Aspect-Based Adaptation for Ubiquitous Software

Arturo Zambrano[1], Silvia Gordillo[1,2], and Ignacio Jaureguiberry[1]

[1] LIFIA, Universidad Nacional de La Plata
50 y 115 1er Piso
1900 La Plata, Argentina
{arturo,gordillo,jauregui}@lifia.info.unlp.edu.ar
[2] CIC, Provincia de Buenos Aires

Abstract. Information should be available everytime and everywhere in the ubiquitous computing world. Environment conditions such as bandwidth, server availability, physical resources, etc. are volatile and require sophisticated adaptive capabilities. Designing this kind of systems is a complex task, since a lot of concerns could get mixed with the application's core functionality. *Aspect-Oriented Programming* (AOP) [1] arises as a promising tool in order to design and develop ubiquitous applications, because of its ability to separate cross-cutting concerns. In this paper we propose an AOP-based architecture to decouple the several concerns that ubiquitous software comprises.

1 Introduction

An ubiquitous application should be highly adaptable, since it will be exposed to a world where runtime conditions change continuously. It must be able to face resource variability, user mobility, user's changing needs, heterogeneous networks and so on, by adapting itself as automatically as possible. As a consequence of the high number of concerns that must be modeled and the manner in which they interact, this kind of system is prone to mismatching designs.

By *adaptive capability* we mean the system's ability to adapt itself to new run-time scenarios, such capabilities which cope with specific issues (for instance: networking, system faults, etc.) should be applied in an automatic way, so that the user is not disturbed. Furthermore, *adaptive capabilities* should be incremental, that is, they should evolve in runtime, catch and store information regarding the system's context for further use.

It is desirable for the adaptive capabilities and the system's core functionality to be handled orthogonally, so that they can evolve individually and promote system's flexibility. Besides, adaptive capabilities should be isolated from each other as much as possible, in order to avoid conflicts among them and to promote the reuse of such capabilities across families of systems.

An aspect-oriented design could lead us to a better separation of concerns for self-adaptive ubiquitous applications, by isolating the different features composing them.

In this paper we present our approach to separate adaptive capabilities from the system main functionality. Section 2 and 3 present concepts related to ubiquitous computing and aspect-oriented programming. In section 4 we present our approach through

F. Crestani et al. (Eds.): Mobile and Ubiquitous Info. Access Ws 2003, LNCS 2954, pp. 215–226, 2004.

an example. The next section presents an analysis of advantages and disadvantages of this approach. After that, a comparison against an OO approach is presented. Section 7 presents implementation issues. Finally, we state our conclusions.

2 Adaptation in Ubiquitous and Mobile Computing

An ubiquitous computing system consists of a (possibly heterogeneous) set of computing devices; a set of supported tasks; and some optional infrastructure (e.g., network, GPS location service) the devices may rely on to carry out the supported tasks. [2]

Several approaches have been proposed to construct ubiquitous software artifacts. As expressed in [3] an architecture-based adaptation could be used to model adaptive systems, but in this approach most layers composing the system are aware of the existence of the others. In this way, changes in one layer could affect the others. It is desirable to use a transparent adaptation mechanism, where adapted components and components dealing with adaptation are independent.

In [11] a reflection-based approach is presented in order to modify application's behavior to adapt it to changing network conditions. It is called reflective architecture, and it allows to perform self-modifications of existing behavior. At the same time, it allows to separate system and mobility adaptation policies through a collaboration interface. We propose the use of AOP as way to enhance the independence between the system and adaptation mechanisms, and the use of the aspect-based adaptation to deal with all the concerns regarding context-awareness.

To adapt system's behaviours it is necessary to know the environment which surrounds the system. The set of properties characterizing the environment defines its *context*. A more formal definition of context is given in [4], where context is defined as: *"the reification of certain properties, describing the environment of the application and some aspects of the application itself"*. *Context* often comprises properties related to spatial and temporal positioning, networking, device constraints, user's needs and the application. A detailed study of *context* is given in [4] and [5].

Efficient execution of mobile systems requires adaptations in harmony with current context, for instance, as it is proposed in [12] *we might choose to have context feature that excludes content based on file-size, such a context feature should be active if the user is using a low bandwidth connection, but it should remain quiescent if there is a high bandwidth conneciton available*.

