Engineering Pervasive Services for Legacy Software

Silvia Gordillo, Gustavo Rossi, Andrés Fortier

Abstract. **In this paper we present a novel architectural approach to engineer applications that provide location-aware services; in particular, we explain how to extend existing software systems with location-aware services. We show how a clear separation of design concerns (e.g. applicative, context-specific, etc) helps to improve modularity. We stress that, by using dependency mechanisms among outstanding components, we can get rid of explicit rule-based expressions thus simplifying evolution and maintenance. We first motivate our research with a simple example. Next, we present the big picture of our architectural approach. Then we detail how to specify location-aware services; we present details of the services' activation mechanisms. We finally we discuss some related work in the field. We conclude with some further issues in which we are now working.**

1 INTRODUCTION AND MOTIVATION

Building applications that provide context-aware services has proved to be difficult; the most important reasons have been extensively reported elsewhere [1, 4]. The only way to guarantee seamless software evolution is to rely on solid software engineering practices; in particular, in order to assure software modularity, a clear separation of design concerns is a must [3]. To make matters worse, applications that provide context-aware pervasive services are not built from scratch; they emerge as a consequence of the evolution of existing software systems, which are modified or extended to provide brand new functionality, according to new communication and hardware possibilities. We have devised a design approach and an implementation framework to engineering location-aware services [11]. Instead of using rule-based approaches, we based our approach on an extensive use of well-known dependency mechanisms in the object-oriented field [8]. In this paper, we elaborate our approach and show how to use it to improve existing software systems with location-aware services.

As a motivating example, suppose an academic system which provides information (e.g. using a Web interface) on courses offered by the university, time-tables, teaching material, etc. For the sake of simplicity suppose that the system has been designed using good object-oriented practices, e.g. it follows the model-view-controller metaphor, in which model, interface and interaction issues are clearly separated and we can identify a set of model's behaviors that provide the intended information [8].

This work was supported in part by SeCyT under Grant PICT 13623.

Silvia Gordillo, Gustavo Rossi and Andrés Fortier are at LIFIA, Facultad de Informatica, Universidad Nacional de La Plata phone: 542214236585; fax: 542214228252; e-mail: {gordillo, gustavo, andres}@ lifia.info.unlp.edu.ar.
Silvia Gordillo is also at CICPBA, Gustavo Rossi is also at CONICET, Andrés Fortier is also at DSIC, Universidad Politécnica de Valencia, Spain.

How do we extend this system in order to provide pervasive services? For example, when a student is in a room in which a course is to be given, he has access to the material of the course; the professor meanwhile can access the list of students in the course and can upload material (e.g. for homework).

In the rest of the paper, we show in a step by step way how to seamlessly enrich existing applications with location specific information (e.g. locations of rooms), and how to engineer pervasive services. We treat services as light-weighted objects that are attached to physical locations (service areas) and made available to the user when he enters into the corresponding location; additionally, these locations may eventually refer to specific application objects (e.g. a room).

The rest of the paper is organized as follows: In Section 2 we present an outline of our architectural approach. In Section 3, we discuss the specification of location-aware services; details on service activation are presented. In Section 4 we analyze some related work and in Section 5 and present our concluding remarks.

2 HIGH-LEVEL ARCHITECTURAL DECISIONS

For the sake of comprehension, we will describe each design problem using a pattern format [3, 5], i.e. we briefly state the problem we faced, the context and the solution. This style, which follows the ideas in [2], allows us to show that the proposed solutions are more general than a particular framework implementation (like ours), and thus can be used in similar situations. We use a coarse grain for describing these architectural design decisions. Many of them deserve a longer explanation at a lower (say, micro-architectural) level, but we omit this discussion for conciseness. We will first concentrate on the most outstanding components and design decisions; details on architectural and lower-level issues related with hardware abstractions, sensing concerns and location models can be read in [11], as we only describe them in a high level way.

