Seamless Engineering of Location-Aware Services[*]

Gustavo Rossi[1,2], Silvia Gordillo[1,3], Andrés Fortier[1]

[1] LIFIA, Facultad de Informática, UNLP.
La Plata, Argentina.
{gustavo,gordillo,andres}@lifia.info.unlp.edu.ar
[2] CONICET
[3] CICPBA

Abstract. In this paper we present a novel approach to design and implement applications that provide 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 mechanism among outstanding components we can get rid of explicit rule-based approach thus simplifying evolution and maintenance. We first discuss some related work in this field. Next, we introduce a simple exemplary scenario and present the big picture of our architectural approach. Then we detail the process of service definition and activation. A discussion on communication and composition mechanisms is next presented and we end presenting some concluding remarks and further work.

1 Introduction

Building applications that provide location-aware services (i.e. those services which depend on the user position) is usually hard. Moreover, maintenance and evolution of this kind of software is even harder. There are many reasons for this:

- Dealing with location (and other kind of context) information is hard because this information has to be acquired from non-traditional devices and must be abstracted to make sense to applications [4].
- Applications usually evolve "organically" [1]; new services are added, location models change, new physical areas are "tagged" to provide services.
- Application objects contain location (or contextual) information (e.g. a room is located in some place in a building) which is usually not clearly decoupled from other application-specific information (e.g. courses given in the room). As a consequence it is difficult to engineer these concerns separately.
- Adapting to context requires different engineering techniques many of which are usually misunderstood. For example, most approaches use the rule-based

[*] This paper has been partially supported by the Argentine Secretary of Science and Technology (SeCyT) under the project PICT 13623

paradigm to express adaptation policies, such as: “When being in a room, provide services A, B and C”. While rules can be often useful (especially when we want to give the user the control of building his own rules), many times rule sets might become too large and thus may be necessary to use more elaborated design structures to improve maintenance and evolution.

Our approach is based on a clear separation of concerns that allows us not only to decouple context sensing and acquisition (as in [12]), but mainly to improve separation of inner application modules, easing extension and maintenance. To achieve this, we make an extensive use of dependency (i.e. subscribe/notify) mechanisms to relate services with locations. In our approach, services are “attached” to locations in such a way that a change in the user's location triggers the necessary behavior to display the available services. Further use of the same philosophy may allow applying it to other contextual information, such as time and role.

The main contributions of our paper are the following:

- We show how to separate application concerns related with context awareness to improve modularity. Also, as a by-product, we indicate a strategy to extend legacy applications to provide location and other context-aware services. A concrete architecture supporting this approach is presented.
- We show how to objectify services and make them dependent of changes of context; in particular we emphasize how to provide location-aware services.

The rest of the paper is organized as follows: In Section 2 we briefly discuss related work. In Section 3, we introduce a simple motivating example both to present the problems and to use it throughout the paper; In Section 4 we describe the overall structure of our architecture. In Section 5 we focus on service specification and activation. We discuss some further issues in Section 6 and present our concluding remarks in Section 7.

2 Related Work

The Context Toolkit [4] is one of the first architectural approaches in which sensing, interpretation and use of context information is clearly decoupled by using a variant of the MVC architectural style [9]. Meanwhile, the Hydrogen approach [7] introduces some improvements to the capture, interpretation and delivery of context information with respect to the seminal work of the Context Toolkit. However, both in [4] and [7] there are no cues about how application objects should be structured to seamlessly interact with the sensing layers. As shown in Section 4, our approach proposes a clear separation of concerns between those object features (attributes and behaviors) that are “context-free”, those that involve information that is context-sensitive (like location and time) and the context-aware services. By clearly decoupling these aspects in separated layers, we obtain applications in which modifications in one layer barely impact in others. The idea of attaching services to places has been used in [8], though our use of dependency mechanisms improves evolution and modularity following the Observer's style [6].