3 Aspect-Oriented Programming

In the application development process, it is common to find a set of concerns which are independent of any application domain and that affect many objects beyond their classes which constitute (in object-oriented programming) the natural units to define functionalities. They are called cross-cutting concerns.

A *cross-cutting concern* is a concern that is spread along most of the modules of a system. Typical cross-cutting concerns are *persistence, synchronization, error handling. etc.* As it is said in [6]: *"...existing software formalisms support separation of*

*concerns only along a **predominant dimension** neglecting other dimensions... with negative effects on re-usability, locality of changes, understandability...*". These secondary dimensions correspond to cross-cutting concerns. This idea is specially applicable to ubiquitous software, where a lot of dimensions are present.

Aspect-Oriented Programming (AOP for short) [7] is one of many technologies resulting from the effort to modularize cross-cutting concerns.

The intuitive notion of AOP comes from the idea of separating the several concerns that are present in any system. For instance, imagine a system where many *logging* operations are performed in order to track system flow control. In such a case, logging sentences are scattered along the modules of this system (e.g. `printf` for a C implementation). The *logging concern* does not have a materialization in this system, making its maintenance difficult (just imagine if it is necessary to change a parameter passed to the `printf` function, due to a change in the form that logging must be done).

The goal of AOP is to decouple those concerns, so that the system's modules can be easily maintained. AOP introduces a set of concepts:

Join Point A join point is a well-defined point in the program flow (for instance a method call, an access to a variable, etc)
Point-Cut A point-cut selects certain join points and values at those points.
Advice Advises define code that is executed when a point-cut is reached.

The program whose behaviour is affected by aspects is usually called *base program*. A join point is a concept which allows specifying points in the execution of the base program that will be affected by an aspect. One or more of these join points (from one or different classes) are identified by a point cut in the aspect layer, associating it with an advice. In this way, when one join point, defined in a point cut, is reached in the program execution, the additional code, defined in the correspondent advice is executed, adapting the original behaviour according to the current aspect. The aspect's code is composed of advises and the point-cuts where those advises must be applied. Advises could be compared to methods (in the *object-oriented* paradigm) defined within the aspects. When using aspects, the idea is to modularize cross-cutting concerns as aspects, which contain the code to handle the concerns. Since the concern is a cross-cutting one, it is necessary to apply the behaviour defined in the aspect in several places of the base program. This is done by defining the join-points and point-cuts that refer to the base program, and linking the code of the advises to the proper point-cuts.

As it will be shown in the next sections, we have used these AOP concepts to adapt the behaviour of an ubiquitous system to different runtime environments.

4 Decomposing Ubiquitous Software Using Aspects

We propose the use of AOP to separate the core functionality concern from the *context-awareness* concerns during design and implementation time, so that the corresponding software structure can evolve independently. By using AOP, the core application can be adapted in a transparent way, since it is not aware of context constraints. At the same time, the abstraction from those details makes the core application easier to design and implement. By encapsulating the adaptation mechanism and separating it from the base

application, a more reusable context representation and adaptation mechanism can be obtained.

4.1 Exemplary Application

To illustrate our approach we will use the following example:

> We must face the design of a personal assistant application for tourism. The aim of our application is to provide the user with relevant information about the place where he is in, for instance, accommodation locations, restaurants, museums, etc. Furthermore, it must report the user's current location.
>
> Implementations of the application must be able to run on a desktop computer, a laptop and PDAs, using wired or wireless connections to the servers. There might be a lot of servers which provide tourism information to the mobile client. It is supposed that a client application can connect to a different server according to the client's geographic location. Since there are different resource availability for each type of client (screen resolution, processing power, memory, etc), and there are other runtime changing issues such as bandwidth, location, etc., the whole system should be able to adapt itself to provide information in the proper way.

The natural architecture is a client-server one, where constraints associated with ubiquity make it more complex. From the client application's point of view, the designer must be conscious of:

- User's mobility: this affects the information that must be requested to the server and displayed. For instance, as the user goes on his trip, the system should report different accommodation vacancies for different cities.
- Variability of resources: the client application running on different devices is capable of using different resolutions to show graphics, variable available memory, etc.
- Variability of available bandwidth: the information should be available on time, therefore the client application should request information sized according to the connection's throughput.

We will analyze the impact of an AO design to reach a better separation of the concerns involved.

Identifying System's Concerns. The system's functionality can be summarize as *to provide the user assistance during a trip, according to some quality attributes: performance and reliability, across changing computational environments. The application relies on several servers that provide requested information*.