2.1 DEALING WITH LOCATION AND LOCATION MODELS

Problem: In many applications (such as in the example), we need to determine the position of the user in relationship with physical regions that correspond to application objects. For example, to decide whether a user is in a room or in its vicinity, we need to know the room's position in some location model [9, 10]. Coupling application objects with their positions or making them location-aware (e.g. adding a variable in class *Room* for representing rooms' positions) have many disadvantages. The most relevant problem is that it pollutes the application model with location information (which tends to be volatile), and with objects that are not important for the original application, e.g. a corridor in which we want to provide certain services. When we are extending legacy software, this problem is even more evident. In summary, how do we enrich applications with location information transparently?

Solution: Build a separated Location layer which contains the abstractions which are necessary to maintain and process location information. Objects in this layer may or may not have a counterpart in the application layer. For example, in this layer we model the located counterpart of application objects, e.g. *location.room*; we also model other objects which have not been defined in the application, but have a spatial meaning, such as *corridor* or *building* and which may be related by spatial relationships (corridor *connects* rooms). The Location layer also comprises lower-level abstractions that implement different location models (symbolic, semantic, geometric, etc) and a component, *Location.User* which contains the actual user position (See 2.2). The relationships between the Location and the Application Layer is shown in Figure 1. Notice that the relationship between objects in the Location and the Application layers resembles the Decorator [5] pattern as located objects "extend" application objects functionality with spatial information and behaviors. This solution makes application objects oblivious of their spatial extensions, therefore allowing their evolution, and the evolution of the located components independent and thus less error prone.
There are three important abstract classes in the framework: *LocatedModel, LocatedObject and Location*. Every class that stands for an application counterpart (as in the case of *Location.Room*) is defined as a sub-class of *LocatedModel*. Purely spatial classes (like *Corridor*) are derived from *LocatedObject*. The position of all objects in the Location layer are described by an object of a class implementing the *Location* type. This type abstracts different location models (geometric, symbolic, etc) which we do not describe in Figure 1. Decoupling located objects from the location model allows us to reason in a higher level way, as the existence of a corridor or a room and their spatial relationships are independent of the way they are represented as locations (e.g. with coordinates, symbols, code bars, etc).

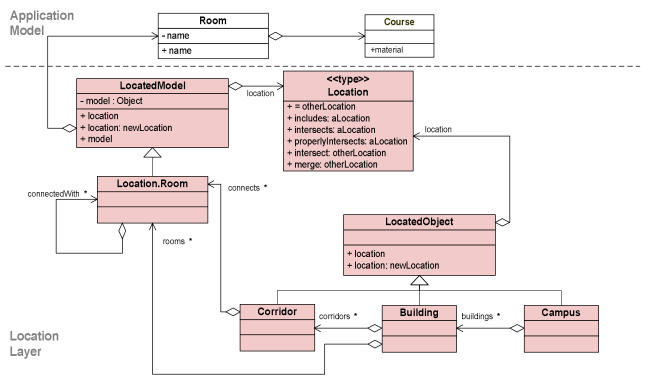

Figure 1: The Location Layer as a decorator on the application model

2.2 MODELING THE USER AND HIS LOCATION

Problem: Location-Aware Services react according to the user's position, so it is evident that we need to record this position. The usual solution is to build an object which comprises the user's actual contextual data, and to query this object to know the actual user's position. There are two problems with this approach: First, considering this object as just a data repository tend to delegate some of its responsibilities to other objects (thus, compromising modularity). Besides, in many applications such as in the university campus, there may be already an object which represents a possible user, e.g. a student object in the location model. Which is the relationship between these objects?

Solution: We model the user's location in the Location layer (*Location.User*). Similarly to other objects in this layer, this location is described using an object of type Location. *Location.User* may be also related with the corresponding application object (if any). We consider *Location.User* as a critical object in the process of triggering the activation of services instead of a passive data repository. Each time this object changes (i.e. the user changes his location), it communicates the change to its counterpart in the Service Layer (See Section 3), thus changing the possible available services. Figure 2 shows *Location.User* and its two sub-classes: *Application.User* and *Located.User* which play the same role as *Located.Model* and *Located.Object* in Figure 1. When the user has a counterpart in the application model (e.g. he is a registered student or teacher), we use *Application.User* which extends the corresponding class (in this example Person). A casual user is represented by *Located.User* objects. As we will concentrate ourselves on the process of activating services, we will not address authorization issues in this paper. The description in Section 3.4 holds for both Application and Located users.

