Journal of Universal Computer Science, vol. 12, no. 3 (2006), 236-251
submitted: 30/10/05, accepted: 15/1/06, appeared: 28/3/06 © J.UCS

Development of Ambient Intelligence Applications using Components and Aspects

Lidia Fuentes
(Universidad de Málaga, Spain
lff@lcc.uma.es)

Daniel Jiménez
(Universidad de Málaga, Spain
priego@lcc.uma.es)

Mónica Pinto
(Universidad de Málaga, Spain
pinto@lcc.uma.es)

Abstract: In recent times, interest in Ambient Intelligence (or AmI) has increased considerably. One of the main challenges in the development of these systems is to improve their modularization in order to achieve a high degree of reusability, adaptability and extensibility. This will help us to deal with the heterogeneity and evolution of the environments in which AmI devices exit. An example would be to easily adapt existing applications when new communication technologies appear. Current approaches apply component technologies to achieve these goals, but more should be done. Our research focuses on applying aspect technologies to components in order to improve AmI application modularization. We present the benefits of aspect technologies with regard to reusability and adaptability, by showing the limitations of PCOM, a component-based AmI middleware platform. We will show a study comparing DAOPAmI, our own component and aspect-based AmI middleware platform and PCOM.

Keywords: Aspects, Middleware, Components, Ambient Intelligence, Pervasive Computing
Categories: D 2, D 2.2, D 2.4, D 2.6, D 2.11, D 2.12, D 2.13

1 Introduction

The term Ambient Intelligence has been adopted by the European ISTAG [ISTAG] (*Information Societies Technology Advisory Group*) to refer to unattended applications that are executed on devices placed in the environment, and which collaborate among themselves to perform complex tasks. According to the definition of AmI by the ISTAG, every AmI device must fulfil at least three properties. The first of these is the *ubiquitous computing* property, which means that all devices must have computing capabilities. The second is the property of *ubiquitous communication*, which is defined as the ability of AmI devices to communicate with other devices everywhere. Finally, the third property is the provision of *natural user interfaces*, which implies that user interfaces must be non-intrusive and user friendly as in they must be capable of gesture or speech recognition.

In addition to these properties it is important to observe that the software and hardware technology of AmI devices is constantly evolving. In order to reflect the importance of this feature in AmI systems, AmI applications are characterized by an additional property, which we name *dynamic evolution*. This property forces the applications developed for these devices to be highly evolvable and adaptable, in order for them to be executed in several kinds of devices with different resource capabilities. In our opinion, this property can only be achieved if the appropriate software technologies are used, which support the dynamic evolution of the hardware by improving system modularization which increases reusability, adaptability and extensibility.

Unfortunately, nowadays most of the effort in AmI applications concentrates itself on developing the most sophisticated applications in order to corner the market, and the profits. Consequently, very complex applications are created, but little effort is put into assuring the compatibility between these applications and future versions, that is, in adequately managing application evolution.

Some platforms like PCOM [Becker, 04], Aura [Sousa, 03] or Gaia [Georgantas, 04] try to overcome this limitation by using advanced software technologies like the CBSD (Component-Based Software Development) [Szypersky, 02]. However CBSD alone is not enough to achieve the appropriate modularity necessary for minimizing the impact of evolution on already developed applications. This is due to the presence of extra-functional properties that usually crosscut several components which makes it difficult for them to be reused in different contexts. In this sense, the AOSD (Aspect Oriented Software Development) [AOSD] paradigm aims to improve system modularization by extracting these crosscutting properties in a new entity called an aspect. An aspect is a property that crosscuts several components. Moreover, it is advisable to model any property that is evolvable over time as an aspect. AOSD proposes that the evolution issues should be managed inside aspects independently of the component or components that are affected by the property. Some examples of properties which we think may be modelled as aspects in AmI applications are: device and service discovery, communication and persistence.

With the aim of studying the benefits that the AOSD approach can provide to AmI applications, our starting point has been the study of the PCOM platform. We have selected this platform since it is one of the most referenced platforms in the AmI community, and because its authors have provided us with its source code, making our study possible. In previous work [Fuentes, 05], we refactored this platform identifying some aspects that were detachable, and also some new aspects that were not considered by the current release of the PCOM platform. We arrived at the conclusion that by applying aspects it was possible to alleviate some of the platform limitations facing both the management of application evolution and the platform adaptation to new technologies.

Consequently, we have defined a new platform (DAOPAmI) that combines CBSD and AOSD approaches, putting their mutual benefits to the service of AmI applications [Fuentes, 04]. In this paper we are going to show how aspects improve AmI application modularization, and therefore, reusability, adaptability and system evolution, by carrying out a comparative study of the DAOPAmI and the PCOM platforms.

After this introduction, in section 2 we study the PCOM platform, showing an AmI application that displays PowerPoint presentations on different devices as a case study. Next, in section 3 we describe the DAOPAmI platform and show the benefits of applying aspects to components by implementing the same application as in PCOM. In section 4, as part of the comparative study, we identify the impact of performing different evolution changes in applications developed on top of both platforms. In section 5 we will show some related work. We will finish in section 6 with some conclusions and an outline of our plans for future work.