To cope with this general requirement, we must analyze which concerns are present. As a preliminary list of concerns of this application, we find the following:

1. System's core functionality: tourism assistant.
2. Visualization Concern: it means that information should be obtained in a format (textual, high or low resolution graphics) that can be displayed by the device.
3. Communication Concern: it means that communication should be optimized according current networking connection.
4. Memory Consumption Concern: this concern refers to the fact that requested information can be stored by the device.
5. Spatio-Temporal Concern: this concern affects the information requested since the system handles spatio-temporal positioned information.

Assuming that the object-oriented paradigm was chosen to model the application we must answer the following questions: *Which of these concerns will be modeled as aspects? Which of them as objects? How is their behaviour related?*

Most activities will be handled as requests made to the nearest server, whose results are presented to the user. It seems to be clear that the last four concerns affect the behaviour of the system's core (which is represented by the first concern), by modifying the way in which information is required. For instance:

- Spatio-Temporal Concern: affects the system by modifying its requests to reflect the current location, so that the server can return accurate information for this location. Geographic positioning can also be used to select the proper server.
- Communication Concern: this concern must deal with available connectivity and users' needs. This concern must modify requests according to current network throughput. For instance, if the user asks for a map, this concern could change the requested resolution for the map, in order to keep the network use within certain bounds.
- Visualization and Memory Consumption Concerns: these are similar to the previous case; here the concerns should modify the request in order to fit current device capabilities.

It would seem that there is a predominant dimension [6][8] where the system's core is located. Other dimensions correspond to those concerns that have some effect on the predominant one. Since these concerns modify system's behaviour for each request (see Figure 1), and they represent different topics of system's adaptation, we have decided to model them as *aspects*, leaving the core system's model as an object model. In fact, the *context model* is an object-oriented one, and the aspects (joint points, point-cuts and advises) are used as *glue* to attach the adaptive behaviour in a seamless way.

Modular Division of System's Functionality. We will focus on the client-side which has to provide pervasive features. As far as this example is concerned, the server-side is composed of a net of servers providing the information that is requested by the clients. Figure 2 depicts a simplified version of the system's architecture (client-side), where the class `Tourism Assistant` represents the base application. The *base* application's interface consists of a set of messages that obtain information from some server. The actual request should be adapted to fit current runtime constraints and user's needs, so that it is affected by the aspectual layer, which takes runtime information from the

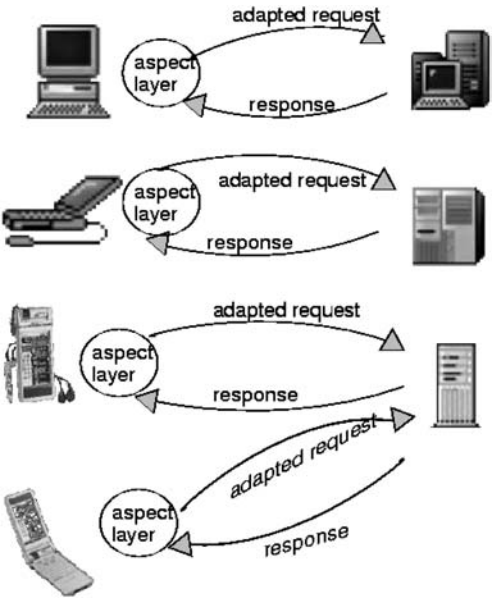

Fig. 1. The Aspectual Layer adapts client's requests

`Context` model. This is an standard object model which holds information about the current system's environment. In Figure 2 it is presented by a single class, but it is indeed a more complex representation of the reality. This model should be shared by all the aspects, so that they can *see* the same scenario.

The notation in Figure 2 has been taken from [9] with minor modification: the label `request*` indicates that the point-cut involves all the messages starting with the `request` word. Each invocation to those messages is intercepted and automatically adapted by the aspectual layer, since *requests* are defined as *point-cuts*. As it can be seen in Figure 2, the system's architecture is divided into three layers. The first layer corresponds to the base application, where no assumptions are made with respect to runtime environment constraints. The second layer is the *Aspectual Layer*, which contains the adaptive behaviour, ie. base application's behaviour is modified in a transparent way through the *point-cut* mechanism. The last layer is the context-aware one, which feeds the *aspectual layer* with runtime information.

