UML Profile Definition for Dealing with the Notification Aspect in Distributed Environments[1]

José Conejero	Juan Hernández	Roberto Rodríguez
Quercus Software Engineering Group	Quercus Software Engineering Group	Quercus Software Engineering Group
University of Extremadura	University of Extremadura	University of Extremadura
Spain	Spain	Spain
+34-927257268	+34-927257204	+34-927257548
chemacm@unex.es	juanher@unex.es	rre@unex.es

ABSTRACT

The CORBA Notification Service allows objects developed under this platform to communicate asynchronously. Nevertheless, the use of this service at implementation level implies a strong coupling between service and objects which use it. This coupling is due to the mixing of code. This mixing of code appears when we deal with two different aspects in the same object: event asynchronous communication and core functionality of the object. The use of aspect-oriented techniques let us avoid this problem whereas the extraction of this tangled and scattered code. In this paper, we propose the definition of an UML profile for dealing with the Event Notification Aspect in Corba distributed environments. This profile simplifies the application of this asynchronous communication. So we will have the event notification details perfectly separated from the basic functionality of the object details at design level. Finally, the use of this profile facilitates the generation of code regarding to the notification aspect. This code will be subsequently integrated with the objects which take part of the communication.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous.

D.2.3 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods.*

D.2.10 [**Software Engineering**]: Design – *Methodologies, representation.*

D.2.13 [**Software Engineering**]: Reusable software – *Reuse models.*

D.3.2. [**Language Classifications**]: Design Languages

D.3.3 [**Programming Languages**]: Language Contructs and Features – *Classes and objects.*

General Terms

Design, Standardization, Languages.

Keywords

Aspect-Oriented Software Design, Profile, UML, Aspect Modeling, CORBA, Distributed Environments.

1. INTRODUCTION

The Corba Notification Service allows objects of a Corba system to communicate asynchronously. This service fulfills a publish/subscribe model by means of an event channel, so that the communication exists between supplier and consumer of event objects which produce and consume events in the channel respectively. This service offers a set of functionalities for the communication of events such as well defined data structures, filters definition, transmission of events on demand, and others [1].

Nevertheless, the use of this service implies tangled and scattered code situations within the objects which use the service. These objects have to implement their core functionality tangled with the functionality regarding to the notification or reception of events. In this paper we propose the extraction of the functionality about the communication of events out of these objects. We propose the treatment of the communications of events in distributed Corba environments as an aspect. Besides we provide a UML profile to model this aspect at design phases.

The rest of the paper is organized as follows: section 2 describes why the notification of events in distributed environments can be treated as an aspect. In section 3 the UML profile defined to model the notification aspect is presented and is applied to a real use case, a control application of "domotic" installations. Finally, section 4 presents the main conclusions and open questions to be discussed in the workshop.

2. THE NOTIFICATION ASPECT

When a developer has to implement a Corba object or component and provide it with the property of sending or receiving events, this developer has to focus in two different concerns of the system. On the one hand, he has to focus on the development of basic functionality of the objects, on the other hand he has to manage with the aspects regarding to the notification of events. However, in most of cases the event notification will be an important feature of our system but won't be part of the core functionality of the objects or components of the system. So, the Event Notification is a crosscutting concern scattered over the objects of the system. This notification is an extra-functional property of these objects and it should be encapsulated in a separated and independent entity.

[1] This work has been supported by the project CICYT under grant TIC2002-04309-C02-01.