2 PCOM

PCOM [Becker, 04] is a component-based middleware platform developed in Java and designed to create unattended autonomous applications that communicate with each other in order to perform collaborative tasks. The PCOM architecture, shown in figure 1, is structured in two well differentiated parts. On the one hand, the lower part, named BASE, manages the communication among devices. On the other hand, the upper part, named PCOM, offers high-level programming abstraction to application programmers. In the following section this architecture will be described in detail.

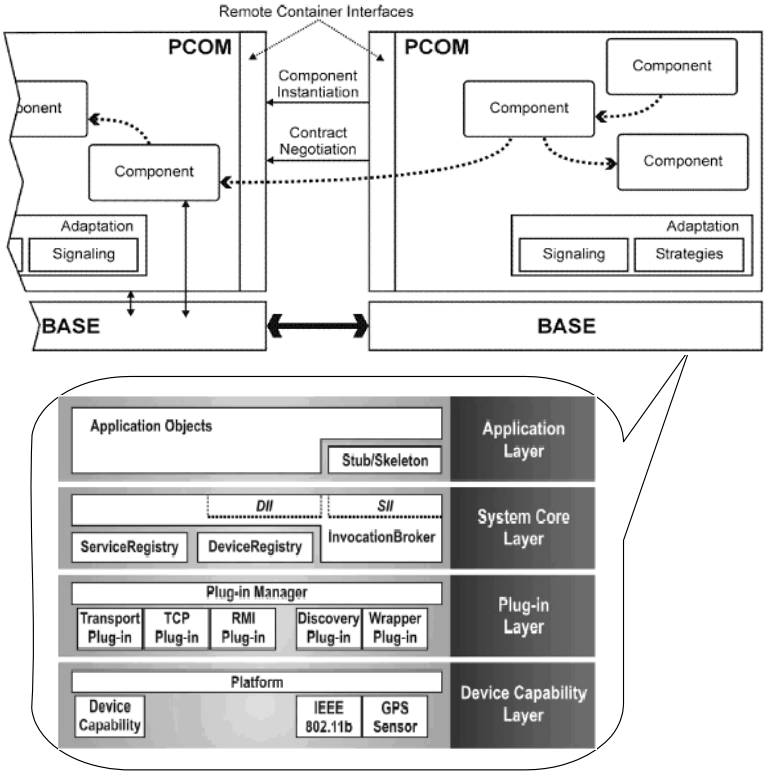

Figure 1: PCOM Architecture

2.1 PCOM architecture

PCOM applications are defined in terms of a set of components that are deployed inside PCOM containers. PCOM containers provide a set of services for the: (1) instantiation; (2) adaptation, and (3) communication of components.

With respect to the *instantiation service* of components a PCOM container defines a remote interface used to: (1) expose the contracts of local components. Contracts define the component name, implementation, interface, resources and their dependences with other components (see the contract example shown in figure 2); (2) offer services to other containers for instance to negotiate the component contract, and (3) instantiate remote components. Before instantiating a component, PCOM examines its contract dependences and tries to find the required components and resources in any local or remote PCOM container.

Therefore, to execute an application, PCOM has to examine all the available component contracts, verifying that all dependences are satisfied. Additionally, PCOM components define proxies to communicate with other components and the instantiation service will initialize them following the component contract definition.

Although the PCOM approach to instantiate and deploy components is similar to the component platform EJB/J2EE, it does not consider the definition of the application architecture (AA) as for example CCM/CORBA does. This makes it difficult to provide a complete view of the application since it is spread among the component contracts, which can even be distributed in different devices. So, as the AA definition is split into different contracts there is not a single description that indicates how components are put together in order to set up the final application.

With respect to the *adaptation service*, PCOM defines a signalling based adaptation mechanism. This mechanism provides several strategies to support automatic adaptation in cases where the execution environment changes.

```
<?xml version="1.0" encoding="UTF-8"?>
<CONTRACT>  <IMPLEMENTATION NAME="Control"
        FACTORY="pcom.component.DefaultFactory"
        IMPLEMENTATION="info.pppc.demo.powerpoint.control.ControlComponent"
        INTERFACE="info.pppc.demo.powerpoint.control.ControlInterface"
       SKELETON="info.pppc.demo.powerpoint.control.ControlSkeleton"
       CONTRACT="info.pppc.demo.powerpoint.control.ControlContract"/>
 <FACTORY_DEMAND/>
  <INSTANCE_PROVISION/>
  <INSTANCE_DEMAND><COMPONENT NAME="Infrared"
         PROXY="info.pppc.demo.powerpoint.control.InfraredProxy">
     <PARAMETER NAME="ID" TYPE="INTEGER" COMPARATOR="EQUAL"
                  VALUE="0"/>
     <INTERFACE TYPE="info.pppc.demo.infrared.Infrared"/>
     <EVENT TYPE="info.pppc.demo.infrared.InfraredEvent"/></COMPONENT>
  ...</INSTANCE_DEMAND></CONTRACT>
```

Figure 2: Control component contract

For example, an adaptation may be required when a remote component is no longer available or a component providing better quality of service (for example more processing speed) is found. These strategies are implemented in the container using plugins, but they cannot be changed once the application starts to be executed. So, if the user wants to implement a new strategy, he has to implement a new plugin, add it to the container, recompile the application and redeploy it again on the device.

