DECOUPLING DESIGN CONCERNS IN LOCATION-AWARE SERVICES

Andrés Fortier
LIFIA. Facultad de Informática. UNLP. La Plata, Argentina
andres@lifia.info.unlp.edu.ar

Gustavo Rossi
LIFIA. Facultad de Informática. UNLP. La Plata, Argentina
CONICET
gustavo@lifia.info.unlp.edu.ar

Silvia Gordillo
LIFIA. Facultad de Informática. UNLP. La Plata, Argentina
CICPBA
gordillo@lifia.info.unlp.edu.ar

Abstract In this paper we present an original approach to design and implement applications that provide location-aware services. Our approach emphasizes a clear separation of the relevant concerns in the application (base behavior, context-sensitive properties, services, etc.) to improve modularity and thus simplify evolution. We first motivate the problem with a simple scenario of a virtual campus; we next discuss which are the most important concerns in the application, we explain why we must separate them and show a simple approach to achieve this separation. We analyze the most important (sub) models in which we decompose a location-aware application and explain the use of dependency mechanisms to trigger behaviors related with the provision of services according to the user position. We briefly describe a proof of concept by means of an archetypical implementation we developed following our ideas. We next compare our work with others and discuss some further work we are pursuing.

Keywords: Location-aware services, location sensing, concern decoupling, modularity

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

1. Introduction

Context-Aware (and in particular Location-Aware) applications are hard to build and more difficult to maintain due to their "organic" nature (Abowd, 1999). For this reason, improving modularity is extremely necessary when designing this kind of software. Dealing with location (and other kind of context) information is essentially hard because this information has to be acquired from non-traditional devices and distributed sources, and it must be abstracted and interpreted to be used by applications (Dey, 2001).

While much research on context-awareness has focused on solving these problems, and many Context-Aware (CA) applications and frameworks have been built in the last years (Bardram, 2005; Hofer et al., 2003; Salber et al., 1999), there is still a poor characterization of those software design issues that make CA software difficult to build. In addition, CA applications have to deal with the following problems:

- Abstracting context means more than changing representation. Even though it is clearly explained in (Dey, 2001), the process of context interpretation usually ends far from application concerns. While interpreted context data is usually dealt as strings, applications are composed of objects, which means we have to deal with this impedance mismatch.
- Adapting to context is hard; design issues related with context-aware adaptation are not completely understood and thus handled incorrectly. For example, although rules can be useful (especially if we want to give the user the control of building his own commands), we claim that more elaborated structures are needed to improve maintenance and evolution.
- Context-related information is usually "tangled" with other application behavior. For example, the location of an application object (which is necessary to detect when the user is near the object) is coupled with others object's concerns, making evolution of both types of characteristics difficult.

Our research deals with the identification of recurrent problems and design micro-architectures in CA software. In (Rossi et al., 2005) we argued that design patterns are an excellent way to record and convey design experience related with CA (Abowd, 1999) adaptation. In this paper, we go further and describe an architectural approach for dealing with the problem of providing CA services (Bardram, 2005). Our approach is based on a clear separation of concerns that allows us not only to decouple context sensing and acquisition (as in (Salber et al., 1999)), but mainly to improve separation of application modules, to ease extension and maintenance. For this purpose we make an extensive use of dependency (i.e. subscribe/notify) mechanisms to provide context-aware services.

Along the paper we will show how to separate application concerns related with context awareness to improve modularity and, as a by-product, we will present a strategy to extend legacy applications to provide location and other context-aware services. In order to be consistent, we will treat services as full fledged objects and make them dependent of context changes.

The rest of the paper is organized as follows: In Section 2 we introduce a simple motivating example both to present the problems and to use it throughout the paper; in Section 3 we describe the most important concerns in this kind of software and introduce our criteria to decompose the application into layers and components. A complete description of each of the different models comprising our architecture is shown in Section 4. In Section 5 we briefly describe an archetypical implementation. In Section 6 we compare our work with related work in this field and finally, in Section 7, we conclude and discuss some further work.

