How MDA Can Help Designing Component- and Aspect-based Applications *

Lidia Fuentes, Mónica Pinto, and Antonio Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{lff,pinto,av}@lcc.uma.es

Abstract

Distributed systems are inherently complex, and therefore difficult to design and develop. Experience shows that new technologies—such as components, aspects, and application frameworks—can be effectively used for building distributed applications. However, our experience also shows that most of the applications built in that way are difficult to be re-used, documented, and maintained. Probably, one of the major reasons is the lack of a clear separation between the concepts used at different levels (application domain, application architecture, supporting application platform, programming language, etc.). In this paper we present our experience with a platform we developed for building distributed applications using components and aspects. In particular, we show how many of the (conceptual) problems we hit when trying to document, re-use, and implement it in different contexts can be naturally solved with the adoption of the MDA concepts. In addition, we describe the process we followed for identifying and separating the entities that live in different "models" (in the MDA sense), and the required transformations between them. MDA offers a good framework for specifying different views of our model, and mappings to platform-specific profiles. In this way, we are able to address the particular needs of different stakeholders: from the designer interested in developing new applications following our (component and aspect-based) modeling approach, to the software vendor that wants to implement a proprietary version of our supporting middleware framework in CORBA, EJB or .NET.

1. Introduction

The increasing complexity of large-scale enterprise applications is driving the Software Engineering community to design and adopt new technologies for the development of distributed systems in timely and affordable manners.

Component-based Software Development (CBSD) has become one of the key technologies for the effective construction of large, complex software systems [12]. CBSD advocates the use of prefabricated parts, perhaps developed at different times, by different people, and possibly with different uses in mind. The goal is the reduction of development times, costs, and efforts, while improving the flexibility, reliability, and maintainability of the final application due to the (re)use of software components already developed and validated. However, everybody agrees that achieving an accurate functional decomposition of a system into separate context-independent components is not easy task.

Aspect Oriented Software Development (AOSD) tries to add a new dimension to the solution, by encapsulating the different cross-cutting concerns of an application into separate *aspects*, which can then be *weaved* together to form a functioning system. Although aspects were originally defined only at the programming level [5], AOSD tries to cover all phases of the software development life-cycle, from requirements to implementation (see http://aosd.net).

Current research tries to combine AOSD and CBSD techniques, in order to obtain all their mutual benefits. Thus, aspects may become reusable parts, which can be weaved among themselves, and then attached to the individual software components. Weaving can be either static (during compilation) or dynamic (at run-time).

So far, most of the efforts from the Software Engineering community have concentrated on the technical issues of these new technologies. However, there is a growing interest in the modeling concerns of the software development process, too. Different approaches are successfully addressing the modeling of CBSD applications [1, 2], although the situation is not so bright when it comes to model software *aspects*. Several proposals try to model them using UML, although current practices show the difficulties to "translate" those UML models into particular aspect-oriented languages or platforms without losing relevant information.

*Proceedings of the IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 124-135, Septiembre 2003, Brisbane, Australia

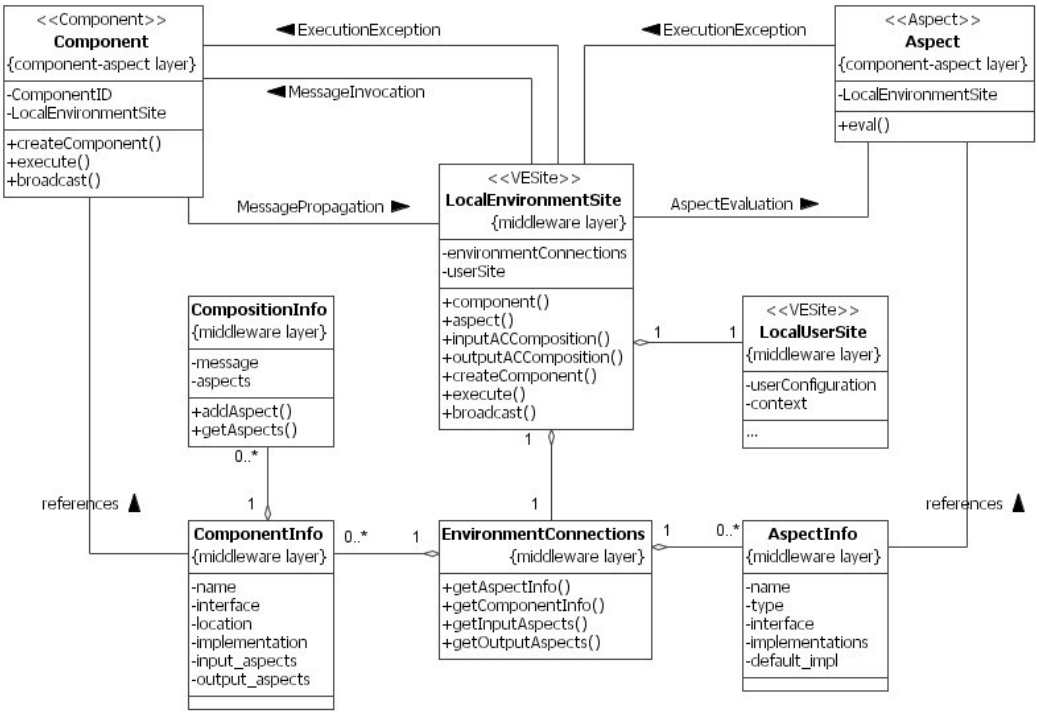

Figure 1. Original UML profile of the DAOP platform

OMG's new Model Driven Architecture (MDA) is a modeling initiative that tries to cover the complete life-cycle of software systems, allowing the definition of machine-readable application and data models which permit long-term flexibility of implementation, integration, maintenance, testability and simulation [6]. In this paper we will try to show how MDA can be used for modeling component- and aspect-based systems in a platform-independent manner. Moreover, we will also try to show that MDA may be helpful not only for modeling and designing systems from scratch, but also for helping document and re-engineer existing systems. Our experience is based on a middleware platform (the "Dynamic Aspect-Oriented Platform", DAOP) that we developed for building distributed applications.