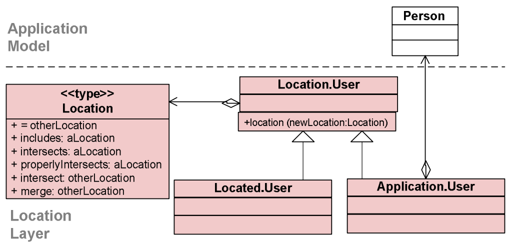

Figure 2: Locating the user

2.3 DEALING WITH HARDWARE AND SENSING

Problem: Hardware for sensing the user's location (and other context variables) evolves constantly. Sensing policies (e.g. push vs. pull) vary according to hardware capabilities. It is clear that low-level details have to be hidden from the application. However, sensed data is in the best case string data and it has to be interpreted to fit into the needs of the application. Moreover, location models (See 2.1) should evolve independently of sensing hardware. How do we provide this independence?

Solution: Decoupling sensors and their logic, from application concerns has been the driver of many research projects. While context widgets in [4] and adaptors in [6] isolate hardware from the application software, we still need a higher level of interpretation to relate sensed data, first with location objects and then with application objects. We thus decided to further decouple the hardware abstractions (similar to Dey's widgets [4]), and the logic for sensing the user's position. These two layers, namely Hardware Abstractions, and Sensing Concerns and their relationships with the previously described components are shown in Figure 3. In Figure 4 we present a more detailed diagram, exemplifying with a simple location sensing widget, an IR port. The Sensing concern acts as a dependency transformer between the hardware and the location layer. The dependency relationship implies that every time something changes in the sensor, the sensing concern object is notified. This object abstracts the sensing policy (e.g. push or pull), as a Strategy object [5], represented by a sub-class of *Sensing Policy*. Once it has the new position, it maps the sensed value into an object of the Location layer (e.g. a room, a corridor, etc). There might be different algorithms for performing this mapping, which obviously depend on both the sensed data and the actual location model (e.g. symbolic or geometric).
This new location is sent to the *Location.user* object, indicating that the user has changed his position. Notice that the sensing concern layer plays the role of an interpreter, enhancing the behavior of Dey´s interpreters [4] to get a slightly higher level location object, which can be related with an application object.

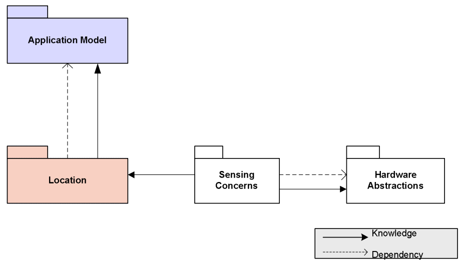

Figure 3: Sensing Concerns and Hardware Abstractions

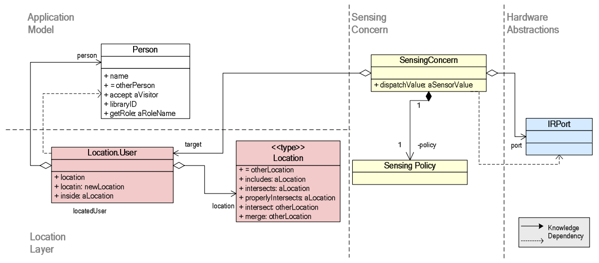

Figure 4: Refinement of Sensing Concern and Hardware Abstractions

3 ENGINEERING LOCATION-AWARE SERVICES

We consider services as full-fledged software artifacts which may be (as in the examples in this paper), extensions of an application's behavior. They may already exist as methods in some application object, e.g. providing the material of a course or they may not use application's methods, but involve application-related objects, e.g. informing which rooms are in this floor.
Some services will be autonomous; others might require the use of "external" (eventually Web) services, e.g. returning the actual temperature in the area. As others, we envision that service engineering will be a critical software design activity and, therefore, design issues related with location-aware services are fundamental to assure the quality (e.g. modularity, reuse, etc) of the engineered services. Following the style of Section 2, we next analyze the most important problems we faced and the solutions we propose and implemented.