Furthermore, the new strategy will be scattered between the application functionality and the adaptation service, making it very hard to reuse in other applications.

Finally, with respect to the *communication service*, all communications in PCOM are delegated to the BASE middleware. BASE supports the sending of synchronous and asynchronous messages among components. Additionally, BASE is able to re-establish communication with other devices when communication errors occur. This capacity is very important in AmI environments where communications among applications are established spontaneously and are continuously changing. To achieve this flexibility, BASE models the different communication protocols using plugins, as shown in the *Plug-in Manager* in figure 1. Although this mechanism allows us to easily add implementations of new communication protocols, one single application normally uses only one. In the latter case, it would be a waste of application resources to use the *Plug-in Manager* for only one protocol. One possible solution to this problem, using aspects, was proposed in [Fuentes, 05].

Now we are going to show a more detailed view of the BASE architecture. BASE is composed of several layers. The first layer, the application layer, upon which the component model of PCOM is built, offers PCOM containers access to the basic communication services. The second layer, named System Core Layer, models three basic BASE objects. The *ServiceRegistry*, which registers the platform basic services used by the containers; the *DeviceRegistry*, which maintains a list of other PCOM devices in the environment and, finally, the *InvocationBroker*, which manages communication between PCOM containers. The third layer, named the Plug-In layer, manages different plugins that implement the services provided by BASE. Finally, the Device Capability layer represents the implementation of each plugin interacting with the available hardware of the device.

We can observe that PCOM offers a pre-defined and fixed set of services, identified by an identifier at the platform level. This makes it difficult for new services, such as security, fault tolerance or persistence to be incorporated into the platform, without modifying the current release of PCOM. Finally, one good feature of PCOM is that it does not require any central or coordination element as is required in other environments like Aura or Gaia. So, each device manages all connections and interactions with the environment by itself. Thus, this architecture is adequate for producing autonomous applications that are executed in resource-constrained devices that do not rely on additional infrastructures.

2.2 Remote control application example

In this section we are going to show an example of a distributed PCOM application, deployed with the current PCOM release. The example uses two AmI devices, each of them executing part of the application functionality in their respective PCOM container. The first container, which we call the *display container*, is executed on a device that displays images on a screen. The second one, which we call the *control container*, is executed on a device that is able to load and execute PowerPoint presentations. These presentations are loaded from the local file system of the device. Additionally, both devices support infrared communications.

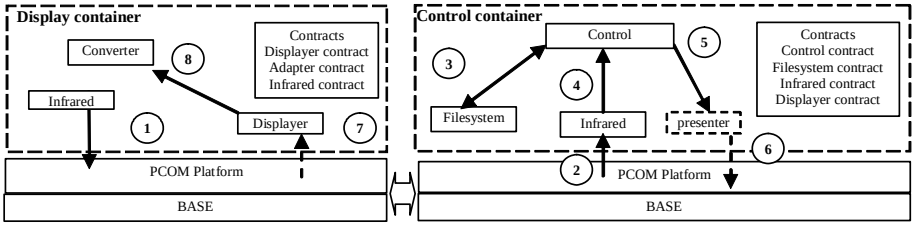

Figure 3: PCOM Example

Figure 3 shows the initial configuration for both containers and their components. In order to start the application in the display container, a *Converter*, a *Display* and *Infrared* components are created. The *Displayer* component shows images on the device screen, while the *Converter* component adapts the received data, (PowerPoint slides) to images that fit on the screen modelled by the *Displayer* component. We must indicate that there is a reference, between the *Displayer* component and the *Converter* component that allows them to communicate. As a result, both component implementations are dependent on each other. Finally, the *Infrared* component continuously sends a signal containing a local display identifier (an integer) to the environment.

With respect to the control container, the application is started initializing the *Filesystem* component, the *Infrared* component and the *Control* component. The *Filesystem* component provides navigation capabilities on the local file system and is able to load PowerPoint presentation files whereas the *Control* component is able to execute these files. This component provides a graphic interface that allows the user to perform basic commands like load, start or stop the presentation. Finally, the *Infrared* component detects remote *Infrared* components and retrieves the identifier sent by them, establishing a connection between the *Control*, and the *Displayer* components. This connection is maintained by the *Control* component using a proxy, called *presenter* that is shown in figure 4. As mentioned previously, components reflect their dependences with other components using contracts: One of these contracts, where the *Control* component demands an *Infrared* component, was shown in figure 2.

```
public class ControlComponent extends DefaultInstance
        implements ControlInterface, CommunicationListener {

//A proxy to the file system.
private FilesystemProxy filesystem;
//A proxy to the presenter.
private PresenterProxy presenter;
//A proxy to the infrared receiver.
private InfraredProxy infrared;
//The dialog of the component.
private ControlDialog dialog;
…
}
```

Figure 4: Control Component Definition

Now we are going to describe how the application is executed in the displayer container. Initially, the *Displayer,* the *Converter* and the *Infrared* component are instantiated and this last one starts to send the identifier to the environment (step 1).