The Corba Notification is based on the implementation of a Publish / Subscribe model where objects that publish events send them to the event channel, provided by the Notification Service, and objects that subscribe themselves to receive these events are notified by the channel of the availability of the events. As well, the Publish / Subscribe model can be of push or pull type depending on whether the notification would be supervised by the supplier or by the consumer of events respectively.

```
* public class Supplier extends
  _PushSupplierImplBase
  {
^  private final static int CONT = 8;
*  private static org.omg.CORBA.ORB orb = null;

* public Supplier (EventChannel channel)
  {
^  //[sentences regarding to own functionality]
*  try {
*    ProxyPushConsumer proxy=getProxy(channel);
*    proxy.connect_any_push_supplier(this);
*    org.omg.CORBA.Any eventData;
*    eventData = orb.create_any();
^    // [DATA PRODUCTION]
*    proxy.push(eventData);
*    proxy.disconnect_push_consumer();
   }
*  catch (Exception ex) {
*    System.out.println("Supplier: " +
*                        ex.toString());
   }
  }

^ //[sentences regarding to own functionality]

* private static ProxyPushConsumer
* getProxy(EventChannel channel) throws Exception
* {
*  SupplierAdmin admin =
* Channel.default_supplier_admin();
*  ProxyConsumer plainProxy =
* admin.obtain_notification_push_consumer(
*  ClientType.ANY_EVENT,new
* org.omg.CORBA.IntHolder());
*  return
* ProxyPushConsumerHelper.narrow(plainProxy);
  }

* /* implementation of abstract methods
* _PushSupplierImplBase.disconnect_push_supplier
* _PushSupplierImplBase.subscription_change */

  } // End of Supplier Class
```

Figure 1. Code of class converted in Event Supplier.

As you can see in Figure 1, we have in the same entity the code of the notification of events and the code which implements the basic functionality of the object or component. So we have two concerns tangled. The lines of code which have been marked with ^ belong to the core functionality of the component, whereas the lines of code marked with * (**and written in bold**) implement aspects regarding to the notification of events. Thus, we can see that the component must inherit from `_PushSupplierImplBase` class. Besides, the component must obtain in its constructor the *proxy* to connect to and generate the data to be sent. Finally the component will send this data in form of events. Other methods the component has to implement to send events are *subscription_change* and *disconnect_push_supplier*. All this functionality appears tangled with the functionality which the components we are developing must implement. The properties of the system which the component must implement (the core functionality of the component) have been represented by the lines *//[sentences regarding to own functionality]*. This functionality won't be related with the Event Notification. This tangle of code will appear in all event suppliers we wish to implement in our system. A very similar situation to the described one will happen in the case of event consumers too.

By means of applying Aspect-Oriented Programming techniques, we can extract the notification code out of these components. This fact implies the increase of reusability in our systems. We obtain reusability on the one hand of the components developed and on the other hand of the event notification which could be applied to other components. Moreover, the maintenance and the quality of software developed following these techniques will be easier and higher respectively. There are several Aspect-Oriented Programming approaches which offer mechanisms to solve this problem at implementation phases, one of the most popular is AspectJ[2]. However, in this paper instead of solving this problem only at implementation level, we offer a mechanism to model the notification aspect by the definition of an UML profile, so we treat the problem at design phase. Thus, we can model this concern in a uniform way in different Corba distributed applications. Besides, the use of this profile helps in the automatic generation of the code regarding to the notification aspect and the later integration of this code with our systems. The code could be generated in the Aspect-Oriented approach which better fulfils our interests.

3. DEFINITION OF THE UML PROFILE FOR THE NOTIFICATION ASPECT

Despite the evolution that Aspect-Oriented Software Development [3] has experienced in last years, the available mechanisms to help to apply the concepts of aspect orientation at design phases are limited. However, several approaches like [6], [7], [11], [12], [13], [14] have emerged to reduce the complexity of applying these concepts in those phases and to help to model aspects. In [4] there is an interesting comparative of some of these approaches.

The main mechanisms that UML provide us to extend its syntax and semantics are the profiles. By means of profiles, we will be able to adapt our UML models to a particular application domain. The profiles are based on three elements that UML includes to manage this extension. These elements are the stereotypes, the tagged values and the constraints. In [6] and [7] we can see some examples of profile which have been applied to AOSD and which help to model well known concerns such as distribution or synchronization.

In order to be able to model at design phases the aspect notification in Corba distributed environments, we propose the definition of an UML profile. We have added some stereotypes to the UML meta-model to define this profile. These elements are: <<*Supplier*>> which represents an event producer, <<*Consumer*>> which represents an event consumer and <<*Channel*>> to model the event channel we want to use. As you can observe in Figure 2, these stereotypes use the <<*Class*>> element of the UML meta-model [5] as base element.