We have analyzed how `requests` are affected by several concerns. This analysis can be extended to the remaining system's functionalities that should be adapted to the runtime scenario.

In this case, *aspects* have been used as a means of adapting the application's behaviour to the current context in runtime. They constitute a layer that provide a completely transparent instrument to obtain this adaptive behaviour. Therefore, the core application can be easily designed and implemented. Furthermore, the base application and the aspectual layer are integrated orthogonally, so that they can evolve independently.

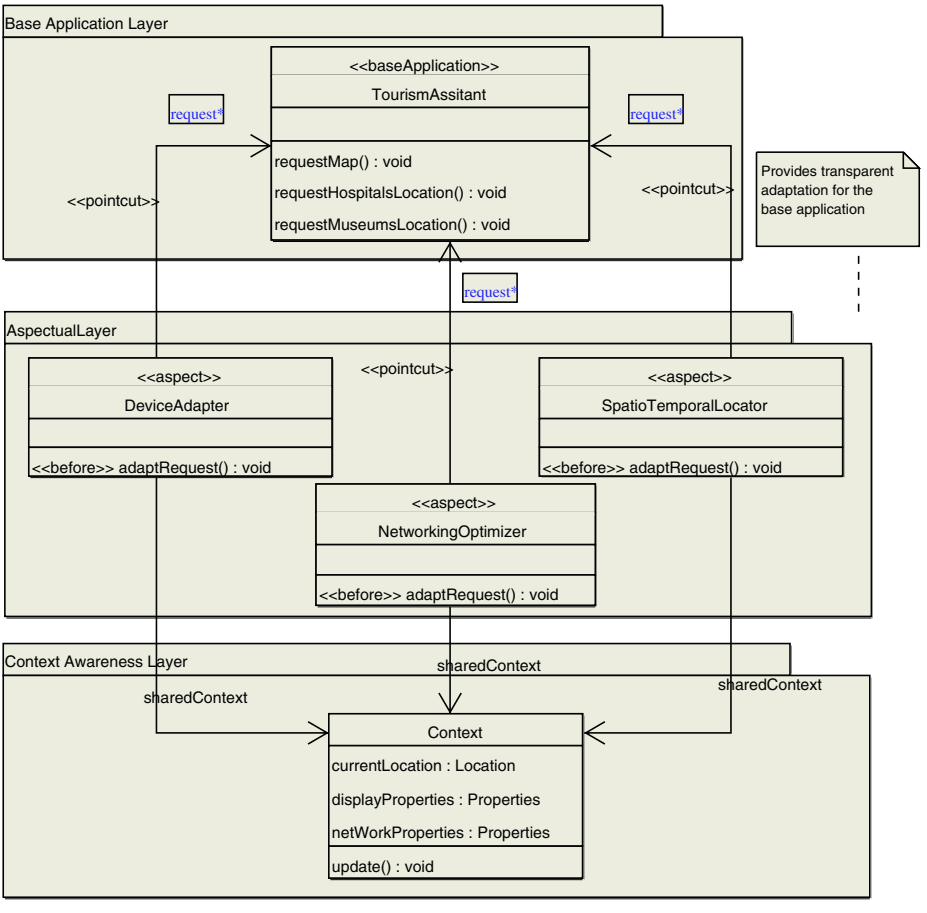

Fig. 2. Simplified Client-Side Architecture

Actual applications developed for desktop computers, laptops, handheld and PDAs may differ in implementation issues, but they can certainly follow this general schema.

Notice that this architecture corresponds to the client-side, where no data will be available at startup, instead, it will be downloaded on demand. Applications following this architecture are able to be deployed in mobile devices using current available technology, such as JVM (J2ME, SuperWaba, etc.) for mobile devices and AspectJ [10]. Since AspectJ generates pure Java code, implementations can run on any platform supporting J2ME.

5 Advantages and Drawbacks

In this section we present some advantages we have found in this approach and drawbacks that should be solved before getting a robust aspect model for ubiquitous applications. We will start by stating some advantages:

- Modifications to adapt the behaviour of base programs are included in the aspectual layer, which is invisible to them.
- Different concerns regarding ubiquity can evolve independently from one to another.
- Since the context representation is stored at the client side, the resulting application is more robust in relation to server failures.
- The separation between the core and adaptive capabilities allows us to reuse context representation and the adaptation strategies.