Meanwhile, in the controller container the *Control* component is instantiated and the *Filesystem* and *Infrared* components are created. After initialization, the *Infrared* component will search for other infrared beacons in the environment (step 2). If one *Infrared* component is found, the connection between the *Control* component and the *Displayer* component will be established. Then, when the *Control* component loads a new presentation using the *Filesystem* component (step 3), the local *Infrared* component establishes a connection with the previously found display container (step 4). Each time that the *Control* component performs an action with the loaded presentation, it sends a command (step 5) to the active *Displayer* component to update its content using the BASE middleware (step 6). Then, the *Displayer* component receives the message, containing presentation data (step 7) and sends it to the *Converter* component. Subsequently, this component adapts and returns the data to the *Displayer* component (step 8), which finally shows it. Additionally, if the control device points to another display device, the connection will be redirected and the presentation data will be sent to the new *Displayer* component, which will start to show the presentation. The old display device, after noticing that the connection has been lost, will release the presentation data.

Through this example we can observe that the use of CBSD in PCOM provides good application modularity splitting the application functionality into several components. However, this is not enough because we also found several limitations when we tried to add new functionality to the application or to reuse the defined components in new applications. These limitations are due to the tangled code present in components. For example, in figure 4, we observe that the *Control* component implementation uses hard code references to access the components required by this component contract. As an example, the presenter proxy is a reference to the *Displayer* component. Hence, although contracts are defined using XML, outside of the component (figure 2), each component maintains direct references to other components.

To support this feature, the component must know the implementation of the referred component and, therefore, this fact will make it more difficult to reuse the *Control* component in new applications that present different data formats. Additionally, in this component we found the graphical interface and the presentation control functionality were mixed in the *dialog* attribute shown in figure 4.

3 DAOPAmI

DAOP [Pinto, 04] is a component and aspect based platform created to develop distributed applications. Starting from the lessons learned from the development of this platform we have developed DAOPAmI [Fuentes, 04], which is an adaptation of DAOP to support AmI applications. DAOPAmI has been developed using the Java Micro Edition (J2ME) platform to allow portability among devices. As we said in the introduction, aspects are extra-functional properties that crosscut several components from which it is advisable that they can be detached. Extra-functional properties should be modelled as independent aspects to increase the reusability of both components and aspects and allowing the adaptation of AmI applications to such

evolving technologies, without affecting the component core functionality. Examples of properties that are usually modelled as aspects are communication (e.g. Bluetooth, 802.11, etc.), persistence (file systems, databases, etc.) or fault tolerance.

3.1 DAOPAmI architecture

Figure 5 shows the architecture of the DAOPAmI platform. The DAOPAmI platform is divided into two main levels. In the upper level, a DAOPAmI application is built in terms of a set of components and a set of aspects (upper part of figure 5). Additionally, an XML file describes the architecture of the DAOPAmI application. This file contains information about the components, the aspects, the composition rules and the deployment information that builds up the application. By using this document DAOPAmI provides a full view of the AA, instead of the limited view provided by the PCOM component contracts.

The lower part of the platform contains the core functionality needed to execute AmI applications. This functionality is split into five main parts. The first one is the Application Architecture Manager (or AAM) that loads the AA information at the application start-up and stores it in its internal structures. This information will be consulted at runtime by the platform to perform the dynamic weaving of components and aspects.

The second part, the Component Manager, is in charge of instantiating the application components by using the information provided by the AAM. It also keeps track of the instantiated components and their states. An important contribution of our approach is that DAOPAmI uses a role name to identify components instead of direct code references. The role name of a component is an architectural name (a string) that indicates the role that the component plays in a specific architecture. This means that when a component sends a message to another component it uses the role name to identify the target component instead of using a direct reference, hence solving the direct references problem previously mentioned in PCOM.