In [5], Dourish proposes a phenomenological view of context. In this work, context is considered as an emergent of the relationships and interactions of the entities involved in a given situation. This idea improves existing approaches, in which context is viewed as a collection of data that can be specified at design time and whose structure is supposed to remain unaltered during the lifetime of the application. Similarly to [5], we don't treat context as plain data on which rules or functions act, but as the result of the interaction between objects, each one modeling a given context concern. In addition, we do not assume a fixed context shape, and even allow run-time changes on the context model.

3 An Exemplar Scenario

Future location-aware systems will not be built from scratch but rather as the evolution of existing legacy systems. Suppose for example, that we want to enhance a university information system with context-aware functionality to make it work as the example in [13]. Our system already provides information about careers, courses, professors, courses' material, time-tables, research projects, etc. and is used (e.g. with a Web interface) by professors, students and the administrative staff.

In our enhanced (location-aware) system, when a student arrives to the Campus he can invoke some general services; when he enters a room he is presented with those services that are meaningful for that room (while still having access to the former services); for example, he can get the corresponding course's material or obtain information about the course, its professor, etc. When he moves to the sport area, he can query about upcoming sport events and so forth. Other kinds of users (professors) will have access to a (partially) different set of services.

In this paper we show how we solved one of the most challenging design problems in the scenario: how to seamlessly extend our application in order to be location-aware, i.e. to provide services that correspond to the actual location context. For the sake of conciseness we barely mention other kinds of context-aware adaptation.

4 Our Architectural Approach. An Outline

To cope with the dynamic and evolving requirements of this kind of software we devised a layered architecture in which we combine typical inter-layer communication styles [3], with the dependency mechanisms of the Observer design pattern [6]. In Figure 2 we present a high level view of the most important modules of the architecture shown as UML packages; we next describe each of them in detail together with the main communication mechanisms, in the context of the example.

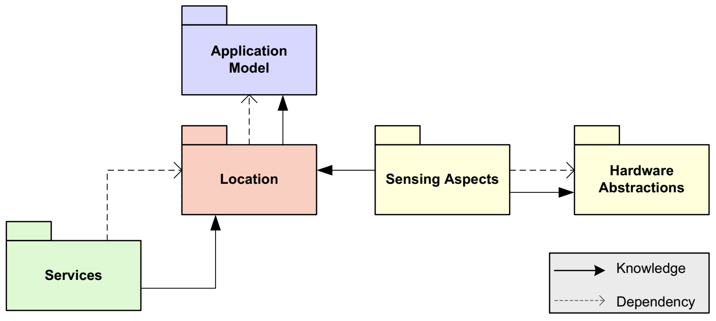

Fig. 1. The Architecture of a Location-Aware Application

4.1 Architectural Layers

In the **Application Model** we specify the application classes with their "standard" behaviors; application classes and methods are not aware of the user's context, neither they exhibit location (or other context-related) information. In our example we would have classes to handle room reservations, time-table scheduling, professors and material associated with each course, etc.

In the **Location Model**, we design components that "add" location properties to those application objects that must "react" when the user is in their vicinity. For example, to be able to say that a user is in room A we first need to create a location abstraction of the corresponding room object; we clearly separate the room from the object which considers the room as a spatial object. The location layer also comprises classes for "pure" location concepts, for example corridors, maps, connecting paths, etc., which do not have a counterpart in the application layer. In our example, we may be interested in representing a map of the university building, where we find rooms that are connected by corridors. The actual user location, which represents one of the user context concerns, is also modeled in this Layer. The design decision underlying the Location layer (separating an object from its location in such a way that the object knows nothing about its spatial properties) is one of the keys to support seamless extension. Decoupling location objects from other application objects allows us to deal with different location models such as symbolic or geometric [10] transparently. In the Location layer, we model location as an interface, so that many different location models can be used interchangeably. In Figure 3 we show an example of the Location layer including the location of some application objects and the user location. The Location layer can be generalized into a Context layer by including time, role, activity, and other contextual information that we do not describe in this paper for the sake of conciseness.