The structure of this paper is a follows. First, Section 2 presents the DAOP platform, its supporting component and aspect model, and discusses the main (conceptual) problems that we came across when tried to document the platform for re-use and evolution. The following three sections describe how MDA can help addressing these problems, and how we think that the MDA concepts can be applied to the more general case of building component- and aspect-based distributed applications. A running example is used to illustrate our proposal. Finally, Section 6 draws some conclusions and outlines some further work.

2. Motivation

2.1. The Dynamic Aspect-Oriented Platform

The *Dynamic Aspect-Oriented Platform* (DAOP) is a middleware platform that defines components and aspects as first-class reusable entities, which can be dynamically weaved to build the final system. Probably, the most relevant feature of DAOP is the component and aspect independence, which means that neither components nor aspects have any information on when or how they are composed. This provides a powerful and very flexible mechanism for late binding. The DAOP platform also provides a set of common services to most distributed applications, such as message delivery, broadcasting, persistence service, etc. DAOP has been successfully used for implementing different kinds of virtual collaborative applications [9].

DAOP was originally documented using UML profiles [10], which have been defined as the natural candidates for documenting such kind of frameworks [4, 7]. A *UML profile* is a set of extensions to UML using its built-in extension facilities. A UML profile for a platform or an application framework is a standard means for expressing the semantics of that platform or framework using UML.

Figure 1 shows the core meta-model of the original UML profile for DAOP. As we can see, DAOP defines two layers,

the *component-aspect layer*—with the components and aspects existing at runtime—, and the *middleware layer*—in charge of composing these entities in a dynamic way. The composition of aspects and components is guided by some architectural constraints, defined by a set of rules describing which aspects can be applied to each component, when they must be applied and in what order, and a list of possible implementation classes. The dynamic composition process is performed by the LocalEnvironmentSite, using the information provided by the EnvironmentConnections object, both parts of the middleware layer. This last object maintains the *architectural constraints* common to all users, while the LocalUserSite contains the profile of each user. For instance, if different implementations of the same aspect are available in a given site, the LocalEnvironmentSite will contain the list of all of them, while the LocalUserSite will indicate the concrete implementation that has to be used for a specific user.

As shown in Figure 1, neither the components nor the aspects contain any direct reference to each other, enabling their dynamic binding. As a consequence, both components and aspects implementations can be replaced without affecting the application execution. In addition, DAOP offers a set of communication primitives for sending and broadcasting messages, which are also available at the Component class level—the root of any DAOP component. Likewise, all DAOP aspects should inherit from the Aspect class, which implements the eval() primitive. This method is the one invoked by the platform in order to evaluate the aspect. The precise moment in which the aspect is evaluated (before or after a method invocation, or before or after a message delivery/reception) will be determined by the way the system designer has specified it in the application's architecture, which is explicitly described in the LocalEnvironmentSite.

2.2. Problems Found

Once DAOP was finished, we hit two major problems. The first one was about designing and documenting a methodology that could make effective use of DAOP for building distributed applications. In that moment we discovered that it was very difficult to differentiate between some of the entities that live at separate (conceptual) levels, especially when they share names. For instance, the software engineer could use "components" and "aspects" to design the system at the software architecture level, which are somehow different entities from DAOP's components and aspects. Even though a mapping can be defined between them, they must not be confused. As a matter of fact, components at the architecture level need to be mapped to DAOP-Components (which are the executable "component instances") and to DAOP-ComponentInfo objects (which contain information about where the component instances are, how to deploy them, etc.).

The second problem was about implementation and evolution. We had implemented DAOP using Java-RMI. How to move it to EJB or CORBA? Likewise, how could we separate what was DAOP-specific from what was inherited from the supporting middleware and programming model? DAOP was designed to be independent from the supporting language and distributed object platform. However, this was difficult to document by just using UML profiles.

As we found out, one of the core reasons behind those problems is that each UML profile sits at one particular conceptual level, i.e., UML profiles are very good for defining the entities that live in a particular *model* (in the MDA sense), as well as the relationships among those entities. However, UML profiles in isolation are not expressive enough for relating entities at different levels, and for defining mappings and transformations between them. This is why in the original UML profile for DAOP some entities from different conceptual levels appear intermixed.

In this sense, the two main entities of DAOP—from the application's designer point of view—are the components and the aspects. However, looking at the UML profile in Figure 1, we can see how they appear together with other entities which are also important to DAOP, but not relevant for modeling applications with DAOP. For instance, the component entity includes a reference to the middleware platform (LocalEnvironmentSite) and to the methods that it offers to send messages (execute(), broadcast()). This information is only relevant to the potential implementor or maintainer of DAOP, but not to the DAOP application designer.

2.3. MDA to the Rescue

We came to the conclusion that, in order to address these problems, the first step was to identify the different conceptual levels involved in the development of an application using our DAOP platform. The following list of *models* was produced.

The Computational Model (C-M) focuses on the functional decomposition of the system into objects which interact at interfaces, exchanging messages and signals that invoke operations and deliver service responses—but without detailing the system precise architecture, or any of its implementation details. This model basically corresponds to an ODP computational viewpoint model of the system, or to Zachman's Framework for Enterprise Architecture Row 3 [13]. Entities of this model are objects (implementing interfaces) and operations, to which we have added some constraints for expressing extra-functional requirements (such as security or persistence, for instance).

The component and aspect model (CAM) defines the basic entities and the structure of the system from an architectural point of view. In our case, components and aspects are the basic building blocks, following our goal of integrating CBSD and AOSD approaches.

The DAOP platform implements the concepts of the CAM model in a particular way. This level is still technology-independent, since it just deals about the way components and aspects are weaved, and how the *abstract* entities of the CAM model can be represented from the computational and engineering viewpoints—but still independently from the middleware platform (CORBA, EJB, .NET) or programming language used to implement them.

The middleware platform provides a description of the system from a technology viewpoint. In this model we decide whether we want to implement the DAOP platform using Java/RMI, CORBA, EJB, or .NET, using their corresponding services and mechanisms.

Looking at the different levels we realized that they can be naturally integrated using the MDA architecture, and that we can make use of the MDA facilities for relating them. In particular, they all can be considered as "platform models" in the MDA terminology [6].

In MDA, a *model* of a system is a description or specification of that system and its environment for some certain purpose. A *platform* provides a set of parts and services that can be used for building systems. Combining both concepts, a *platform model* provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application [6].