3.1 DESCRIBING SERVICES AS MODULAR APPLICATION'S EXTENSION

Problem: Services should be engineered independently of other application components. It should be possible to make them dynamically available to users, according to different conditions (location, role of user, etc). It should be possible to specialize them or compose them to obtain more complex services. However they may have a close relationship with applications' behaviors, i.e. activating a service might imply the invocation of an application object's method. How do we balance modularity with the need to relate services with application objects?

Solution: First, we created a Service Layer separated from the others framework layers. Then we defined Services as objects, following the Command [5] design pattern. An abstract class Service is defined to contain the interface common to all services. For each possible service, we defined a class, representing the concrete service, e.g. *GetMaterial*. When a service is activated (e.g. because a user enters in a place where the service is available), we instantiate the corresponding class and allocate this object to serve the user. Allocating a service to the user means to initialize an instance variable of the service (defined in the abstract class) to refer to the user (See 3.2) and add the service object to the set of active services for that user (See 3.2). Other relevant information is defined when the service is started (See 3.3 and 3.5).
Treating services as first-class citizens allows manipulating them uniformly, building service compositions, making their activation dependant of the behavior of other objects, etc. We also decouple the instantiation of a service from its execution, which allows us to manage services' queues, logs, etc. The relationship between Services and application objects is a knowledge relationship as shown in Figure 5. We will discuss the relationships among the Services and the Location layer in Section 3.2.
In Figure 6 we show the abstract class *Service* and its two subclasses: *User Service* and *Internal Service*. An internal service is a service which is used either by other services or by the framework itself, while user services are provided to the user. The concrete class *GetMaterial* has an instance variable *course*, which will be set, when the service is activated, to refer to the object representing the course given in the actual room (where the user is located) as explained in Section 3.2. In the same way we can define new services, which might depend on the user's location or not.
From the point of view of service configuration (e.g. attaching services to locations and allocating them to users), our frame-

work can be considered a black-box framework [2], as the configuration process (as discussed in the following sections) can be done using objects composition. Meanwhile, the specification of new services is understood as a framework extension, and as the developer needs to know which methods he has to provide (e.g. to specialize definitions in the abstract class Service), the framework can be also considered a gray-box framework [2], because extensions use basically sub-classing mechanisms.

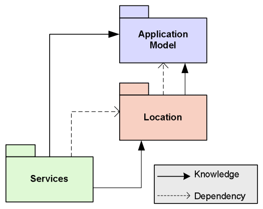

Figure 5: The Service Layer and the relationships with other layers

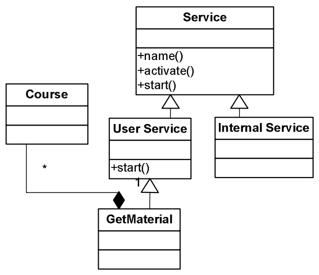

Figure 6: Services as First-class objects

3.2 LINKING SERVICES TO LOCATIONS AND PROVIDING LOCATION-AWARENESS

Problem: How do we state that a certain service should be active (available to the user) in a certain location? How do we proceed to its activation? The usual solution is to write a set of rules in which the condition checks whether the user is inside an area and the action consists in activating the service. While rules can be decoupled from the application (e.g. similar to business rules) and designed as objects (See for example the material in [11]), they tend to become monolithic when the application evolves; for example when new services emerge, or a service needs to be attached/detached to a different area we have to edit (add or modify) rules. Besides, the information on which services are provided in a location, and the code for invoking those services, are tangled in the corresponding rules; this makes the maintenance activity more difficult as we need to read all rules to grasp the big picture.