Some shortcomings have been found:

- Some concerns could require contradictory adaptation strategies and this could origin conflicts among them. For instance: if there is a fast network connection but a poor screen display, then the *network concern* would encourage heavy high resolution images downloads, whilst the *visualization concern* would require low resolution images download. There must be a mechanism to define which concerns take precedence or govern the others.
- In some cases, concern goals should be overridden by user defined goals, this could involve defining explicit interactions from base program toward the aspectual layer. This is not usual in the literature on aspect-orientation. Another approach could be treating user's preferences as a new *concern* modeled through aspects.

6 A Comparison against a Pure OO Approach

At this point it is interesting to discuss why a pure object-oriented is not powerful enough to correctly model this kind of systems.

Object oriented technology has proven to be a useful modelization technique. Having objects encapsulating internal state and behavior is specially useful to model abstractions in one dimension. But, as we stated in 3, abstractions belonging to several dimensions can not be easily integrated in a single unique model. Eventually, crosscutting concerns responsibilities will be spread along main dimension abstractions.

A good object oriented design for the given application might separate context awareness issues, but, in order to get applications behavior adapted by context awareness, some kind of explicit invokation will be needed. This need for an explicit invokation (the unique invokation mechanism available in object orientation) will couple at some point both, application and adaptation models. Finally, this binding between the two models will result in a dependence, making necessary to have both models in order to have a functional application. In the presented approach, the application model is completly unbound from context awareness model, allowing us to design and implement the application, forgetting non-functional constraints (such as context-awareness ones).

7 Implementation Issues

In this section we present some examples regarding how the proposed approach can be implemented using current available programming tools. For the following example

we have chosen AspectJ to provide aspects implementation. The base application can be developed using J2ME or Superwaba (a free open source Java dialect, that runs on palms and handhelds). XML has been chosen as the language used to express client's request.

For the example presented in 4.1 we will analyze the default system's behaviour and then how it is modified by aspects attending to specific concerns.

Let us analyze the request that asks the server for information about the location of the device.

```
1. public String request(){
2.     return "<REQUEST>
3.      <USER_ID>232</USER_ID>
4.      <POSITION_INFO/>
5.      </REQUEST>";
6.   }
```

Fig. 3. Base application XML request

The method `request()` in Figure 3 generates the XML request that will be sent to the server. In this example, it asks for information regarding the position of the mobile device. The expected answer can be a textual description (street names) or a map indicating the current device position in a user-friendly way. The choice between the different representations depends on the display capabilities of the device. Furthermore, in the case of a map, the picture resolution should be chosen according to the device resolution, free memory space and network bandwidth, as we discussed in 4.1.

More specifically, an aspect working on the bandwidth concern can add extra information indicating the current available bandwidth, so that the server can perform the necessary adjustments on the information to be sent as response.

A simplified implementation for this aspect is shown in Figure 4.

```
1. public aspect BandwidthAspect {
2.  String around(): call( BaseApplication.request(..)){
3.   String request = proceed();
4.    request+= "<BANDWIDTH_CONSTRAINTS>
5.     <MAX_SIZE>" + context.currentBandwidth() + "</MAXSIZE>
6.     </BANDWIDTH_CONSTRAINTS>";
7.    return request;
8.   }
9.  }
```

Fig. 4. Bandwidth aspect implementation

Notice that the bandwidth aspect defines a point-cut (Figure 4 line 2) for the method `request()` in the base application, as was expressed in 4.1. When this method is invoked, the control pass to the aspect which modifies the original XML request by

adding parameters indicating the current bandwidth. This information is obtained from the `Context` object that holds runtime information. In Figure 4 line 3 the base application method is invoked and its result is modified in the following lines.

The outcome XML request presented in Figure 5 corresponds to several aspects working on other concerns, such as memory consume, geographical location (through GPS) and image resolution, performing their adaptations.

```
1.<REQUEST>
2.  <USER_ID> 3232 </USER_ID>
3.  <POSITION_INFO> <TYPE> MAP </TYPE>
4.  </POSITION_INFO>
5.   s<CURRENT_POS>
6.            <LAT>20 20' 21"</LAT>
7.            <LONG>24 21' 0"</LONG>
8.  </CURRENT_POS>
9.  <IMAGE_CONSTRAINTS>
10.   <WIDTH>320</WIDTH>
11.   <HEIGTH>200</HEIGTH>
12.  </IMAGE_CONSTRAINTS>
13.  <BANDWIDTH_CONSTRAINTS>
14.   <MAX_SIZE> 2KB </MAX_SIZE>
15.  </BANDWIDTH_CONSTRAINTS>
16.  <MEMORY_CONSTRAINTS>
17.   <MAX_SIZE> 6MB </MAX_SIZE>
18.  </MEMORY_CONSTRAINTS>
19. </REQUEST>
```