As discussed in the next paragraph, new instances of Location can be built "opportunistically" to create service areas, i.e. those areas to which a set of services apply (e.g. one might have services that are meaningful in an area covered

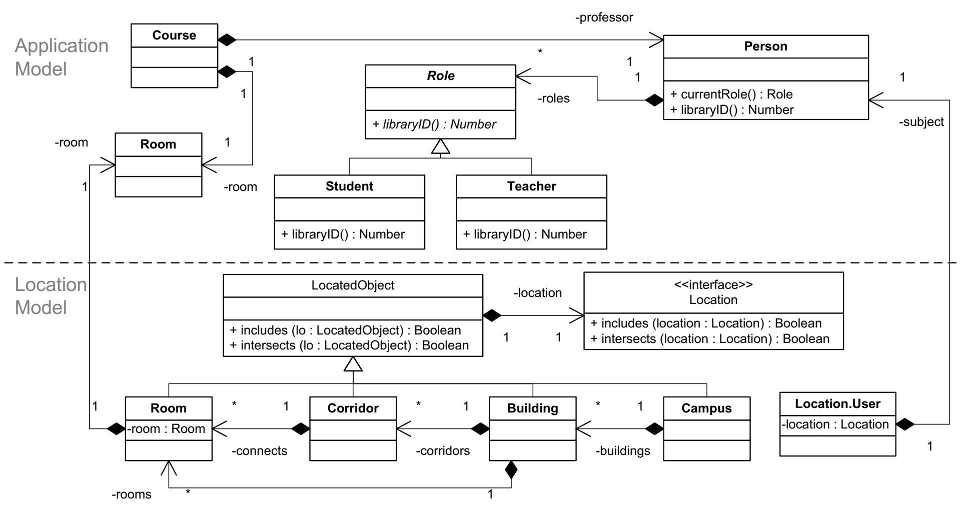

Fig. 2. Location Model vs. Application Model

by the union of a corridor and a set of rooms, even though we do not consider the area as a full-fledged LocationObject).

The **Service** layer contains the (location-aware) services. These services are modeled as objects that will be further associated to certain geographic areas (that belong to the location layer) by means of a subscription mechanism. In this layer we also include an object that manages the actual user services (in particular service subscription and activation) and coordinates other user's aspects. The User object (or Service.User) plays a similar role as the Context component in [4]. A high level description of the Service Layer and the relationships with objects in the Location model are shown in Figure 3. It is important to notice the way in which we express that a service is available in a certain area: since we don't want to clutter the location layer with services stuff, the logical relationship between services and physical areas is expressed by the concept of a service area. A service area class is used at the service level; an object of this class knows which physical area it covers, by means of the location relationship (shown in Figure 3); its main responsibility is to know which services are currently available (in the area) in such a way that, when a user is standing in a location included in the service area, the new available services are presented to the user (see section 5). A service area thus, acts as a Mediator [6] between a set of services and the area in which they are available.

In the **Hardware Abstractions** layer we find the components used for gathering data, such as IButton, InfraredPort, GPSSensor, and so on; these abstractions are similar to Dey's Widget components [4].

The **Sensing** layer comprises those higher level aspects that plug the lower level sensing mechanisms (in the hardware abstraction layer), with those aspects that are relevant to the application's context and that have to be sensed. This