MDA distinguishes between platform independent models (PIM) and platform specific models (PSM). The first ones focus on the operation of a system independently from the platform it will be implemented in. A PSM combines the PIM with the details that specify how that system uses a particular type of platform.

MDA also defines *model transformations*, processes that allow converting one model to another model of the same system. The general model transformation from a PIM to a PSM is illustrated by the MDA pattern, shown in Figure 2. MDA defines many ways in which such transformation can be done. In our case we will use MDA *mappings*. A mapping provides specifications for transforming a PIM into a PSM for a particular platform [6]. Model instance mappings define *marks*. A mark represents a concept in the PSM, which can be applied to an element of the PIM to indicate how that element is to be transformed. The marks are used by the software architect to take the PIM and mark it for use in a particular platform. The marked PIM is then used to prepare a PSM for that platform. This is the approach we will follow.

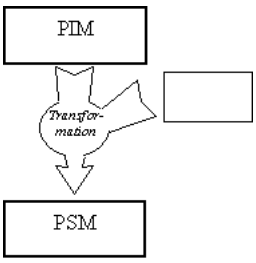

Figure 2. The MDA pattern for model transformation

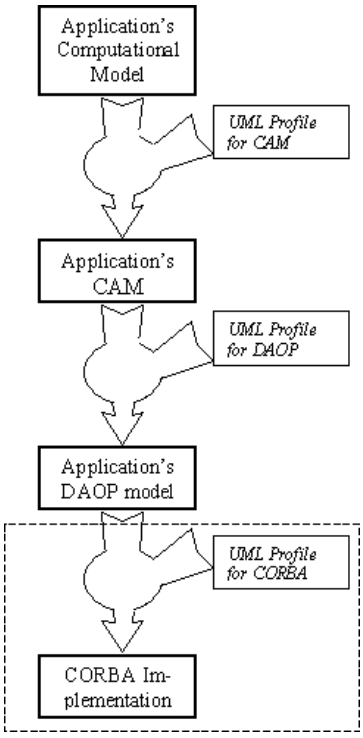

Figure 3. The stack with the different models and the MDA transformations between them.

With all this, the different conceptual levels previously described can be expressed using the MDA as Figure 3 shows. It contains four models and three transformations between them. Please notice how each PSM of a transformation becomes the PIM of the next, and that the platform applied in a transformation is defined in terms of an UML profile for that platform.

In the drawing, our starting point is a Computational Model of an application (the *Application's C-M*). The first transformation uses the entities defined in the metamodel of the UML profile for CAM to mark the C-M, together with a set of specific mappings. The model produced by that transformation contains the model of the application described in terms of the CAM. This model is then marked using the specific marks defined in the metamodel of the UML profile for DAOP, and then transformed into a DAOP model of the application. The only entities in this last model are those of DAOP. In order to implement the system we transform it again, using another transformation. In the drawing we have used the UML profile for CORBA, which will produce a CORBA implementation of the system. Of course, other implementation alternatives are possible (EJB, .NET, even EDOC), as discussed in Section 5.

Please notice as well the homogeneous treatment of the last part of the software development process in MDA, since it treats implementation as another model. Actually, MDA adopts the OMG definition of *implementation*, as a specification which provides all the information needed to construct a system and to put it into operation. This structure also keeps models "clean", in the sense that each model does not mix entities that belong to different conceptual levels. The following sections describe all these models and transformations in more detail.

3. Applying MDA: from C-M to CAM

3.1. An example of C-M

In this section we use a running example to describe the transformation process from C-M to CAM. This example is based on Virtual Office (VO) application we have developed as part of a bigger project [9]. A VO integrates different CSCW applications and documents in a *shared space*. The main goal is the construction of a *secure*, *persistent* and *integrated* shared space that provides to geographically dispersed users the *awareness* they need to communicate and collaborate as if they were co-located in a real place.

Figure 4 shows a simple specification of a virtual office (many details have been omitted for space reasons) using a computational viewpoint. This UML diagram contains the metamodel of the VO, in which the main class, Room, models the shared space where users join to collaborate. Rooms may contain a set of resources, such as documents and collaborative tools. Examples of collaborative tools are whiteboards, chats, and meeting applications. All user interactions with a Room are done through a graphical user interface, which is represented by the InterfaceToUser class. Finally, class RoomAwareness represents the computational objects that display awareness information about the users and other resources currently in the room.

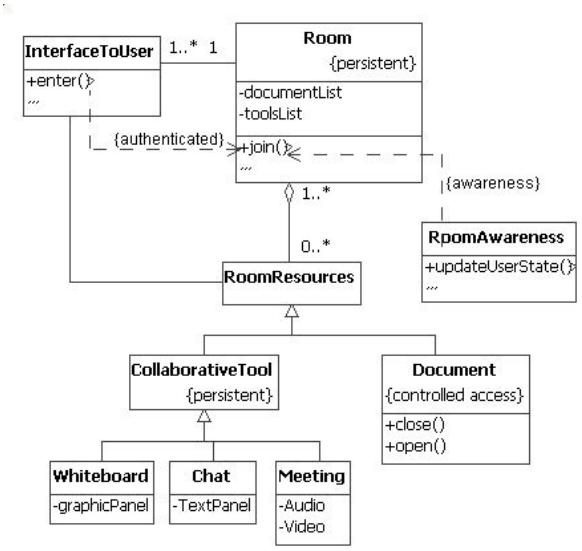

Figure 4. A C-M of the Virtual Office example

These classes model the functional behavior of a VO. However, we have mentioned before a set of extra-functional features, such as *security*, *persistence* and *awareness* that need to be modeled, too. We have used UML dependencies and constraints to model such extra-functional requirements. For instance, the {persistent} constraint appears in the Room and the CollaborativeTool classes, meaning that the information owned by a room, and the data generated by users while working with the collaborative tools, must be persistent. The {controlled access} constraint associated to the Document class indicates that only users with the right kind of permissions can access a document.