2. Motivating Example

Suppose we are adapting an existing software system in a University Campus to provide context-based services (in particular, location-based ones), in the style of the example in (Sousa and Garlan, 2002). Our system already provides information about careers, courses, professors, courses material, timetables, etc. We now want that users carrying their preferred devices can get information or interact with the system while they move around the campus. For example, when a student enters a classroom, he can get the corresponding course's material, information about its professor, etc. At the same time, those services corresponding to the containing location context (the Campus) should be also available. When he moves to the sport area, the course's related services disappear and he receives information about upcoming sport events and so forth. It should be noticed that different contextual information such as the user's role or activity might also shape the software answer.

The first design problem we must face is how to seamlessly extend our application in order to be location-aware, i.e. to provide services that correspond to the actual location context. The next challenge involves adapting the behavior to the user's role (a professor, student, etc) and other meaningful contextual parameters such as current time or user's activity. While applications of this kind have always been built almost completely from scratch, we consider that this will not be the case if context-aware computing becomes mainstream; we will have to adapt dozens of legacy applications by adding new, context-aware behaviors.

When working with CA applications we can find typical evolution patterns such as adding new services related to a particular location, improving sensing mechanisms (for example moving from GPS to infrared), changing the loca-

tion model (from symbolic to geometric), and so on. While most technological requirements in this scenario can be easily fulfilled using state-of-the art hardware and communication devices, there are many design problems that need some further study. The aim of this paper is to focus on a small set of those problems, mainly those that characterize the difficulties for software evolution. We stress on those features specific to this particular example because it is a good stereotype of a family of software applications with similar problems.

3. Identifying and Separating Design Concerns

As previously mentioned, well-known approaches to context-aware applications design have clearly identified some broad concerns that must be separated for achieving modularity: sensing (implemented for example as Widgets in (Dey, 2001)), interpretation (also mentioned as context management in (Hofer et al., 2003)) and application. Layered architectural approaches such as in (Hofer et al., 2003), or MVC-based ones like (Salber et al., 1999) provide the basis for separating those concerns using simple and standard communication rules. However, applications (the third concern) are considered as being monolithic artifacts that deserve little or no attention. It is easy to see in the motivating example that the gap between application objects (in particular their behaviors) and the outer (context-related) components is not trivial. Of course, one could argue that once captured and interpreted, context information is not different to other "old-fashioned" application data, and thus we can use the very same techniques, which allowed us to survive in the past when dealing with input information. As a simple counter example let us take into account location data: to check that a user is in a campus' room, we must compare his position with the room location; is this location an attribute of the room object? What happens if we use different location models? Should we clutter the room object with these variants? Moreover, suppose that we are adding location-aware functionality to an existing system; should we change the base application behavior and write the code for providing location-awareness inside application objects? Following this thread, we may ask ourselves how to cope with services: are they supposed to be application behaviors (i.e. should we consider the services as methods of the room object?) or should they be decoupled into independent objects? The same problems also appears when dealing with other contextual information that cross-cut application objects.

In our research we have identified a set of concerns that should be clearly separated to improve evolution and maintenance: applicative, location and service concerns should be as independent as possible.

In the rest of the paper we will elaborate our strategy for building location-aware software and we will describe the previously mentioned concerns and how they interact with the lower-level ones (such as the sensing concern).

4. Designing Location-Aware Services

In the following sub-sections we assume that we need to extend an existing application with location-aware services. This application implements the base behaviors on top of which services are built. For the sake of understanding we first describe the overall architecture and concentrate later on each software component. The preceding example is used throughout the paper.

The Overall Architecture

To improve the description of the important architectural decisions, we present two orthogonal views showing different design concerns and how they relate with each other: an application-centered view and a sensing view.