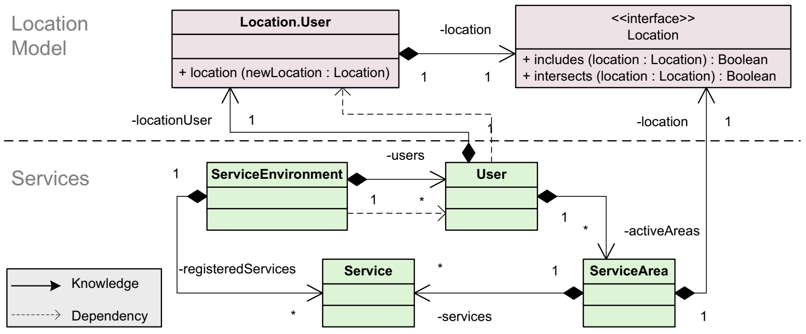

Fig. 3. The Service Layer

decoupling (which could be considered an improvement of Dey's interpreters [4]) guarantees that the location model and the sensing mechanisms can evolve independently. For example, we can use a symbolic location model [10] to describe locations, and infrared beacons as sensing hardware; we can later change to a non-contact iButton seamlessly (by hiding this evolution in the sensing layer). For the sake of conciseness we do not explain these (lower-level) layers in detail.

4.2 Communication Mechanisms

Relationships among objects in different layers follow two different styles: typical knowledge relationships (such as the relationship between the object containing a room's location and the room itself that resembles the Decorator pattern [6]) and dependency relationships (in the style of the Observer pattern [6]) that allow broadcasting changes of an object to its dependent objects.

A refinement of Figure 1 showing some of the outstanding dependency relationships is presented in Figure 4. When a change is detected in the Hardware Abstractions Layer (e.g. by an IR Port), a Location Aspect object is notified and it sends a message to the corresponding Location.User object (in the Location Model). The dependency mechanism between Location and Services allows notifying the User object which in turn notifies the Service Environment. This chain of events allows that the services corresponding to the actual user's position are made available to the User object which finally presents them to the actual user. In the next section, we elaborate our strategy to specify and manage services.

5 Service Specification and Activation

We consider (context-aware) services as possible independent artifacts which are developed individually and do not need to interact with each other (in fact they might even ignore that there are other services running). We also view them as extending some existing application behavior and thus they might need to interact with application objects. In this sense, even though being "isolated",

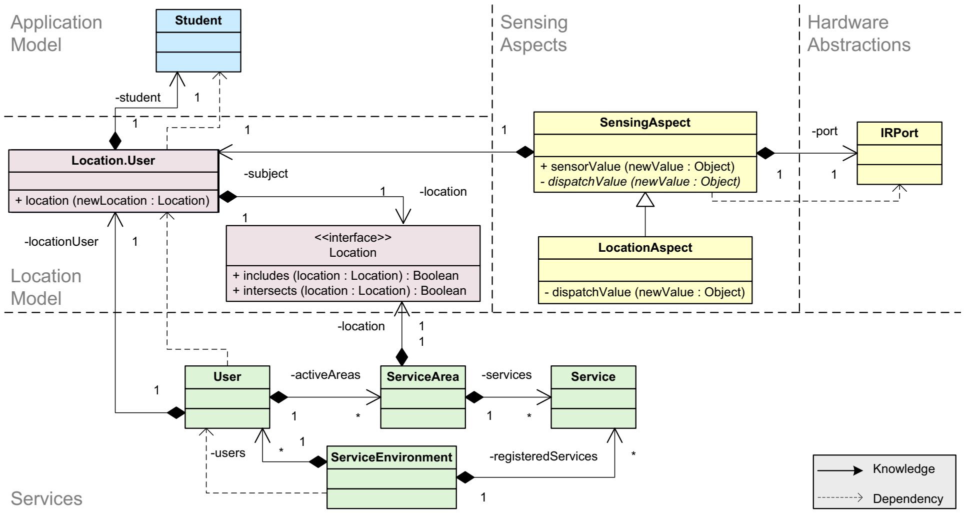

Fig. 4. Examples of knowledge and dependency mechanisms

they can be considered as parts of the application behavior (in fact a way to trigger application behavior).

Service users are immersed in a service environment (which in fact reifies the real-world environment). The Service.User object (i.e. the user considered from a services point of view) knows those services to which the corresponding user is subscribed to, and which services are currently available. The service environment is in turn responsible for handling available services, configuring service areas and mediating between users and services.

Services are modeled as first-class objects; this allows our framework to treat them uniformly and simplify the addition of new services.