Fig. 5. XML request after aspect's modifications

As it can be seen, some aspects might perform opposite modifications on the request. This kind of problem could arise in those base applications actions that are adapted by several aspects. For those cases where adaptation is done through requests modifications, our approach is to let all the modifications to be made, and solve the conflicts at the server side. That is to say, the server should overcome conflicting requests by stating some priority order among them. Then, there should be some strategy (at server side) that decides which modifications are the most important ones. For those base application actions that should be adapted and where several aspects are working on, it is possible to define a precedence order for the aspects (AspectJ supports such feature).

8 Conclusions

In this work we have analyzed how application behaviour can be affected and adapted by the runtime context in ubiquitous software mobile devices. Such an adaptation is necessary to optimize the use of the scarce device resources. This optimization concern

comes at a price: it can make application's development more complex. We have also addressed this problem, by providing a transparent way to modularize and decouple these optimization issues from the main application. We propose a possible decomposition of an ubiquitous system into aspects, and we analyze the consequences of the AO design.

We think that ubiquitous applications present high complexity which can be successfully targeted by the *aspect-oriented* paradigm. To conclude, we claim that aspect orientation is a fundamental tool that should be fully exploited to modularize intrinsic concerns in ubiquitous systems.

Acknowledgments

The authors thank Dr. Gustavo Rossi for his useful comments, and LIFIA for its support.

References

1. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold: Aspect Oriented Programming: Introduction. Communications of the ACM, Vol. 44. (2001) 29–32
2. D. Salber, A.K. Dey, G.D. Abowd: Ubiquitous Computing: Defining an HCI Research Agenda for an Emerging Interaction Paradigm: Tech. Report GIT-GVU-98-01. IFIP Working Conference on Engineering for Human-Computer Interaction. Georgia Tech. (1998)
3. Shang-Web Cheng, David Garlan, Bradley Schmerl, Joao Sousa, Bridget Spitznagel, Peter Steenkiste and Ningning Hu: Software Architecture-based Adaptation for Pervasive Systems. Lecture Notes in Computer Science Vol. 2299. Springer-Verlag (2002) 67–82
4. Gerti Kappell, Birgit Prll, E Kimmerstorfer, Wieland Schwinger and T.H. Hofer: Towards a Generic Customisation Model for Ubiquitous Web Applications. 2nd International Workshop on Web Oriented Software Technology in conjunction with the 16th European conference on Object-Oriented Programming ECOOP. (2002)
5. Gerti Kappell, Birgit Prll, Werner Retschitzegger and Wieland Schwinger: Customisation for Ubiquitous Web Applications. Int. Journal of Web Engineering and Technology (IJWET), Inaugural Volume, Inderscience, Volume 1, No. 1, (2003) 79–111
6. Stephan Herrmann and Mira Mezini: PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments, ACM OOPSLA 2000 Proceedings. Vol. 26, Issue 1. ACM Press New York (2001) 188–207
7. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin:Aspect-Oriented Programming. 11th Europeen Conf. Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241. Springer-Verlag (1997) 220–242
8. Stephan Herrmann and Mira Mezini: On the Need for a Unified MDSOC Model: Experiences from Constructing a Modular Software Engineering Environment. OOPSLA 2000 Proceedings. ACM Press New York (2000)
9. R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier and L. Martelli: A UML Notation for Aspect-Oriented Software Design. Proceedings of the 1st international conference on Aspect-oriented software development. ACM Press New York (2002) 106–112
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold:, An overview of AspectJ.Proceedings ECOOP 2001. Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag (2001) 327–353

11. A. I. Periquet and E. Lin, "Mobility Reflection: Exploiting Mobility-Awareness in Applications by Reflecting on Distributed Object Collaborations," Technical Report 97-CSE-6, Southern Methodist University, 1997.
12. Lonsdale, P., Baber, C., Sharples, M. and Arvanitis, T. (2003) A context awareness architecture for facilitating mobile learning. In Proceedings of MLEARN 2003, London: LSDA.