Solution: Register services to locations using a dependency mechanism such that when a user enters into a place, all services registered to that place are notified, and thus they are made available to the user. This solution involves the specification of several components to guarantee flexibility (of service definition), and modularity (of the underlying software). The most relevant are the following:

Service.User: This object has two fundamental roles; one which is purely informational: it knows the actual available services for the user; the other role relates with the process of updating these services: it is dependent (a kind of Observer [5]) on *Location.User*; therefore every time the user changes his position (and therefore *Location.User* changes), *Service.User* is notified and it triggers a set of behaviors to determine which services are not longer available and which ones should be added. *Service.User* implements the most important dependency mechanism for implementing location-awareness. In Figure 7 we show these relationships.

Service Area: Services are not always provided in logical areas, such as a room, a bar, or a corridor, nor should they depend on sensing hardware (e.g. if a room has many sensors inside it); instead they may be defined opportunistically in aggregations of areas or part of areas; we call these aggregation Service Areas. For example, we might want that the services corresponding to a room are provided also in the surrounding of the room, e.g. in a part of a corridor (assuming that the user can be sensed to be there). A Service Area has a knowledge relationship with the corresponding Location object which defines it (in the Location layer), and with the Services which are provided in the Area. The *Service.User* object also has a knowledge relationship with the area in which he is located (See Figure 7).

Service Environment: The Service Environment acts as a Mediator [5] between the *Service.User*, and the service areas (instances of *Service Area*). When the user changes his position, *Service.User* collaborates with the environment to determine if the user has left or entered a new area. The *Service Environment* then sends the message *leaveArea* (or *enterArea*) to *Service.User* which will update the current services accordingly.

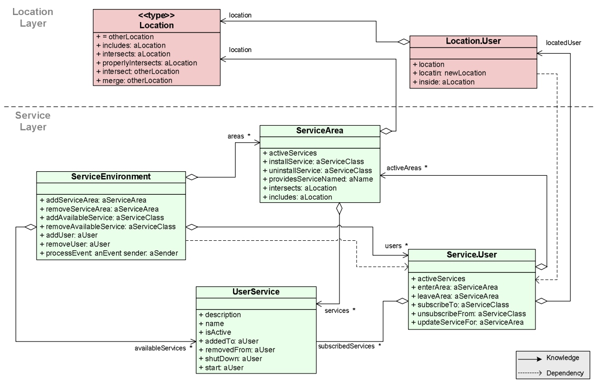

Figure 7: Components of the Service Layer

Figure 7 shows a static diagram with the relationships among these classes and also showing how these classes interact with classes in the Location Layer. Figure 8 shows a simplified sequence diagram with the process of activating a set of new services for a given user, i.e. making these services available to the user. When the *Location.User* object receives the message indicating a change of position, it notifies *Service.user* by means of the dependency mechanism. *Service.user* gets the new position and interacts with *Service.Environment* to analyze if the new position implies that the user entered or left a service area. *Service.Environment* interacts with *Service.Area* (not shown in the sequence diagram) and sends either the *message enterArea* or *leaveArea* to *Service.user*. The effect of executing these methods is that a new set of services is allocated to the user: those corresponding to the actual service area in which the user is located.

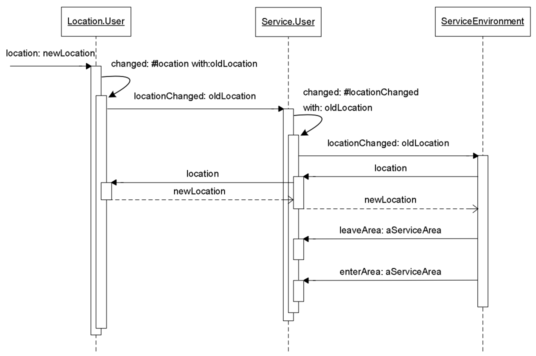

Figure 8: Activating a Location-Aware Service

3.3 RELATING SERVICES WITH APPLICATION OBJECTS

Problem: As previously discussed, there are services which may need a strong interaction with application objects; for example suppose that we specify a service which may be provided in rooms, and which returns the material of the course that is currently scheduled in the room. We may also associate a service with research laboratories to provide information on the corresponding research projects. In both cases, the service object needs to interact with a specific application object (a room, a research lab), which will eventually mediate with other objects (course, project, etc). However, services which only deal with application data should not be cluttered with details on location issues; they should only concentrate on their task. How do we instantiate services to realize the correct relationship?

