a component are affected by that aspect. An example of the composition between components and aspects is showed below.

```
composition ::=
 def comp authentication input
  (apply-comp room, document)
  (apply-mes all)
 enddef

 def comp persistence output
  (apply-comp room, document, user)
  (apply-mes all)
 enddef

 def comp multipleview input
  (apply-comp room)
  (apply-mes access, add, paint)
 enddef
```

We want to point out that although this information will be normally established during the design of the service and provided to the middleware layer statically when the service is instantiated the first time, the middleware layer will also offer the possibility of modifying it dynamically at runtime, for instance the organizer of a *virtual office* that is defining the profiles of each type of worker. This issue makes the resulting environment very flexible and adaptable because you can add, delete or modify the architecture of the system by putting different versions of the same component or aspect at runtime. This dynamic modification of the architecture also allows the change of the service behaviour. A valuable benefit is that you can achieve this adaptability without any modification in the functionality of components.

Figure 4 shows the middleware layer that performs the dynamic composition of components and aspects at runtime. When a component in the framework invokes a method over other component, the source component does not have an explicit reference to the target com-

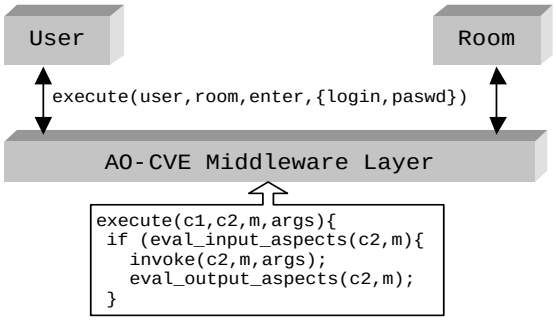

Figure 4. The AO-CVE middleware layer composition process

ponent. Invoking the method *execute(source comp,target comp,method,args)*, the execution goes through the middleware layer. This method, first applies the *input aspects* associated with the target component. If the application goes well, it invokes the method in the target component using reflexive programming. Finally, it applies the *output aspects* associated with the target component.

4. Conclusions and Future Work

The work presented in this paper is in its early stages. Our application domain is the CVE and the originality of this proposal is that it is the first aspect-oriented approach for the construction of CVEs that allow us to define a more *configurable*, *extensible*, *scalable* and *adaptable* collaborative application according to *users preferences*.

Another contribution of this work to AOP are the dynamic composition between components and aspects at runtime through a middleware layer that is responsible of this composition. In an interactive environment like this, it is very important to assure that the communication and computation overload introduced by the dynamic composition of the system entities, does not affect the quality of the service.

The work presented here is a first approach and it is difficult to discuss about performance issues yet. However, we have experience in the development of MultiTEL [16], a distributed dynamic framework, implemented in Java for the development of multimedia telecommunication services over the Web. MultiTEL is built upon a component-connector model with a dynamic composition mechanism. We have implemented and evaluated several multimedia services in MultiTEL with excellent results. More information about MultiTEL can be found in the Web (http://www.lcc.uma.es/˜lff/MultiTEL/).

Our future goals are the complete definition of the middleware layer (defining components, aspects and services description languages), the definition of the application framework for the development of CVEs and the implementation of a working prototype, concretely a virtual office. A further goal of our research is to address the integration between other third-party collaborative solutions and our framework.

References

[1] C. A. Constantinides, A. Bader, T. H. Elrad, M. Fayad, and P. Netinant. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, March 2000.

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

[3] G. Kiczales et al Aspect-Oriented Programming *Proceedings of ECOOP'97*, LNCS 1241. Springer-Verlag.

[4] M. Roseman and S. Greenberg. Teamrooms: Network places for collaboration. *Proceedings of ACM CSCW*, 1996.

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

[6] M. Sohlenkamp and G. Ghwelos. Integrating communication, cooperation and awareness: The diva virtual office environment. *Proceedings of ACM CSCW*, 1994.

[7] M. Pinto, M. Amor, L. Fuentes and J.M. Troya Collaborative Virtual Environment Development: An Aspect-Oriented Approach. *Proceedings of DDMA'01*, April 2001.

[8] M. Pinto, M. Amor, L. Fuentes and J.M. Troya Runtime Coordination of Components: Design Patterns vs. Component-Aspect Based Platforms. *Proceedings of the AOP workshop at ECOOP01*, June 2001.

[9] M. Pinto, M. Amor, L. Fuentes and J.M. Troya Supporting Heterogeneous Users in Collaborative Virtual Environments using AOP. *next publication in Proceedings of the CoopIS'01 workshop*, September 2001.

[10] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10), October 1997.

[11] M. Fayad, D. Schmidt, and R. Johnson. Building application frameworks: Object-oriented foundations of framework design. September 1999.

[12] R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10), October 1997.

[13] K. Czarnecki, U.W. Eisenecker, and P. Steyaert Beyond Objects: Generative Programming. *Proceedings of the AOP workshop at ECOOP97*, June 1997.

[14] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales Aspect Oriented Programming. *Workshop Report, ECOOP'97 Workshop on AOP*, June, 1997.

[15] D. Baumer, D. Riehle, W. Siberski, and M. Wulf Role Object. In Pattern Languages of Program Design. *Addison-Wesley*, 2000.

[16] L. Fuentes and J. M. Troya. A java framework for web-based multimedia and collaborative applications. *IEEE Internet Computing*, 3(2), March/April 1999.