While these extra-functional properties affect just a single element (e.g., a class), there are other properties such as *authentication* and *awareness* whose behavior may crosscut different classes. For instance, a user that wants to *enter* to a room must be authenticated first, and this implies that the user has to introduce some identification information. Only if the user is registered in the system he or she would *join* the room. This behavior is represented in Figure 4 by a dependency relationship from the enter() method in the InterfaceToUser class to the join() method in the Room class. Similarly, a dependency relationship from the updateUserState() method in the RoomAwareness class to the join() method in the Room class indicates that once a user joined the room, the RoomAwareness class must be notified to display the updated user awareness information.

The fact that some of these extra-functional requirements clearly crosscut several modeling elements lead us to use AOSD technologies, as the natural candidate for modeling and implementing the system.

3.2. The Component and Aspect Model (CAM)

Most of the advances and achievements in AOSD are at the programming language level, but there is still a lack of widely-agreed high level notations for expressing and modeling aspects, specially those that model aspects independently from the implementation language, supporting middleware, and weaving mechanisms used. A standard UML notation would greatly facilitate to identify and express aspects at design level.

In the meantime before such a standard notation emerges, we have defined our own "Component and Aspect Model" (CAM), which tries to address many of the requirements of component- and aspect-based applications. Figure 5 shows the Metamodel of CAM (part of the UML Profile we have defined for it), that defines the main entities of CAM and the relationships among them. These entity names are the ones that can then be used as UML stereotypes for modeling applications using CAM.

In CAM, following the standard practices of CBSD and AOSD, components interact by exchanging messages and by emitting events. Aspects can be applied to (incoming and outgoing) messages, events, and also to component operations (on initialization, before or after an operation execution, etc.).

The main entities of the CAM model are Components and Aspects. Although in principle there is no restriction on the granularity of these elements, the way they are composed may impose some recommendations about their size and level of encapsulation. Thus, if components are distributed and interact by exchanging messages, and we want aspects to be dynamically composed at run-time by the underlying platform, we may probably wish to model both components and aspects as high-level, black-box, coarse-grained entities (e.g., as in DAOP or JAC [8]). However, those aspects that only contain a couple of sentences can usually be much better modeled by fine-grained white-box aspects that are applied directly to components (e.g., as in AspectJ). Since CAM was devised as a general model, it offers both possibilities and hence defines different "applies to" relationships for aspects (see Figure 5). Nevertheless, the distributed nature of the applications we are trying to design with CAM clearly moves us to consider the first case, and therefore in the following we will consider both components and aspects as coarse-grained, encapsulated entities, that can be later dynamically composed at run-time by the distributed DAOP platform.

Since in the CAM model aspects are treated as "special" kinds of components, both components and aspects share some common features. For instance, both may have a set of StateAttribute that will represent their public state, i.e., the information that should be made persistent—to be able to restore the state of a component or aspect instance. This information is used for implementing some requirements such as fault tolerance or persistence.

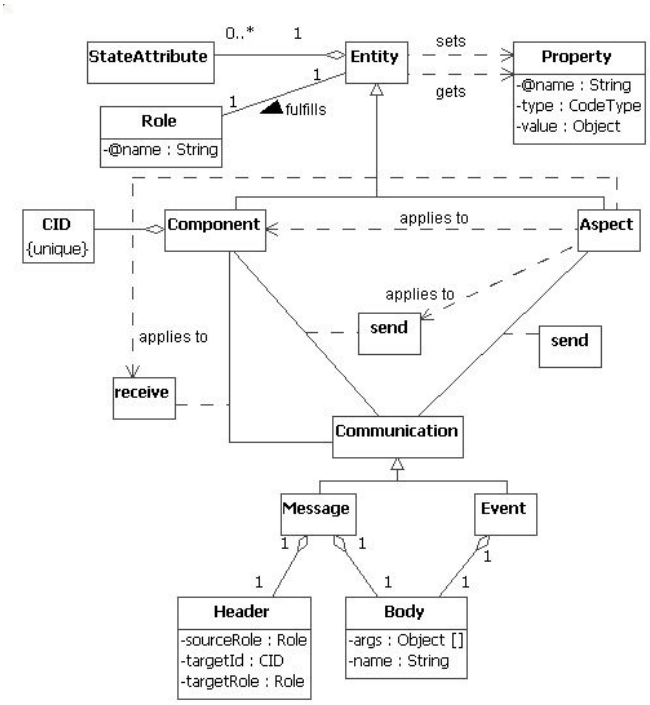

Figure 5. Metamodel of CAM

In order to detach components and aspects interfaces from their final implementations, we assign a unique role name (in class Role) to reference both components and aspects. These role names are architectural names that will be used for component-aspect composition and interaction, allowing loosely coupled communication among them—i.e., no hard-coded references need to be used for exchanging information, but just the role name of the target of a message. In addition, components can also be addressed by a unique identifier (class CID) that refers to a component instance, in case particular instances need to be addressed.

The AOSD community offers different approaches for weaving aspects, depending on the points where the pointcuts can be placed. Some approaches support the definition of pointcuts at anyplace of the code (e.g., before, after, around, ...), mainly because they are based on code insertion. Other approaches use different kinds of message interception, so the aspect evaluation is triggered by the delivery of a message or an event. This allows aspects to be applied to black-box components, closely to the CBSD philosophy. In CAM we have tried to produce a general model, and therefore CAM-aspects can be applied to the components's methods (allowing the former approaches), or just when a message or event is sent or received (allowing the