Solution: We provide a pre-built initialization method in the abstract class Service; when a service is started, executing a method in Service, it must first initialize itself (this process is performed using a template method [5]). The standard initialization method (*init*) returns the application object corresponding to the actual user's location (e.g. a room object), which is sent as a parameter for service execution. This object is obtained by collaborating with the actual user's location object, which may have an explicit relationship with an application object. For pure spatial objects, i.e. those which do not correspond to any application object a *nil* object is returned. Designers can either re-define *init*, by including it in the concrete class (therefore being invoked by the template method) or perform other additional initializations, e.g. when more application or spatial information is needed for the service execution. For example the service can get the course which is scheduled and set the relationship shown in Figure 6.

3.4 MANAGING SERVICE GRANULARITY

Problem: In most applications, services are associated with coarse-grained physical objects, e.g. meaningful physical areas in the university. Moreover, once the user is sensed to be in a Service area, we can assume that the services allocated to the area are bound to the corresponding application object as discussed in Section 3.3. For example when the user enters in a room, the service *GetCourseInformation* will refer to the course object being scheduled in that room. Suppose, however that we want to provide services with a finer grained physical scope. For example we might have an art exhibition in the room and we want to provide additional information on artworks, as in augmented reality applications, e.g. when the user stands in front of an artwork he can get information on the painter or technical data (such as material, painting technique, etc). Assuming that we can sense the user's position precisely, should we define finer grained service areas and allocate the same service (type) to each of these areas?

Solution: In our conceptual schema, services are allocated to areas. In the example above, it is clear that the service area is the room and that defining new areas for each of the physical objects (artworks), poses a problem of maintenance; adding a new artwork requires the definition of a new service area. We instead need to adapt the behavior of the location-aware services to the specific object the user is facing. The solution emerges by analyzing the flow of control that results when the user is sensed to be in a new location (in front of an artwork). As explained in Section 3.2, the Service environment detects that the user did not exit a service area and he did not enter another one (this conclusion follows by analyzing the physical object in the corresponding Location model). Therefore, the previously allocated services still apply. However, the physical position of the user has changed, and this change has been registered in *Location.User*. Then, when the service *GetArtworkInformation* is started, the process described in Section 3.3 is performed and the service is bound to the correct physical object (the artwork). In other words, we can make (location-aware) services adaptable to finer grained physical objects, without changing the overall architecture, nor the underlying location model: we just need to slightly re-write the *start* behavior of those services.

4 RELATED WORK

The Context Toolkit [4] has been our first source of inspiration for providing a clear separation of concerns in our architecture. Our hardware abstractions and sensing concerns are similar to Dey's [4] context widgets. Hydrogen [6], meanwhile, introduces some improvements to the capture, interpretation and delivery of context information with respect to the seminal work of the Context Toolkit. Both the Context Toolkit and Hydrogen are aimed at providing a reusable context infrastructure that can be used by several applications. In this sense, the application concern is not dealt with and therefore there are no cues about how to structure application objects, particularly when they involve some information which is important for deciding about context changes.

Our view is slightly different; we focus on how to seamlessly extend existing applications with location-aware services. Even though our architecture also provides reusable components, our main goal resides in how to bridge context information with application objects. Our approach proposes a clear separation of concerns between those object features that are "context-free" (attributes and behaviors), those that involve information that is context-sensitive (like location and time) and the context-aware services. In this sense our approach proposes a set of micro-architectural styles to add location and services to application objects, which inverts the usual relationship between these aspects. While naïve software approaches make objects aware of their positions and services, we use decorators and commands [5] to achieve the same result but making the application oblivious of these additions. This approach also improves traditional rule-based approaches like [12], which tend to hardcode service activation conditions in rule conditions; these conditions either refer explicitly to application objects (which as a consequence must know their location), or contain location information, thus making them a critical point during maintenance.

By clearly decoupling these aspects in separated layers, we obtain modular applications in which modifications in one layer barely impact in others. Our idea of connecting services to places has been used in [7], though our use of dependency mechanisms improves evolution and modularity following the Observer's style [5,8]. From an architectural point of view, our work has been inspired in [2]: the sum of our micro-architectural decisions, such as using dependencies or decorations also generate a strong, evolvable architecture.