5.1 Creating New Services

New services are defined as subclasses of the abstract class Service (Figure 4), and thus a specific instantiation of a service is an object that plays the role of a Command [6]. The specific service's behavior is defined by overriding appropriate methods of Service such as *start()* (used to perform initialization stuff), *activate()* (triggered when the users selects the service form the available services list), etc. For example, the CourseMaterial service is defined as a subclass of Service, and the message *activate()* is redefined so that a graphical interface is opened to display the course material.

5.2 Subscribing to Services

Users can access the set of available services (in an area) and decide to subscribe (or unsubscribe) to any of them. In our model, the service environment knows which services are registered, and therefore the users can query the environment for existing services in order to subscribe to them. The details of the subscription mechanism are beyond the scope of this paper; however, to mention some few interesting aspects in our framework, once a user is subscribed to a service, he can customize it (for example, the service can indicate its availability by playing a sound or by other means), or he can define additional context-aware constraints over the service (such as allowing or disallowing activation in certain situations).

5.3 Attaching Services to Spatial Areas

To provide location-awareness and to avoid the use of large rule sets, services are associated with (registered to) specific areas, called service areas. When the user enters into a service area, all services registered to the area (to which the user has subscribed) are made available. Service areas are defined to achieve independence from the sensing mechanism, i.e. they do not correspond to the scope of a sensing device (e.g. a beacon) but to logical areas. These logical areas can be specified programmatically or interactively; they can be obtained by applying set operators to existing areas (rooms, corridors, etc) or defined arbitrarily in terms of a location model.

As an example, suppose that we want to offer a location service in which a map appears showing where the user is standing. Clearly, we would expect this service to be available in the university building or even in the entire campus. If we are working with symbolic location we would probably have a 'Building' location that is the parent of the location tree that encompasses the rooms inside the building. So, in order to provide the location service for the building, we would create a new service area that is associated with the 'Building' location (see Figure 3); with this configuration, when the user enters the building (and as long as he is inside of it) he will have that service available. Now suppose that we would like to extend this service to the entire campus; using our approach we would just need to change the area covered by the service area, which in case of symbolic location means changing the location 'Building' to 'University Campus'. Similarly, if we want to add new services to that area, we do it by adding a service to the list of services known by the service area.

5.4 Service Activation

As explained abstractly in 4.2, when the user's movement is captured by a sensor it triggers a set of messages; concretely, it sends the *location (newLocation)* message to the Location.User corresponding to the actual user. This message triggers a change in the location model that is captured (by means of the dependency mechanism) by the User object in the service layer. This object interacts with its environment to calculate, if the user entered or left a service area, and the corresponding services are made available to the user, according to his subscriptions. As mentioned before, a service is presented to a user if it is available in a service area and if the user is subscribed to it. A subset of these interactions is shown in Figure 5 by means of a UML sequence diagram.