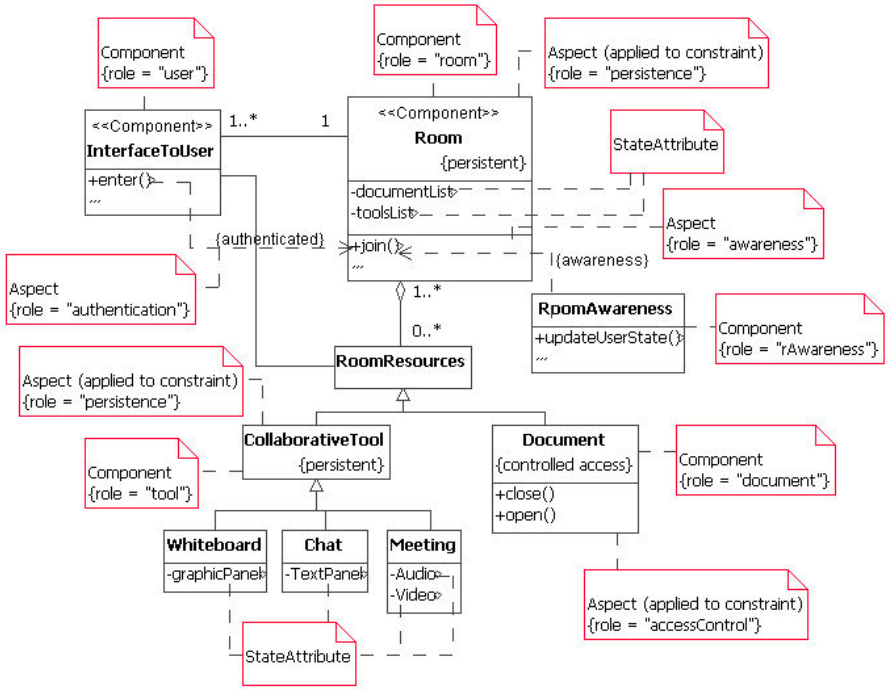

Figure 6. Marked C-M

latter ones)—see the dependency relationships in Figure 5. We understand the meaning of messages and events in the CBSD sense, since events are messages with no information about the source and/or target component. In this case, the platform resolves the target component(s) by using a coordination aspect.

Since software components usually have some context dependencies, i.e, sometimes they need to share data, the Property class encapsulates this kind of data dependency. Properties can also be used to solve dependencies between non-orthogonal aspects. Properties are identified by a unique name, their type, and current value.

3.3. CAM Markings and transformations

So far we have described a C-M for the VO system, and the Metamodel of CAM. Let us consider now how the C-M can be transformed into a CAM model of the Virtual Office application using the MDA mechanisms.

In order to define this transformation we will "mark" the C-M using the classes defined in the CAM metamodel (showed in Figure 5) as stereotypes. Although not included here for the sake of brevity, the full UML profile for CAM not only contains the classes that comprise its metamodel, but also information about the UML elements to which those class names can be applied as stereotypes.

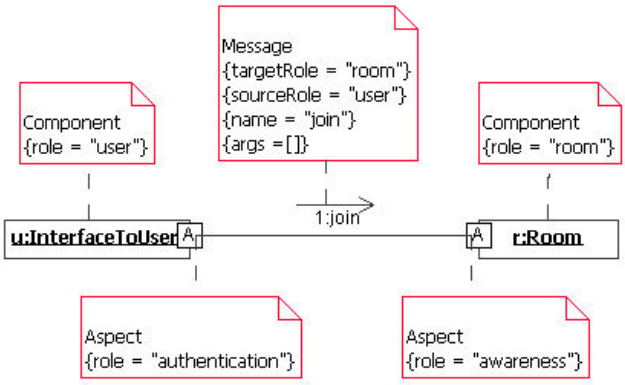

Figure 7. Marked C-M Collaboration Diagram

For instance, only UML classes can be stereotyped as Components, but no other UML elements such as constraints or notes. However, many different UML elements can be marked as Aspect, such as classes, method constraints, dependency relationships between methods, association ends of interaction diagrams, etc. The transformation process will apply different mapping rules for the same mark depending on the marked element.

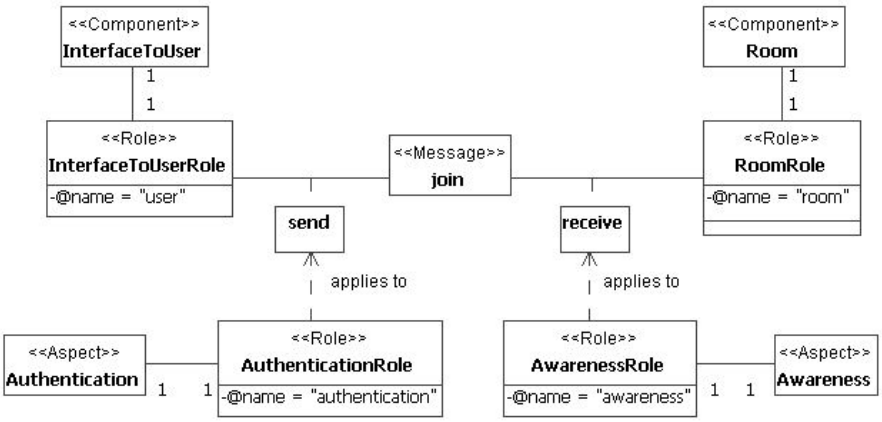

Figure 8. Transformed Diagram

After applying the marks defined in the CAM metamodel, we obtain the marked C-M showed in Figure 6. It is important to notice that marking an element as a Component or Aspect requires to specify the role name that the software developer wants to assign to that component or aspect (it is a mandatory attribute of a CAM entity). Intuitively, this is needed because components and aspects in our model are always addressed by the "role" they will fulfil in the system.

Not only class diagrams can be marked, but all kinds of UML diagrams. As an example, Figure 7 shows a marked collaboration diagram. The original (unmarked) diagram specified what happens when the user instructs the system (using the GUI interface object) that wants to enter() a room: the instance of the InterfaceToUser class invokes the join() method of an instance of the Room class. In this case we have marked both classes as components, and the join() method as a CAM message. We have also marked the ends of this association with two Aspect marks that have role names authentication and awareness.

Once the diagram is marked, the transformation process from C-M to CAM describes a set of *mapping* rules for each of the elements. The result of the application of such mapping rules in this case is a class diagram, which is shown in Figure 8.

Generally, a transformation defines a whole set of mapping rules, one for each mark and for each marked element. In order to illustrate the mapping rules, we will describe here the three rules that have been applied to the marked (C-M) collaboration diagram in Figure 7 to obtain the corresponding (CAM) class diagram shown in Figure 8.

1. **Components transformation**. This rule transforms UML classes marked as Component in the C-M, to UML classes in the CAM. For each of the C-M classes, two CAM classes are produced, and a 1-to-1 association between them. The first class represents the CAM component (stereotyped ≪Component≫), and the second one a CAM role (stereotyped ≪Role≫)—whose name attribute holds the name of the role specified in the mark. For instance, this rule can be applied to C-M classes InterfaceToUser and Room, which are marked as Component with roles "user" and "room", respectively. The result of applying this mapping rule to these classes produces the four UML classes (InterfaceToUser, InterfaceToUserRole, Room, and RoomRole) shown in Figure 8, with the stereotypes and relationships defined for them.