5 CONCLUDING REMARKS AND FURTHER WORK

In this paper, we have described a set of abstract architectural components and their associated communication mechanisms, which provide a substrate for seamlessly extending existing object-oriented software to support location-aware services. The most important goal of our approach is that it provides an original way of mapping application objects to their located counterparts (i.e. the objects which describe their positions in a particular reference system). By using dependency mechanisms instead of rules we improve maintenance; the cost we pay is that the underlying design is more complex than typical rule-based systems which usually comprise a rule model, a context model and application objects. We are now researching on the following areas:

-Regarding service specification we are now studying how to use services as proxies of Web Services; while the use of objects to manipulate services is straightforward, many of our design structures relies on pure object-oriented constructs which have to be slightly modified to deal with XML-based services.

-We are studying abstraction and composition mechanisms at the service level, both to express service's behaviors and activation conditions. For example we may have an abstract service which is activated in every room (e.g. Get material) but which may be refined into more specific ones according to the room, or other conditions hold on application objects (e.g. the kind of course in the course or other constraints).

-We are building interactive tools to improve the specification of services and service areas to our framework.

-We are extending the approach to other kind of context data. Traditionally, context data has been treated as plain data which can be queried (e.g. activities are described as a string such as "working"); by objectifying such data, services dispatch can be dealt with by delegating the corresponding decisions to the involved object (e.g. an instance of a sub-class of Activity or Role). For other kinds of contextual information, e.g. measurable context data, we are extending the notion of service area to use the same kind of strategies which we use for spatial information. We are also studying how to deal with n-dimensional areas, where each dimension deals with a different kind of context data.

-We are improving the architecture by incorporating an event model to simplify the management of dependencies. Event models help objects which receive a notification to delegate to specific event managers; the impact of this approach is that we can dynamically add new kind of events (e.g. to manage a new kind of context information), without having to edit the working code.

REFERENCES

[1] G. Abowd. "Software Engineering Issues for Ubiquitous Computing". Proceedings of the International Conference on Software Engineering (ICSE 99), ACM Press, 1999, pp. 75-84.

[2] K. Beck, R. Johnson: "Patterns generate architecture". Proceedings of the European Conference on Object-Oriented Programming, Ecoop '94 Lecture Notes in Computer Science.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, M. Stal: "Pattern-Oriented Software Architecture". John Wiley, 1996.

[4] A. Dey: "Providing Architectural Support for Building Context-Aware Applications". PHD, Thesis, Georgia Institute of Technology, USA, 2001.

[5] R. Gamma, R. Helm, R. Johnson, and J. Vlissides:_"Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[6] T. Hofer, M. Pichler, G. Leonhartsberger, W. Schwinger, J. Altmann: "Context-Awareness on Mobile Devices - The Hydrogen Approach", Proceedings of the International Hawaiian Conference on System Science (HICSS-36), Minitrack on Mobile Distributed Information Systems, Waikoloa, Big Island, Hawaii, January 2003.

[7] T. Kanter, "Attaching Context-Aware Services to Moving Locations", IEEE Internet Computing V.7, N.2, pp 43-51, 2003.

[8] G. Krasner, S. Pope, "A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80", Journal of Object Oriented Programming, August/September, 1988, 26-49.

[9] U. Leonhardt, "Supporting Location-Awareness in Open Distributed Systems", Ph.D. Thesis, Dept. of Computing, Imperial College London, May 1998. http://www.doc.ic.ac.uk/~jnm/ul_thesis.pdf

[10] S. Pradhan, "Semantic Location." Personal and Ubiquitous Computing 4(4): 213-216 (2000).

[11] G. Rossi, S. Gordillo, A. Fortier: "Seamless Engineering of Location-Aware Services", Proceedings of CAMS 2005, 2nd Workshop on Context-Aware and Mobile Services, Ciprus, October 2005, Springer Verlag.

[12] UWA Project. www.uwaproject.org