Application view. This view (shown in Figure 1(a)) concentrates on the application model. In the first layer we specify the application model with its "standard" behaviors; application classes and methods are not aware of the user's context. In our example we would have classes to handle room reservations, professors and material associated with each course, etc. Note that in this layer the concept of a "user" does not exist, though we might have objects that correspond to different user roles, such as students, professors and so on.

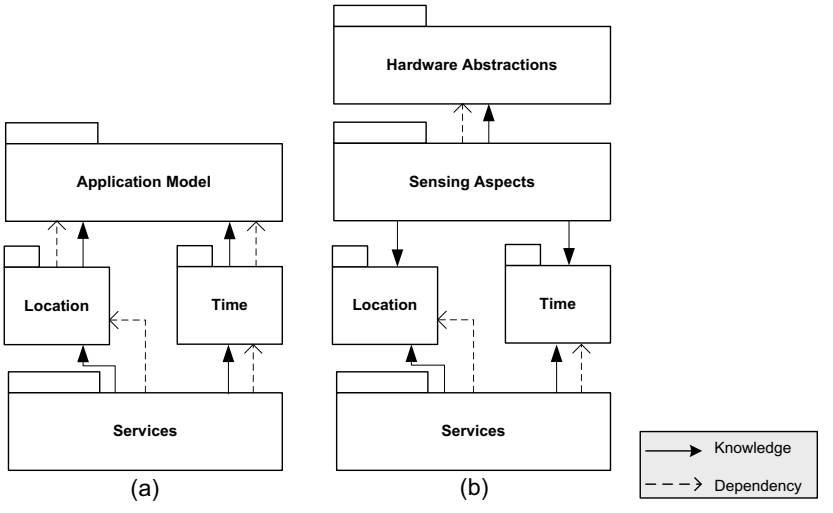

Figure 1. A layered architecture for Location-Aware Services

The second layer contains a set of components that extend the application model with information needed to provide context-aware behavior. For example, the campus, the sport field and the rooms have an associated location that is used to determine if the user is inside one of those areas. It is important

to notice that Location objects do not belong to the application concern; according to our approach the basic behavior of a room should not be cluttered with geographic information. As described in section 3, decoupling location from other application objects allows us to deal with different location models transparently.

Finally, the third layer contains the (location-aware) services. These services are modeled as objects that will be further associated to certain geographic areas by means of a subscription mechanism.

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) and dependency relationships (in the style of the Observer pattern (Gamma et al., 1995)) that allow broadcasting changes of an object to its dependent objects. In Figure 1(a) we also show an additional Package (Time) as an example of other context-related modules that may be included in the second layer.

In Figure 2 we show a small example exploiting the packages in Figure 1(a). Classes like `Course`, `Teacher` and `Room` belong to the application model and have no location-related behaviors. Location-aware classes (in the bottom of the diagram) "observe" application model classes and add additional context behavior. Notice that in this layer we introduce the notion of a `Location.User`, i.e. the location aspect of the user of our context-aware application. This aspect relates with the actual user's role through a `Person` instance.

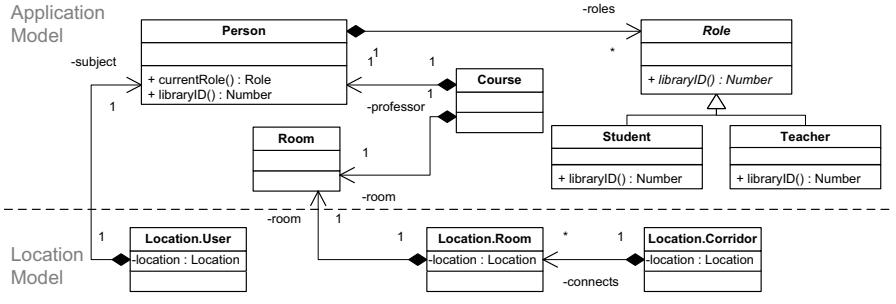

Figure 2. Location-Aware Classes vs. Application Classes

Sensing View. In Figure 1(b) we present another architectural view of our approach. In the first layer we find the hardware abstractions used for gathering data, such as IButton, InfraredPort, GPSSensor, and so on; this abstractions have some points in common with Dey's Widget components (Dey, 2001).