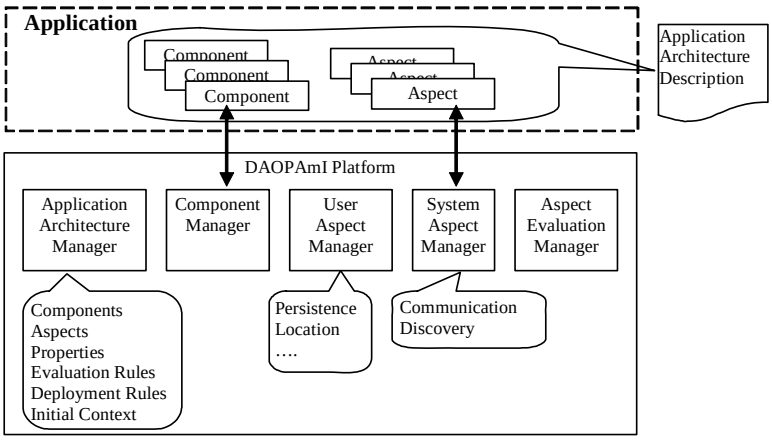

Figure 5: DAOPAmI Architecture

With respect to aspects we differentiate two kinds, system aspects and user aspects. The main difference between them is that system aspects are executed continuously by AmI applications, whereas user aspects are optionally used in applications and can be enabled or disabled during the application execution. In our implementation the system aspects are equivalent to the fixed services provided by PCOM, but with the important difference that the initial set of services offered by the platform can be extended with new aspect definitions. Therefore DAOPAmI defines two aspect managers. The System Aspect Manager (SAM) for system aspects such as the discovery and the communication aspects. And the User Aspect Manager (or UAM) for user defined aspects such as for example persistence or security. Additionally, the aspects defined in the DAOPAmI application level use these system and user aspects and, as a result, they are parametrized in the AA file.
Finally, the last part is the Aspect Evaluation Manager (or AEM), which is responsible for applying the rules loaded in the AAM when necessary. We must indicate that in DAOPAmI, communication among components takes place using synchronous or asynchronous messages and events, and that the evaluation rules are applied dynamically when messages are sent or received by components and also when they are created or finalized. Thus, if we change the AAM information at runtime we will modify the application behaviour automatically, and also how the AEM applies the composition rules.

In developing the DAOPAmI platform we are trying to demonstrate that by using aspects it is possible to solve problems that the current PCOM platform is unable to solve. Consequently, we now discuss the flexibility offered by our approach by presenting some situations in which both the platform and the applications on top of it need to be adapted. Firstly, one common problem when developing AmI applications is that we must support different devices with different requirements. DAOPAmI copes with this problem by the definition of Device Profiles. The device profiles concept is similar to the J2ME profiles. In our implementation the device profile is a file that describes the device capabilities such as CPU type, memory or supported communication protocols. Using this information we can automatically generate a platform version that fits the device characteristics and the application needs. For example, in applications that do not define user aspects we can remove the UAM from the platform. In a device that provides support for only one communication technology, we can replace the default SAM implementation with a more efficient version customized to that technology. Another possible example is to provide a static component and aspect binding if the application does not use the DAOPAmI dynamic composition mechanism. All this customization is possible because the platform configuration had been established externally and is not hard coded.

Secondly, imagine that we need to add a new service, for example a Bluetooth communication service. To add this new functionality, we only need to model it as a system aspect and add it to the SAM. In order to use it, we simply define how to use the service, parametrizing it, and modifying the AA file adequately. This is possible thanks to the use of the AA file that describes the application and provides us with a full view of the AA and, consequently, it is easy to modify it to add the new functionality.

Thirdly, suppose that we need to change an aspect in execution time, for example to adapt HTML presentation data instead of PowerPoint data. If a HTML converter

aspect is available in the application, we can change the AA at runtime, using the AAM, and replace the previous aspect for the new implementation and adapt the application behaviour automatically.

Finally, let's suppose that we want to provide our application with different graphic user interfaces that the user can change at runtime. If we provide different implementations of a GUI component, that maintain compatible interfaces, we achieve this effect in our application changing the default component implementation in the AAM. The rest of the components and aspects are not affected by this change. This mechanism can also be used to change aspect implementations without recompiling anything.

3.2 Remote control application in DAOPAmI

In this section we are going to show how to implement in DAOPAmI the previously presented application. In order to achieve a more adaptable application than the one developed for PCOM, our goal is to modularize the application using components and aspects. Therefore, our first step was to decide which part of the application should be modelled as components and which one as aspects (figure 6). As a general rule, we have decided to model a component as an aspect if its functionality can be seen as a crosscuting property of the AmI application domain or if it is highly probable for it to be replaced due to technological evolution.

With respect to the first container (see figure 3), we will maintain the *Displayer* component because it implements a concrete and independent functionality. However, we should transform the *Converter* component into an aspect. We consider it an aspect because first, it can evolve independently from the other components in the application in which it is used, and secondly, the application can be executed without the *Converter* functionality. Additionally, modelling this component as an aspect we can develop other *Converter* aspect versions that will be able to adapt presentation data with several different formats such as html, text files or video to images.

In the case of the *Infrared* component, we have decided to transform it into a system aspect. The main reason is that if the device does not support the infrared communication technology it will be possible to replace it completely with other technologies such as Bluetooth or RFID which provide a similar functionality.

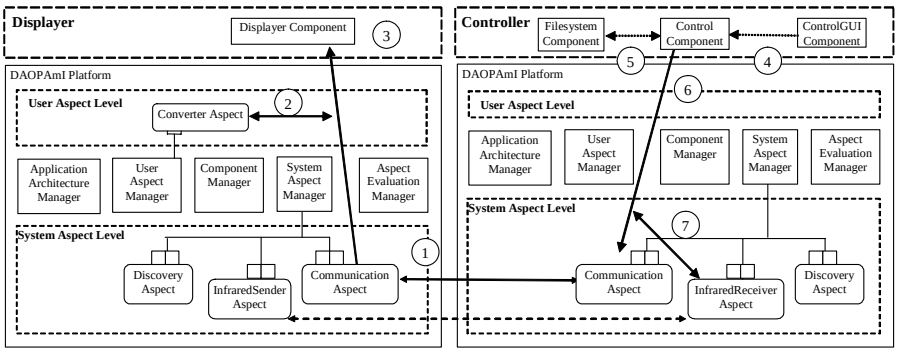

Figure 6: DAOPAmI Example

Moreover, we have modelled this aspect as two different aspects in order to reflect the dual role that it plays in the application. The first aspect, named *InfraredSender* and located in the displayer container will send information to show that the local *Displayer* component is ready to receive presentation data. The second aspect, named *InfraredReceiver* is located in the control container. This aspect determines which displayer device we are pointing at with the control device when a message is sent.