The reason to use the <<Class>> element is that the <<Supplier>> and <<Consumer>> elements are primary citizens in our approach. So we can use the <<*Supplier*>> or <<*Consumer*>> elements in our models in the same way we use the <<*Class*>> elements because we are extending the meta-model but we have not modified its semantic. Our new stereotypes follow the same semantic rules than the <<Class>> element. This is one of the basic rules to build UML profiles [8]. Moreover, the <<*Supplier*>> element models the aspect of producing events so when we relate this element with a class or component of our model, we will convert it in an event producer. To manage this transformation we have also defined the stereotypes <<*Supplier_crosscut*>> and <<Consumer_crosscut>>. The <<Supplier_crosscut>> and <<Consumer_crosscut>> elements use as base element the <<Association>> one of the UML meta-model. So these elements represent relations between components of UML class diagrams. We use them to express the crosscutting relations between the <<Supplier>> and <<Consumer>> and the core classes of the system.

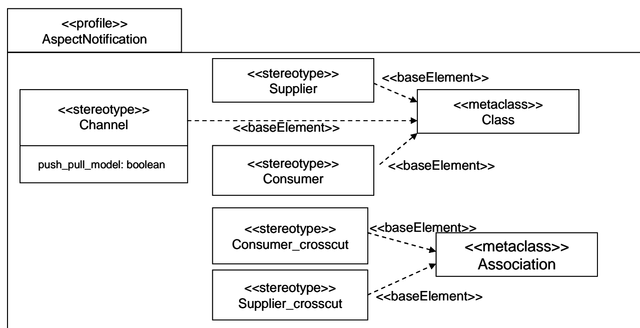

Figure 2. UML profile for the notification aspect in Corba.

In Figure 2 you can see the defined profile with the new elements. Besides the new stereotypes, we have added a tagged value called push_pull_model. This tagged value allows the distinction of the kind of notification we can use in each case. As we said in Section 2, in a push model notification, the events supplier will make invocations to the push method of the proxy (contained in the <<*Channel*>> element), whereas the event consumer must implement the push operation of the *PushConsumer* interface. On the other hand, if we choose a *pull* model, the events supplier will implement the *PullSupplier* interface which encapsulates a *pull* operation. The consumer will invoke the pull operation of the corresponding proxy. Thus, we can specify the kind of notification to use by modifying the tagged value push_pull_model. This will involve <<*Supplier*>> and <<*Consumer*>> will change their behavior.

Besides the commented stereotypes, in order to specify how the <<*Supplier*>> and the <<*Consumer*>> crosscut with the core classes of our system, we have added one more stereotype. This is the <<*Crosscut*>> stereotype. This stereotype uses as base element the <<*Association Class*>> element of the UML meta-model. We use the attributes of the <<*Association Class*>> to express any kind of *Introduction*. Is to said, by means of this <<*Introduction*>>, we can add members to the core class. In order to express the points where the <<*Supplier*>> or the <<*Consumer*>> modify the behavior of the core class, we use the operations of the new <<*Crosscut*>> element. So, we can specify where to apply the functionality of the <<*Supplier*>> or <<*Consumer*>> elements. If we adopt the AspectJ terminology we could say that with this last mechanism, we define the *Pointcuts* where to apply the functionality. We have used the <<*Pointcut*>> terminology to express this concept.

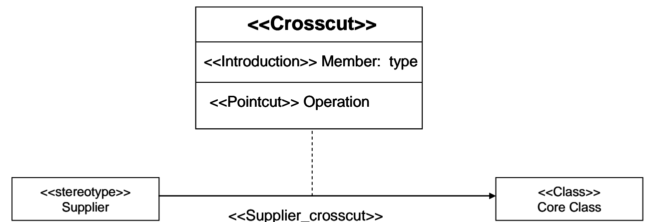

Figure 3. <<Crosscut>> stereotype

Considering that the new defined stereotypes <<Supplier>> and <<Consumer>> encapsulate the extra-functional property of notification, by means of the <<Supplier_crosscut>> and <<Consumer_crosscut>> relations, we will be able to add this notification to the classes which will use this event communication. Therefore, our elements of type <<Supplier>> represent an aspect which encapsulates the code regarding to the production of events, whereas the <<Consumer>> elements contain the event reception. In Figure 4 we can observe an example of the functionality added by the <<Supplier>> element.