2. **Messages transformation**. This mapping rule transforms UML elements of a C-M collaboration diagram marked as Message to UML classes in the CAM with the ≪Message≫ stereotype. Each of these classes will be associated to the CAM classes that model the role names of the source and target component(s) of the message. A send association class will also be created between the role name of the source component and the message. Similarly, a receive association class is created between the message class and the role name of the target component. The example in Figure 8 shows the effect of applying this mapping rule to the collaboration diagram in Figure 7.

3. **Aspects transformation**. As mentioned earlier, the association ends of an association in a C-M collaboration diagram can be marked as Aspect, indicating that a given aspect (identified by its role name) should be applied when a message is sent/received at that end. These marks will be transformed to UML classes in CAM, stereotyped ≪Aspect≫, and a dependency relationship (applies to) will link the aspect class to the

operation it applies to. As we do for components, we create in CAM two classes for each element marked as Aspect, one modelling the aspect and other modelling the aspect role name. In our example we have two aspects: Awareness and Authentication, which are transformed into the classes and dependencies relationships shown in Figure 8.

4. Applying MDA: from CAM to DAOP

Once we have a CAM model of an application, in this section we will discuss how it can be transformed into its corresponding DAOP model. The first thing we need to define in order to achieve such a transformation is a UML profile for the DAOP platform, and then we need to define the mapping rules between the metamodels of CAM and DAOP.

4.1. The DAOP Platform

In the first place, we need to realize that the concepts of "components" and "aspects" at the DAOP level differ from the same concepts at the CAM level. More precisely, DAOP components and aspects refer to the CAM-component and CAM-aspect "instances", and therefore the information handled by DAOP about them should only contain how to locate and deploy them, as well as the services they offer. More precisely, the new metamodel of DAOP only contains information about the services and facilities that the DAOP offers to components and aspects (the elements that appear above the DAOP Platform class in Figure 9), together with the information the platform should store to provide such services (the classes below the DAOP Platform class in Figure 9). Please notice the difference between this new metamodel and the original one (shown in Figure 1).

DAOP Services and facilities. Since CAM components and aspects are able to create or destroy components, DAOP defines the corresponding methods as part of the ComponentFactory interface. Aspects may also create or delete other aspects using the AspectFactory service. For example, a monitor aspect would initiate a fault tolerance aspect after observing some anomalies in the behavior of the system.

DAOP also offers a set of component communication primitives. As in other component platforms (e.g., CORBA), DAOP allows components to send synchronous and asynchronous messages. Please notice that the CAM model only specified that components could send messages, but gave no details about how this could be implemented, allowing for both communication mechanisms. DAOP also allows components to throw events to other components, although events do not specify the targets. This mechanism is very useful to decouple components, and is specially well suited to enable components and aspects re-use. DAOP does not specify how to implement the distribution of events, allowing the implementor of DAOP to decide the preferred mechanism (e.g. publish-and-subscribe mechanism).

All DAOP-aspects should implement the AspectEvaluationService interface. This is a mandatory requirement since the platform will invoke the eval() method to evaluate an aspect. This is the only requirement an aspect must fulfill to be recognized by the platform as a valid DAOP aspect implementation.

Another important feature of software component instances is the use of *resources*, in the sense Szyperski defines them as "typed frozen objects" which are used to maintain a certain degree of data dependency with the environment [12]. Furthermore, based on our own experiences we also realized that we needed some kind of service that allowed all entities in the platform to store and consult information in order to configure themselves. The PropertyService shown in Figure 9 offers methods to set and get shared data (i.e., properties) from the platform.

One of the facilities offered by the DAOP platform is the persistence service, which provides all the methods required to store and retrieve component and aspect states. The implementation of this service is responsibility of the developer. This service is also the natural candidate to implement a persistence aspect, or some kinds of fault tolerance mechanisms.

Architecture of DAOP Platform. The internal structure of the DAOP platform is basically the one originally showed in Figure 1, although slightly improved since it now reflects the fact that "architectural" components and aspects live at a different level, and that DAOP only maintains information about the component and aspect instances. The DAOP platform contains an ApplicationArchitecture object that stores the architectural description of the application; and the ApplicationContext that holds the current list of components, aspects and properties instances. DAOP maintains the definition of the components' and aspects' services inside the application architecture, and references to their local or remote implementations in the application context. The ApplicationArchitecture is described in terms of components, aspects and a set of composition rules. As shown in Figure 9, component and aspect composition is defined in terms of their role names and a message name. This class contains explicitly when each aspect has to be applied, following the CAM model. More than one component instance may fulfil the same role within an application, but all of them must conform to the same component specification (likewise for aspects).