With respect to components in the control container (see again figure 3), we have decided to split the *Control* component functionality into two components. One of them is the *ControlGUI* component, which manages the graphical user interface. The other one is the *Control* component, which manages the controller application logic. This division allows us to have different GUI component implementations, which allows us to give each application a different appearance.

Next, the *Filesystem* component remains unchanged. This component can be replaced by other implementations that will provide support to alternative file systems such as a database or a web server. Finally, the original *Infrared* component is modified and modelled as an aspect named *InfraredReceiver* as mentioned previously.

After separating the functionality of the application into components and aspects the next step to develop a DAOPAmI application is the description of the AA. In figure 7, we show part of its AA configuration. Notice that, every aspect and component in the figure is referred to using its role name and that the composition rules make reference to these names.

```
<applicationArchitecture>
 <components>
  <component role="Control" binding ="STATIC">
    <providedInterface><href>ControlProvidedInt.xml</href></providedInterface>
    <requiredInterface><href>ControlRequiredInt.xml</href></requiredInterface>
    <implementations><implementation name="PowerPoint"><lang>java</lang><class href="controlImpl1.class"/></implementation>
      <default-impl>PowerPoint</default-impl>
    </implementations> </component>
  <component role="ControlGUI">…</component>
  <component role="Filesystem">…</component></components>

<aspects>
  <aspect role="Converter" binding ="DYNAMIC" kind="USER">
    <evaluatedInterface joinPoint="BEFORE_RECEIVE"/>
    <implementations><implementation name="PDA"><lang>java</lang><class href="ConverterImpl1.class"/></implementation>
      <default-impl>PDA</default-impl>
    </implementations></aspect>
  <aspect role="InfraredSender" binding="STATIC" kind="SYSTEM"><parameters><href>commParams.xml</href></parameters></aspect>
…</aspects>

<compositionConstraints>
  <aspectEvaluationRules>
   <BEFORE_RECEIVE>
    <source-comp>Control</source-comp><target-comp>Displayer</target-comp><targetMessages><message name="nextSlide"/></targetMessages>
    <applyAspects><concurrent><aspect role="Converter"/></concurrent></applyAspects>
   </BEFORE_RECEIVE>
  …</aspectEvaluationRules> </compositionConstraints>

<deploymentInformation>
  <deployComponent role="Control" location="local"/>
  <deployComponent role="Displayer" location="remote"/>
  <deployAspect role="InfraredSender" location="local"/>
…</deploymentInformation>

<initialContext>
  <createComponent role="Control" roleInstance="control1"/>
</initialContext>
</applicationArchitecture>
```

Figure 7: Application Architecture in DAOPAmI

There the *Control* component definition is shown in the AA file, enclosed by a *component* tag. This component describes its provided and required interfaces, the messages that it sends and receives, in the *requiredInterface* and *providedInterface* tags. In the example, the component defines only one possible implementation and is declared as having a *STATIC* binding, as a result, it will not be possible to modify its implementation during the execution of the application.

The *Converter* aspect definition is also shown. Its definition, enclosed in an *aspect* tag, indicates that this aspect is *DYNAMIC* and *USER*. This means that the aspect can be removed or replaced during execution dynamically and that the UAM manages the aspect. Additionally, the *joinpoint* property indicates that the aspect will only be evaluated before a component receives a message. A *joinpoint* is an application execution point usually located before or after a component method invocation or before or after a component creation or finalization. In the *joinpoints*, the aspects are evaluated modifying the application behaviour. A more detailed explanation of this part of the AA configuration file can be found in [Pinto, 03]. See also part of the system *InfraredSender* aspect definition (denoted by the *kind* tag value). Additionally, system aspects are parametrized in an independent file (marked by an *href* tag in figure 7). These parameters are usually aspect initialization data or information about data conversion. Additionally, this file provides information about how components and aspects are deployed (*deploymentInformation* tag) and which components must be instantiated by the application (*initialContext* tag).

Finally, the last step is to initiate the application. Notice that once the application is initiated in the devices, the information about the AA is available at runtime. We first describe how the displayer application part works. Figure 6 shows the DAOPAmI application configuration. The application is initiated when the *Displayer* component is created. After creating this component an aspect evaluation rule (not shown in figure 7 for space reasons) indicates that the *InfraredSender* aspect must be evaluated. As part of its evaluation, this system aspect will start to send information about the device availability. Then, if the *Displayer* component receives a message, for example a *nextSlide* message to show a new slide (step 1), the AEM consults the information provided by the AAM, and decides that the *Converter* aspect (step 2) must be evaluated. The aspect is evaluated before receiving the *nextSlide* message in the *Displayer* component and (step 3) will adapt the presentation data to the data format expected by the *Displayer* component. In figure 7 we can see this behaviour reflected in the *BEFORE_RECEIVE* rule.