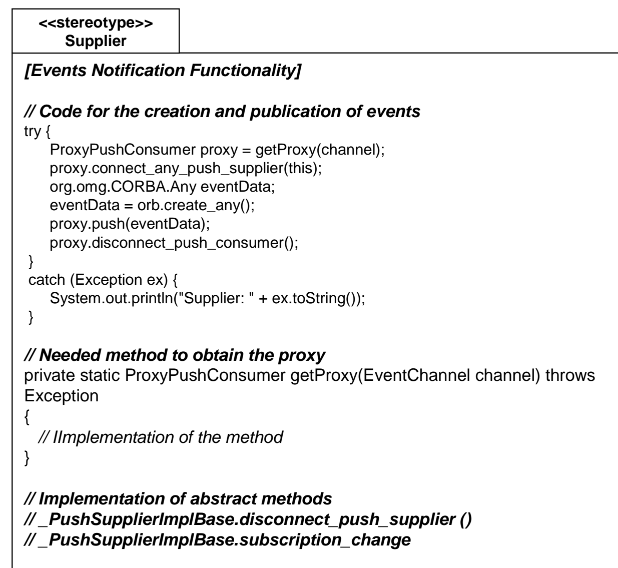

Figure 4. Functionality regarding to the events production.

3.1 A real case: the notification in EIB (European Installation Bus)

In this section we are going to show how to apply the defined profile to a real environment. This is an application developed and framed within a national research project of the Ministry of Education, Science and Technology of Spain. This real use case consists in a distributed application to control EIB “domotic” installations [9]. This application is based on the use of several software components which allow encapsulate the behavior of each device of our physic installation. The application will be compound of as many software components as “domotic” devices

exist in the installation. Besides these components to model the devices, there is a software component called *driver*. The component *driver* constitutes the bridge between the software components and the physic installation establishing a connection through a RS-232 interface with the EIB "domotic" bus. When some of the devices change its status, this change must be notified to the software components which model the physic devices. According the filters used, these components will receive some events and not others. For this implementation, we have used dCon. dCon is a commercial notification service which have been implemented by the DSTC (Distributed Systems Technology Centre of Australia).

Applying the defined UML profile, we will be able to have the events notification separated from the core functionality of each component. In Figure 5 you can see the simplified class diagram to model our application of domotic control. We can see that both the production and reception of events and the aspects regarding to the channel have been perfectly separated from the components of our installation. We don't show the <<Crosscut>> association classes to simplify the diagram. There would be a <<Crosscut>> association class in each <<Supplier_crosscut>> or <<Consumer_crosscut>> relationship.