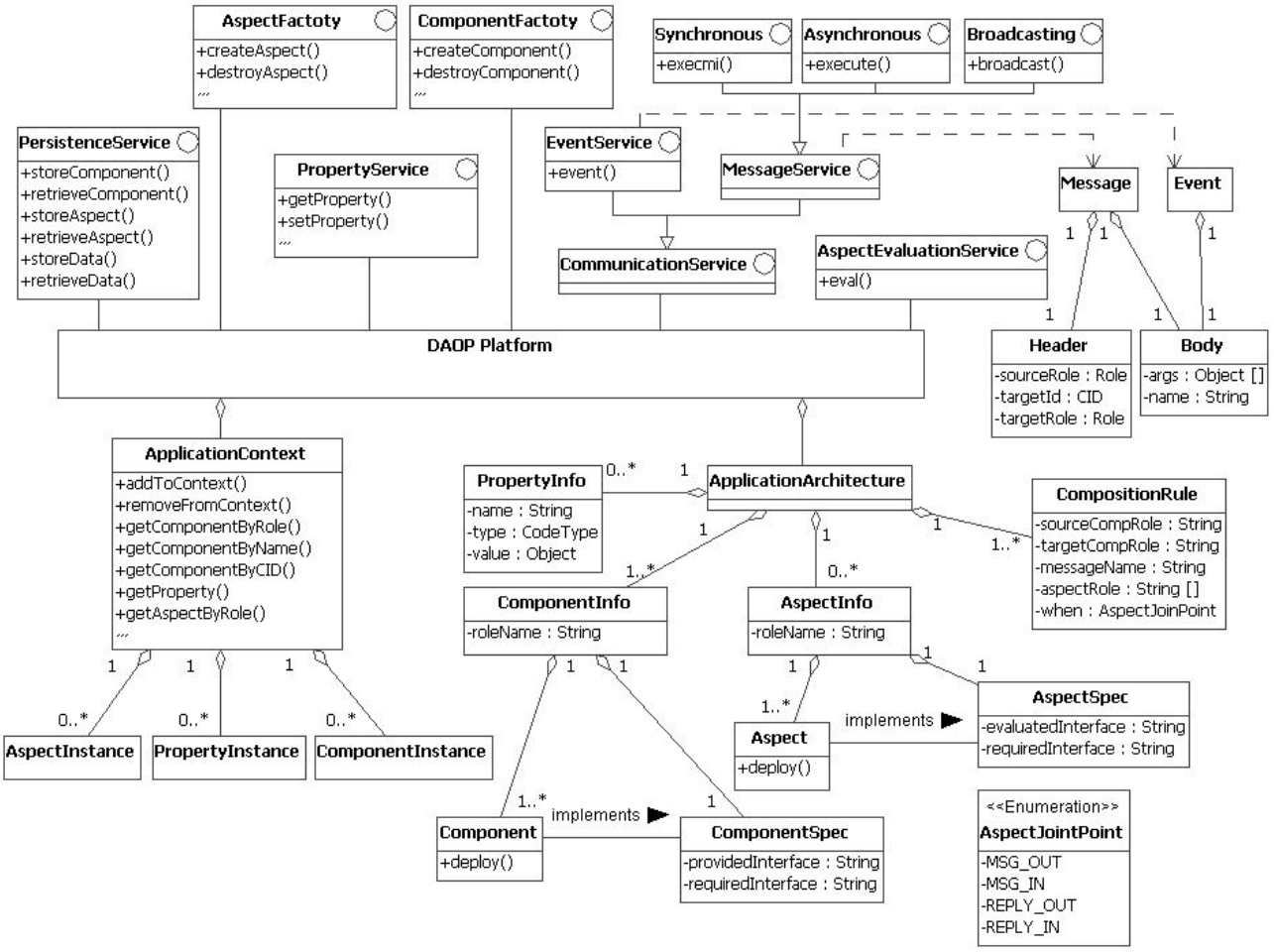

Figure 9. Metamodel of DAOP

4.2. DAOP Markings and Transformations

By adopting the same methodology that we applied for CAM, we have to define the marks and transformations needed to obtain a model of the application's CAM, expressed in terms of DAOP entities. Basically the information that the transformation process has to generate from the marked CAM is: the communication mechanisms between components, the aspect evaluation mechanisms, and the information that describes the architecture of the CAM application.

A CAM model for a specific application will specify the messages and events that components and aspects in the application are able to send and consume. On the other hand, the DAOP platform provides communication primitives to send synchronous and asynchronous messages, and to broadcast them. Therefore, messages in CAM have to be marked to specify which DAOP communication mechanism should be used.

Similar to component communication, the aspect joint points that we defined in CAM have to be mapped to DAOP joint points. As shown in the *AspectJointPoint* enumeration type in Figure 9, DAOP aspects can be applied to incoming messages and events (MSG_IN), outgoing messages and events (MSG_OUT), outgoing response messages (REPLY_OUT), and incoming response messages (REPLY_IN). Taking into account these joint points, CAM aspects that are applied when a component or an aspect sends a message or an event, will correspond to DAOP MSG_OUT aspects. Similarly, CAM aspects applied in the reception of a message or event are considered MSG_IN aspects in DAOP.

In addition to aspects that affect the emission or reception of a message or event, other kind of CAM aspects are those applied to component operations (on initializa-

tion, before or after an operation execution, etc.). However, DAOP does not distinguish between applying an aspect at the reception of a message or event, and applying it before a method execution in a component. Thus, CAM aspects to be applied before a method execution are mapped to MSG_IN aspects. Similarly, CAM aspects that have to be applied after the execution of a method will correspond to REPLY_OUT aspects in DAOP.

Of particular interest in the CAM to DAOP transformation is the information that DAOP stores about the architecture of the CAM application (using the ApplicationArchitecture object). As part of the CAM to DAOP transformation process, an XML document that contains the description of all the components and aspects in the CAM model is generated. Components and aspects are identified by their role names and are described in terms of their provided and required interfaces. The XML document contains also the composition rules described in the CAM model. (That is, the document is nothing else but the UML diagrams of the CAM model expressed in XML.)

During the execution, when an instance of the DAOP platform is created, the XML document is parsed and the structure of the CAM application is stored in the ApplicationArchitecture object (see Figure 9). This is precisely what we called the Application's DAOP model in Figure 3. The DAOP platform will use this information to deploy components and aspects and to compose them dynamically at runtime. Please notice the importance of this fact, since all the architectural information described in the UML diagrams at the CAM level can be used by the DAOP platform as-is. This avoids the usual "gap", or loss of information, between different conceptual levels—e.g., the information about the software architecture of an application is usually lost at implementation level.

5. Applying MDA: Implementing DAOP in Different Platforms

Once we have a model of the system using the DAOP elements, we could think of implementing the system in a particular distributed object platform such as CORBA, EJB, .NET, etc. Following the MDA approach, we could consider the DAOP platform as a PIM, and then transform it to one of these platform models. This is currently part of our ongoing work.

In Figure 3 we showed a possible implementation, using the UML profile for CORBA to mark the DAOP model elements, and defining a set of mappings from DAOP marked elements to CORBA elements. Since these elements are just CORBA interfaces and elements, the transformed model would be (easily?) implementable in CORBA.

As a matter of fact, if we count with a CORBA implementation of the DAOP elements and mechanisms, the system developer would need to provide just the code for the application-specific objects, i.e., their behavior. The rest of the system, including all the services provided by the DAOP platform, as well as many of the most commonly used aspects, would be already implemented. At this level, the degree of reuse obtained is comparable to the one provided by traditional Application Frameworks [3], in which the user simply has to customize the application-specific classes in order to build the final system. The benefits of using the MDA approach is that the user can "work" at a higher level of abstraction, and then use the MDA transformations to help him refine the model.