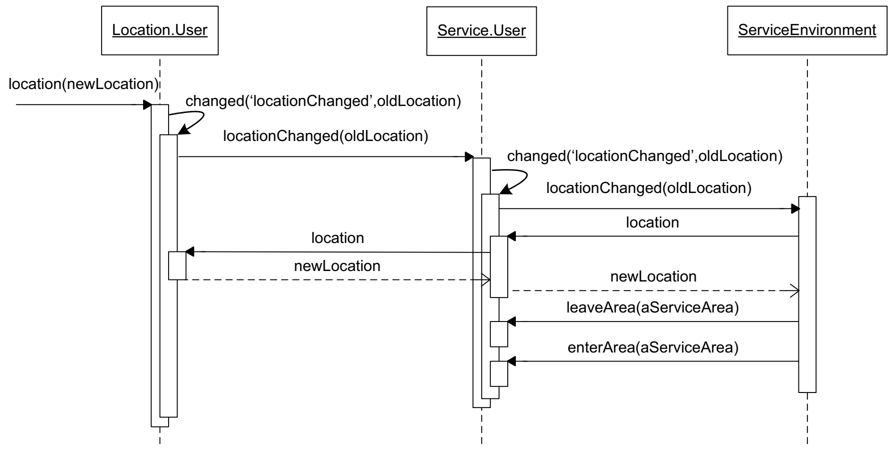

Fig. 5. Changing the Set of Active Services

6 Discussion

The impact of the previously described architectural style and communication mechanisms in the application's structure is somewhat evident. Separating hardware and sensing abstractions allows maintaining the benefits of well-known frameworks for context-awareness (such as [7, 12]). We introduced two additional layers (Location and Service) which refine the Application Model by clearly separating contextual attributes (such as position), low-level models (such as the actual location model) and context-aware behaviors, expressed as service objects.

Viewing Services as objects allows simplifying evolution; new services are added by just creating new classes, associating them to the service environment and to specific service areas, and publishing them to allow users' subscriptions. Finally the communication structure, expressed with a dependency mechanism that generalizes the Observer pattern, provides a seamless way to activate/deactivate services. Once associated with a spatial area, which can be done either programmatically (e.g. sending a message) or interactively, the Service Environment calculates which services can be accessed by a user by just reacting to a chain of notification events. In this way we get rid of large rule sets thus simplifying addition, deletion or upgrade of services (for example to associate them to different areas).

7 Concluding Remarks and Further Work

We have presented a novel approach for designing location aware services which uses dependency mechanisms to connect locations, services and application objects. We have also shown how we improved separation of different design concerns, such as applicative, spatial, sensing, etc. We have built a proof of concept of our architectural framework using a pure object oriented environment (VisualWorks Smalltalk). By using native reflection and dependency mechanisms we easily implement the architectural abstractions shown in the paper. We used HP iPaq 2210 PDAs as user devices; location sensing was performed using infrared beacons which can be adapted to provide ids or even URLs as their semantic locations [11]. Our approach represents a step forward with respect to existing approaches in which context information is treated as plain data that has to be queried to provide adaptive behaviors.

We are now working on the definition of a composite location system that allows symbolic and geometric location models to coexist seamlessly. We are also planning to enhance the simple dependency mechanism to a complete event-based approach, delegating specific behavior to events and improving at the same time the framework's reusability. We are additionally researching on interface aspects to improve presentation of large number of services and service maps.

References

1. Abowd, G.: Software Engineering Issues for Ubiquitous Computing. Proceedings of the International Conference on Software Engineering (ICSE 99), ACM Press, 1999, pp. 75-84.
2. Beck, K., Johnson, R.: Patterns generate architecture. Proceedings of the European Conference on Object-Oriented Programming, Ecoop '94 Lecture Notes in Computer Science.
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. John Wiley, 1996.
4. Dey, A.: Providing Architectural Support for Building Context-Aware Applications. PHD, Thesis, Georgia Institute of Technology, USA, 2001.
5. Dourish, P.: What we talk about when we talk about context. Personal and Ubiquitous Computing 8, 1 (2004) 19-30.
6. Gamma,R., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.
7. Hofer, T., Pichler, M., Leonhartsberger, G., Schwinger, W., Altmann, J.: 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.
8. Kanter, T.: Attaching Context-Aware Services to Moving Locations. IEEE Internet Computing V.7, N.2, pp 43-51, 2003.
9. Krasner, G., Pope, S.: A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object Oriented Programming, August/September, 1988, 26-49.
10. Leonhardt, U.: Supporting Location-Awareness in Open Distributed Systems. Ph.D. Thesis, Dept. of Computing, Imperial College London, May 1998.
11. Pradhan, S.: Semantic Location. Personal Technologies, vol 4(4), 2000, pp. 213-216.
12. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. Proceedings of ACM CHI 1999, pp 434-441.
13. Sousa, J.P., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. (3rd IEEE/IFIP Conference on Software Architecture) WICSA 2002: 29-43.