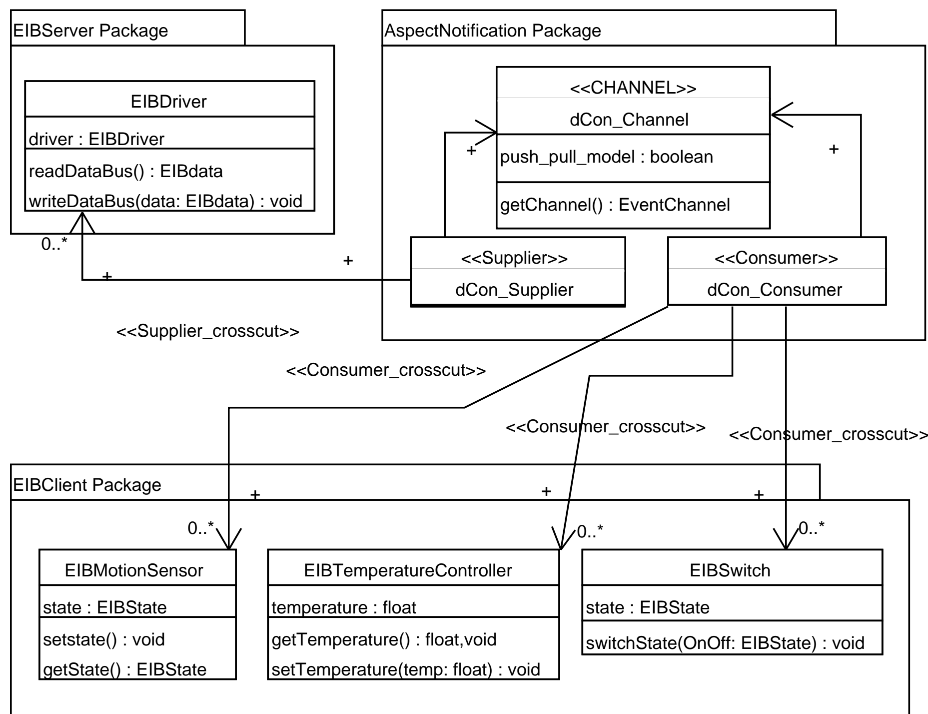

Figure 5. Simplified class diagram of the domotic application and the notification aspect.

By dealing with the notification aspect at modeling level, the independence of the aspect oriented platform and the Corba version chosen are guaranteed. The next step to fill the gap between design and implementation would be the generation of the aspect-oriented code (in my favorite approach) corresponding to the defined aspects (Supplier, Consumer, Channel) as well as the relationships between those aspects and the classes which they crosscut. Nowadays we plan to work in a tool to make this generation of code from a class diagram as the one shown in an automatic way.

If we choose the AspectJ like the aspect-oriented language target of the automatic generation of this notification code, the code which we would obtain for the notification of events is represented in Figure 6. For the reception of events, the AspectJ code to implement the aspect of reception would be similar to the code of Figure 6. Nevertheless, in the last case, this code would include the functionality of reception instead of delivery.

```
public aspect notification_supplier {

 declare parents : Productor extends
PushSupplierImplBase;
 public final static int Productor.CONT = 8;

 // Used attributes
 public static org.omg.CORBA.ORB Productor.orb =
null;

 after(Productor objeto) : execution
(Productor.new(..)) && target(objeto) {

 try {
 // [ Class to implement the obtaining of the
events channel ]

 Canal miCanal = new Canal();

 // [ Méthod to obtain the events channel ]

 EventChannel channel = miCanal.getChannel();
 ProxyPushConsumer proxy = getProxy(channel);
```

```
 proxy.connect_any_push_supplier(this);
 org.omg.CORBA.Any eventData;
 for (int i = CONT; i > 0; i--) {
     eventData = orb.create_any();
     eventData.insert_long(i);
     System.out.println("Productor:Event number
"+i);
     proxy.push(eventData);
  }
  proxy.disconnect_push_consumer();
 }
  catch (Exception ex) {
    System.out.println("Productor: " +
  ex.toString());
 }
}

 private static ProxyPushConsumer
Productor.getProxy (EventChannel channel) throws
Exception
 {
  // [ Method introduced within the object or
component ]
  SupplierAdmin admin =
channel.default_supplier_admin();
  ProxyConsumer plainProxy =
admin.obtain_notification_push_consumer(
  ClientType.ANY_EVENT,new
org.omg.CORBA.IntHolder());
   return
ProxyPushConsumerHelper.narrow(plainProxy);
 }

 public void Productor.disconnect_push_supplier
()
 {
  // [ Method introduced within the object or
component ]
 }

 public void Productor.subscription_change
(EventType[] added, EventType[] removed) throws
org.omg.CosNotifyComm.InvalidEventType
 {
  // [ Method introduced within the object or
component ]
 }
}
```

Figure 6. Code of the Notification Aspect with functionality regarding to production of events.

In order to clarify the interaction which occurs between the <<Supplier>> and <<Consumer>> elements and the core classes of the system, we show now a Use Case diagram and the Sequence Diagram which implements it. The Use Case diagram is shown in the Figure 7. By means of this figure, we represent the temperature adjustment process of a home by the user of the "domotic" installation.