In the controller part, the application is executed as follows. First, the application instantiates the *Control* and the *ControlGUI* components. After creating the *Control* component, the *InfraredReceiver* aspect is evaluated. This system aspect will try to find an *InfraredSender* aspect, located in a remote displayer device. If it succeeds, the aspect keeps the information about the displayer device in order to be used later. Otherwise, the user will be asked to point to a valid displayer device on which the presentation will be shown. Next, using the *ControlGUI* component the user loads a presentation file. To do this, the *ControlGUI* component sends a *loadfile* message to the *Control* component (step 4), which upon receival then sends a *retrieveFile* message to the *Fylesystem* component (step 5). This component retrieves the file data and sends a *retrieveFile* message to the *Control* component. Finally, the *Control*

component instantiates the PowerPoint presentation. All these steps are shown in figure 8 with part of the *Control* component implementation explained later.

Now, if the user wants to show the loaded presentation he points to a displayer device and presses the next slide button shown by the display of the *ControlGUI* component. This component sends a *nextSlide* message to the *Control* component that sends another *nextSlide* message, with the slide data as parameter, to the remote *Displayer* component (step 6). But before sending this message, the *InfraredReceiver* aspect is evaluated (step 7) to determine if a valid displayer device has been selected.

If so, the message will be delivered to the target component using the data previously stored by the *InfraredReceiver* aspect. Let us suppose now that the user points to a new displayer device and shows a new slide. In that case, the *InfraredReceiver* aspect will notice that the identification data provided by the new *InfraredSender* aspect differs from the information that it has. So, the aspect will send a message to the previous *Displayer* component ending the presentation. Then it will update the current displayer device data, and, it will send a *nextSlide* message to the new *Displayer* component changing the target component dynamically.

To conclude this section, we must remark that using the DAOPAmI approach the *Control* component does not contain direct references to any other component or aspect. It only uses role names to communicate with other components as is shown in figure 8, where the first argument of the *execute* method indicates the role of the target component that will receive the message indicated by the third argument. Therefore, in the *Control* component there is no reference to the *InfraredReceiver* and, thus, it will be possible to completely replace the *InfraredReceiver* and the *InfraredSender* aspects by other equivalent ones.

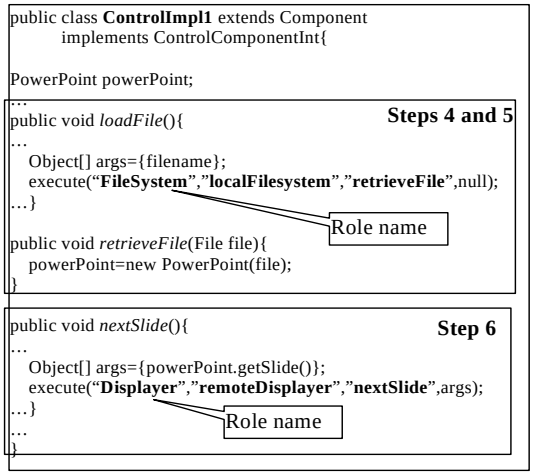

Figure 8: Control Component Implementation

4 Comparing PCOM and DAOPAmI

After describing both approaches we can conclude that both platforms show reliable solutions to develop AmI applications. But there are some problems related to PCOM and some advantages that make the DAOPAmI platform approach more flexible. Now we are going to comment on these problems and advantages.

In the PCOM application it is difficult to obtain a complete view of the application architecture since each component acts like an independent application trying to find the appropriate resources in the environment before execution starts. This behaviour is indicated in their contract making it difficult to figure out what the application is trying to do only by examining the individual component contracts. Additionally, there is no explicit information about which messages can be interchanged by components. In DAOPAmI the application behaviour is clearly expressed using the AA XML file and all messages sent and received by components are expressed explicitly in the AA. Also the rules that drive the composition between components and aspects are expressed in the AA information.

Another problem in PCOM is that it provides common functionality to AmI applications using a fixed list of platform services that are modelled as plugins. It is possible to add new implementations of these services using plugins, but if we need to add a new common functionality, for example authentication, we have to modify the platform implementation in order to integrate it into the core functionality. Moreover, the provided services cannot be changed once the application is started. In DAOPAmI common services are modelled as aspects, with two main advantages. The first one is that the user can add new aspects not implemented by the platform such us authentication, access control, etc. So, we can add new functionality to applications without modifying the existing components. In addition, in DAOPAmI we can change the current component or aspect implementation at runtime, modifying the application behaviour dynamically. This flexibility and adaptability is impossible to achieve in PCOM.

Another problem in PCOM is that components manage communication using proxies to send and receive messages. As was shown in figure 4, this implementation solution introduces dependences among components. So, if some component implementation is changed or we need to add a new component to the application, we have to modify the component contracts, change the component code and recompile the application to update it. This makes the component less reusable and the application difficult to modify, maintain and evolve. DAOPAmI solves this problem using role names to refer to components instead of hard coded references. In this example, this change may affect only the control component, but in more complex systems this change will affect a lot of components.