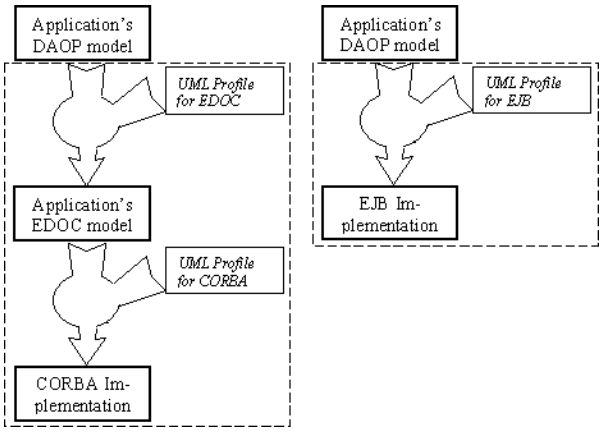

Figure 10. Two other alternatives for implementing the system

Another important benefit of using MDA is that it allowed DAOP to be defined independently from the underlying implementation platform. DAOP is currently implemented using Java/RMI, but we were concerned on how to "move" it to other middleware platforms. Therefore, apart from the CORBA implementation shown in Figure 3, it would relatively easy to define other implementation alternatives. For instance, we could also defined marks and mappings to EJB. Or even use EDOC as an interim platform, and define transformations from DAOP to EDOC. In this case we could make use of the transformations currently being defined from EDOC to CORBA, to EJB, or in the future to any other distributed object platform. The resulting "stacks" for two of the alternatives (EDOC+CORBA, or just EJB) is shown in Figure 10. An interesting exercise here would be to compare the two resulting implementations obtained by: (1) directly applying the UML profile for CORBA to the DAOP model; and (2) using DAOP to EDOC, and then from EDOC to CORBA. Alternatively, we can also define a UML profile for Java/RMI, and compare the result of such MDA transformation with our current implementation of DAOP.

6. Concluding Remarks

In this paper we have presented our experience with MDA, and how it has been useful for identifying and separating different conceptual levels, which appeared intermixed in a component- and aspect-based middleware platform for the development of distributed collaborative applications.

We found out that it is not only a matter of establishing such conceptual differences, something which can be achieved in other ways. The main—and unique—value of MDA is the provision of the right kind of mechanisms for expressing the different levels, the entities of each one, and for defining transformations between them.

The result is a set of models which allow an easy documentation, customization, and evolution of the systems being produced. Furthermore, many design and implementation alternatives are also possible, since it is just a matter of building the "stack" of models that suits our stakeholder's particular requirements. In addition, the upper levels of the stack provide very high-level conceptual models, which are completely independent from the underlying platforms. In particular, the CAM provides a very general model for designing component- and aspect-based applications, independently from the supporting CBSD or AOSD technology finally used to architect or implement the system.

In this paper we have only used part of the MDA, without entering into its facilities for model storage and exchange, or using the Computation Independent Model (CIM). But even focusing on model transformations, there are still many unresolved questions in MDA. For instance, providing some kind of automated support for the model transformations is pending. The MDA Guide offers a first classification of the different sorts of model transformations, but we have seen that their application-specific nature make them difficult to automate. As described in Section 4.2, we have automated the transformation from CAM to DAOP, using a XML-based architectural description language [11] to represent the application's CAM. This was easy because DAOP was designed to be faithful to the CAM model, and specially because DAOP makes use of the application's architecture as defined by the CAM diagrams. However, we are not sure about how far (or close) we are from automating the rest of the MDA transformations. We also need to complete our implementation of DAOP in different middleware platforms, and compare the efficiencies of the implementations of the VO application obtained through the different mappings (using direct mappings of UML Profile for CORBA or EJB, or the indirect mappings through EDOC and then to CORBA or EJB). We also plan to study how the process of creating marks can be generalized, and how many mapping rules can be identified. Finally, some of the MDA core concepts still need to be further clarified, namely the exact nature of the CIM, and how to map the business model of a system (probably a combination of the ODP enterprise and information viewpoint specifications) to its computational model using the MDA mechanisms. In any case, we need to count first with a set of examples and experiences that make use of MDA. Our contribution tries to provide a small step in this direction.

Acknowledgements We are very grateful to the anonymous referees for their insightful comments and suggestions, that greatly helped improving the contents and readability of the paper, and for providing very useful hints and guidelines for future extensions. This work has been supported by Spanish CICYT Project TIC2002-04309-C02-02.

References

[1] J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software*. Addison-Wesley, 2000.

[2] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.

[3] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.

[4] L. Fuentes, J. M. Troya, and A. Vallecillo. Using UML profiles for documenting web-based application frameworks. *Annals of Software Engineering*, 13:249–264, 2002.

[5] G. Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.

[6] Object Management Group. *MDA Guide (Draft Version 2)*, Jan. 2003. OMG document `ab/2003-01-03`.

[7] OMG. *Model Driven Architecture. A Technical Perspective*. Object Management Group, Jan. 2001. OMG document `ab/2001-01-01`.

[8] R. Pawlack, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient framework for AOP in Java. In *Proc. of Reflection'01*. Springer-Verlag, Sept. 2001.

[9] M. Pinto, M. Amor, L. Fuentes, and J. M. Troya. Collaborative virtual environment development: An aspect-oriented approach. In *International Workshop on Distributed Dynamic Multiservice Architectures (DDMA'01)*, pages 97–102, Arizona, Apr. 2001. IEEE Computer Society Press.

[10] M. Pinto, L. Fuentes, M. Fayad, and J. M. Troya. Separation of coordination in a Dynamic Aspect-Oriented Framework. In *Proc. of the First International Conference on AOSD*, pages 134–140, The Netherlands, Apr. 2002. ACM.

[11] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: An architecture description language for dynamic component and aspect-based development. In *Proc. of Generative Programming and Component Engineering (GPCE'03)*, Sept. 2003.

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

[13] J. A. Zachman. *The Zachman Framework: A Primer for Enterprise Engineering and Manufacturing*. Zachman International, 1997. `http://www.zifa.com`.