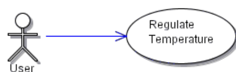

Figure 7. Use Case Diagram of the Temperature Regulation

When the user of the installation regulates the temperature, there are several objects or components of the system which take part in this temperature control. On the one hand, the component *EIBTemperatureController* must change its state (the wished temperature) in order to attend the user's request and on the other hand, there is a component which will update the current state of the *EIBTemperatureController*. The component which will play this role is the *EIBTemperatureControllerGUI*. Once the EIBTemperatureController has changed his state, it should notify this change to the other components of the system. However this event notification instead of be implemented within this component, will be implemented in a *<<Supplier>>* element. So the Event Notification keeps separated from the components where this functionality is scattered over. When the rest of components receive this notification, the component connected to the heating or the air conditioning control will execute the corresponding actions. The Figure 8 shows a sequence diagram that represents the adjustment temperature process and the functionality added to the *EIBTemperatureController* component. We are assuming a *push* notification model.

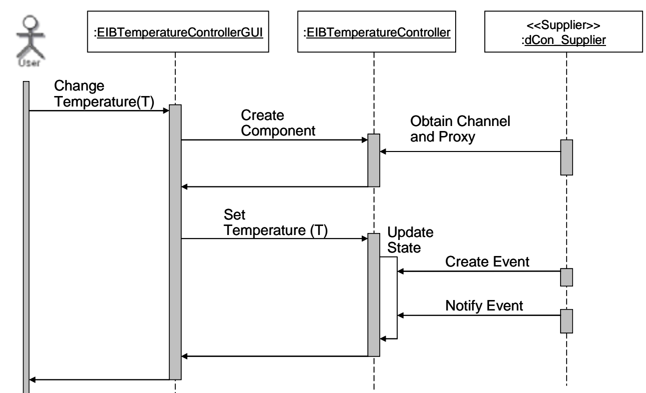

Figure 8. Control temperature process sequence diagram.

In the Figure 8 we can see the *<<Supplier>>* element added to the system. This element adds the event notification functionality to the *EIBTemperatureController* component. We can observe in the figure that the arrows which leave the *<<Supplier>>* element are directed towards the *EIBTemperatureController* component. Thus, we won't break the Obliviousness principle that the AOP approaches must fulfill [16] because the *EIBTemperatureController* doesn't know anything about the functionality which will be added to it. In this figure, we have represented the dynamic crosscutting that exists between both elements. We didn't show the static crosscutting (Introductions) because this crosscutting takes place out of the process shown in the diagram.

Now, we could consider what will happen if we change of programming language to implement our objects or if we decide to use some different Corba version, like Corba 3.0 (Corba Component Model, CCM). Although there will be some evident changes at implementation level, the changes at design level will be insignificant. The fact of dealing with the notification aspect at modeling level allows us to use the profile in the same way abstracting us of these changes.

If we choose to change the programming language used to develop our objects, we must modify the code which we would add to the classes of our system. This is the code regarding to the notification. This code depends on the particular service we use and the programming language used in each case. Obviously, we must change the way to inject this code in those classes. So we will generate the aspect-oriented code in the platform that better

adapt to the objects of our system (AspectJ if we implement the objects in Java, AspectC++ if the objects have been implemented in C++,…). However, all these changes affect to the system at implementation level. On the other hand, if we decide to use the CCM platform to develop our applications, the defined UML profile could be applied to the components in the same way. Considering that components in the CCM platform could act like events supplier and consumers too, the changes to apply the profile in the systems are focused at the implementation level again. In this case we must modify the code to add to the classes as well as the aspect-oriented platform which will depend on the programming language used to implement the components. Thus, in case of CCM, besides to inject code in the components, it is necessary to inject the events to supply or consume by each component in the IDL files of these components.

4. CONCLUSION AND FUTURE WORK.

It is evident that the Aspect-Oriented Programming has become an indispensable complement to the Object Oriented Programming. Nevertheless, we still need something more. It is necessary fill the gap between design and implementation in this area. This gap is being filling by the AOSD. However we realize we have an important lack of standard mechanisms to apply the aspect orientation at design phases in an useful way. In this paper we have proposed a particular case of aspect identification in a real domain and its specification in a higher level than implementation by the UML profile defined.

The application of the profile to the aspects modeling of our systems allows us to maintain the own functionality of the objects perfectly separated from the event notification. So we have a mechanism to deal with the notification aspect in Corba distributed systems in a uniform way.