An additional advantage of DAOPAmI is derived from the combined use of device profiles and the description of the AA that helps us to automatically generate a platform implementation that fits the target device capabilities. This reduces the application size because we only include the platform parts that are needed. Moreover both the aspects and components are integrated statically or dynamically depending on the AA file specification. As a consequence, we get a better application performance in resource-limited devices that execute AmI applications. PCOM does

not provide such a mechanism and thus the developer must decide which functionality must be included in the deployed platform.

5 Related work

An example of aspect-based AmI development platforms is MIDAS [Falcarin, 04]. This platform tries to solve the dynamic adaptation problem, but the platform does not consider all the specific problems associated with AmI devices. For example, MIDAS solves the configuration problem using dynamic aspects and changing the configuration of the application at runtime, but it relies on the use of a listener register in a remote server to notify of application changes. This continuous connection to receive notifications is not always possible in AmI applications and the dynamic loading and unloading of application components is based on a reflection mechanism that it is not usually available in all AmI devices.

Other work for developing an AmI application using aspects is being conducted by [Young, 05]. This work is centred around the implementation of AmI application product lines. These product lines develop complete applications considering the device capabilities and encapsulating the different functionalities inside of aspects. So several versions of the same application can be obtained and the aspects can be reused in new applications. Unfortunately, the use of AspectJ [AspectJ, 05] as an implementation language does not provide support for the dynamic adaptation of applications.

6 Conclusions and future work

In this paper, we have shown two applications that provide the same functionality using two middleware platforms which provide different approaches to solving the AmI development problems. We have proven that the main difference between them, the use of aspects, is a key concept to solve the problem of software evolution and application adaptation over time.

Although today several frameworks exist which are suitable for supporting and developing AmI applications, these frameworks do not consider the Dynamic Evolution problem. The only option for handling this problem is rewriting, recompiling and completely replacing the old application for a new version. DAOPAmI tries to overcome this problem thanks to the combined use of AOSD and CBSD.

Currently we are working on the complete implementation of the DAOPAmI platform and the development of tools that helps in the automatic generation of different middleware platform versions using device profiles suited to each AmI device.

Acknowledgements

This work is partially financed by IST-2-004349-NOE AOSD-Europe and the Spanish Ministry of Technology and Science, CICYT, under grant TIC2002-04309-C02-02. We want to give special acknowledgments to C. Becker and M. Handte for

providing us with the PCOM source code. This code has helped us to identify specific aspects in the AmI domain.

References

[AspectJ, 05] AspectJ Web Site, 2005, http://eclipse.org/aspectj

[AOSD] Aspect-Oriented Software Development Web Site, 2005, http://www.aosd.net

[Becker, 04] C. Becker et al. PCOM – A Component System for Pervasive Computing, In Proc. Of second Int. Conf. on Pervasive Computing, 2004, Orlando, Florida, United States of America, 14-17 March

[Falcarin, 04] P. Falcarin, G. Alonso, Software Architecture Evolution through Dynamic AOP, European Workshop in Software Architectures, 2004, St. Andrew, United Kingdom, 21-22 May

[Fuentes, 04] L. Fuentes, D. Jimenez, M. Pinto, Towards the development of ambient Intelligence Environments using Aspect-Oriented techniques, In proc. of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2004, Lancaster, United Kingdom, 22-26 March

[Fuentes, 05] L. Fuentes, D. Jimenez, M. Pinto, Experiences Refactoring Ambient Intelligence Applications with Aspects, In proc. of the Linking Aspect Technology and Evolution Workshop, 2005, Chicago, Illinois, United States of America, March 14

[Georgantas, 04] N. Georgantas, V. Issarny. User Activity Synthesis in Ambient Intelligence Environments, In proc. of the European Symposium on Ambient Intelligence, 2004, Eindhovem, Netherlands, November 8-10

[ISTAG] ISTAG. Information Societies Technology Advisory Group Web site, 2005, http://www.cordis.lu/ist/istag-reports.htm

[Pinto, 03] M. Pinto, L. Fuentes, J.M. Troya, DAOP-ADL: an architecture description language for dynamic component and aspect-based development, In proc. Of second International conference on Generative Programming and Component Engineering, 2003, Erfurt, Germany, 22-25 September

[Pinto, 04] M. Pinto, L. Fuentes, J.M. Troya. A Dynamic Component and Aspect Oriented Platform. The Computer Journal, Vol. 48, number 4, Pp. 401-420, 2004

[Sousa, 03] J.P. Sousa, D. Garlan. The Aura software Architecture: an Infrastructure for Ubiquitous Computing. Technical Report. CMU-CS-03-183, 2003

[Szypersky, 02] C. Szypersky, Component Software. Beyond Object-Oriented Programming, Addison-Wesley, ACM press, 2002

[Young, 05] T. Young, G. Murphy. Using AspectJ to Build a Product Line for Mobile Devices. Demonstration in Aspect Oriented Software Development Conference, Chicago, Illinois, United States of America, March 16