As future work, we are considering the utilization of our profile in order to extend the UML Corba Profile [15] defined by the OMG. Thus, by means of the utilization of the Corba Profile, the applications developed using this profile would have the Event Notification perfectly separated from the core functionality of the system.

After the evolution that AOSD has experienced in last years, we can assure that the modeling of well-known aspects of our systems can be relatively easy. These aspects would be synchronization, coordination, persistence, etc. But we could ask some questions which they haven’t been answered yet. What happens with generic aspects which could appear in our systems? How could they be modeled in a generic way? And how does the new UML 2.0 specification or the new Model Driven Architecture help to do these designs? These and other questions should be discussed in the workshop.

5. REFERENCES

[1] Object Management Group (OMG*). CORBA Notification Service Specification*. WebSite: http://www.omg.org/cgi-bin/doc?formal/2002-08-04, (2002)

[2] AspectJ Project. Web Site: http://eclipse.org/aspectj.

[3] Aspect Oriented Software Design (AOSD). Web Site: http://www.aosd.net.

[4] Reina, A., Torres, J. and Toro, M. *Towards Developing Generic Solutions with Aspects*. In Proceedings of 5th International Workshop on Aspect Oriented Modeling with UML at the 7th International Conference on the Unified Modeling Language (UML). Lisbon, Portugal (2004).

[5] Object Management Group (OMG). *Unified Modeling Language Specification. v. 2.0.* Web Site: http://www.omg.org/docs/ptc/03-09-15.pdf , (2003).

[6] Aldawud, O.,Elrad, T., Bader, A. *UML Profile for Aspect-Oriented Software Design.* In Proceedings of 3rd International Workshop on Aspect Oriented Modeling with UML at the 2nd International Conference on Aspect-Oriented Software Development (AOSD). Boston, United States (2003).

[7] Herrero, J. *Propuesta de una Plataforma, Lenguaje y Diseño, para el Desarrollo de Aplicaciones Orientadas a Aspectos.* Tesis Doctoral, Universidad de Extremadura, (2003).

[8] Object Management Group (OMG). *Requirements for UML Profiles*. Web Site: http://www.omg.org/docs/ad/99-12-32.pdf, (1999).

[9] Conejero, J., Hernández, J., Pedrero, J. *Una plataforma JAVA para el Control y Monitorización de Instalaciones Domóticas EIB*. In Proceedings I JavaHispano Conference, Madrid, Spain (2003).

[10] Object Managment Group (OMG): *Object Constraint Language Specification.* Web site: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, (2003).

[11] Suzuki, J. and Yamamoto, Y. *Extending UML with Aspects: Aspect Support in the Design Phase*. In Proceedings of 3rd Aspect Oriented Programming Workshop at the 13th European Conference on Object Oriented Programming (ECOOP). Lisbon, Portugal (1999).

[12] Basch, M. and Sanchez, A. *Incorporating aspects into the UML.* In Proceedings of 3rd International Workshop on Aspect Oriented Modeling with UML at the 2nd International Conference on Aspect-Oriented Software Development (AOSD). Boston, United States (2003).

[13] Pawlak, R., Duchien, L., Florin, G, Legond-Aubry, F., Seinturier, L. and Martelli, L. *A UML Notation for Aspect.Oriented Software Design*. In Proceedings of 1st International Workshop on Aspect Oriented Modeling with UML at the 1st International Conference on Aspect-Oriented Software Development (AOSD). Enschede, Netherlands (2002).

[14] Zakaria, A., Hosny, H. and Zeid, A. *A UML Extension for Modeling Aspect-Oriented Systems.* In Proceedings of 2nd International Workshop on Aspect Oriented Modeling with UML at the 5th International Conference on the Unified Modeling Language (UML). Dresden, Germany (2002).

[15] Object Management Group (OMG). UML Profile for CORBA v. 1.0. Web Site: http://www.omg.org/cgi-bin/doc?formal/02-04-01 (2002)

[16] Filman, R. and Friedman, D. *Aspect-Oriented Programming is Quantification and Obliviousness.* In Proceedings of Workshop on Advanced Separation of Concerns at the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Conference, Minneapolis, United States (2000)