The second layer comprises higher level sensing aspects implemented as objects that plug the lower level sensing mechanisms (in the hardware abstraction layer) with the aspects that are relevant to the application's context that have to

be sensed. This decoupling guarantees that the location model and the sensing mechanisms can evolve independently. For example, we can use a symbolic location model (Leonhardt, 1998) 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.

Modeling and describing the user. As shown in Figure 1(a) we decided to model each context concern in a separate package. This idea is also applied to model the user: we consider the user model as being composed of different aspects, each one acting (differently) on the services that are available to the user. In our example, these services depend on the user's location and thus we need to model a user's aspect that handles the location concern. If, in the future, we decide that the way in which services are presented to a user may also depend on his preferences (explicitly stated by him or inferred from his usage history) we will need to add a new view which handles this aspect. Once the different concerns are modeled, we need some object to coordinate all views and decide how changes affect the user's services. We decided to design this coordinator in the service layer. This object, from now on called user object (or `Service.User`) knows, and it is dependent of, every concern that affects the user and reacts based on those concern's changes. The user model can be thought as cross-cutting the Services and Location layers; it comprises packages that belong to each of them.

Application Layer

In our architectural framework, the application model contains those classes specific to the intended domain and whose behavior do not depend on contextual information. In Figure 3 we show a simplified class model for the exemplary application. Notice that, for example, a room can return the courses occurring on that room at a particular time; a course meanwhile can provide its content material, the list of enrolled students, and so on. Also, the different roles modeled in the application might be eventually used for role-aware services. Notice also that there is no information on location, space, etc.

Location Layer

In the location layer we design components that seamlessly "add" location properties to those objects (in the application model) 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. By clearly decoupling the location from the application object we can use different location models (Leonhardt, 1998) in an unobtrusive way. In Figure 4 we show the class diagram of a simple location "map" of the campus. Note

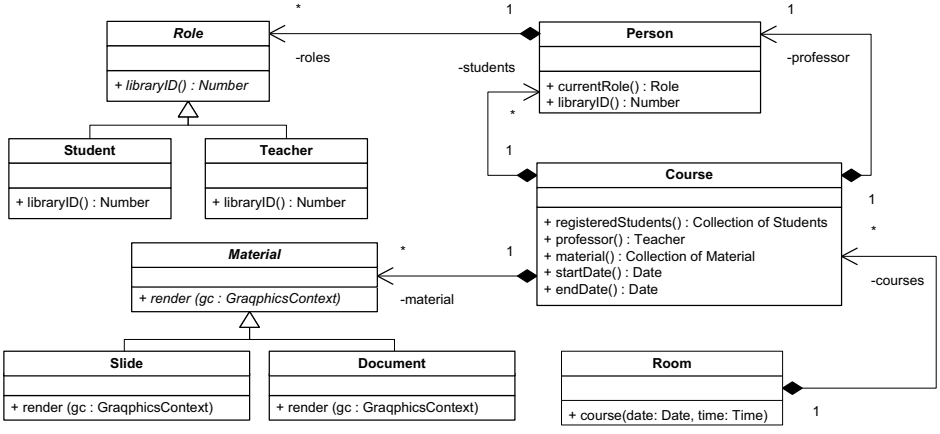

Figure 3. Class Diagram of the University Campus

that the location layer also comprises classes for "pure" location concepts; for example corridors and maps don't 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.

To achieve higher levels of reuse, we further decouple location objects from the specific location model we use for them. As a result of this separation, we end up with location objects (like rooms and corridors) that are aware of having a specific location, but that are independent of the location model being used. This independence is achieved trough the `Location` interface, which specifies the basic behavior that every location model should implement. Using this approach, implementation details (for example, knowing if a location is inside another) are hidden in each location model and allows us to change between location models dynamically without any impact on the system.

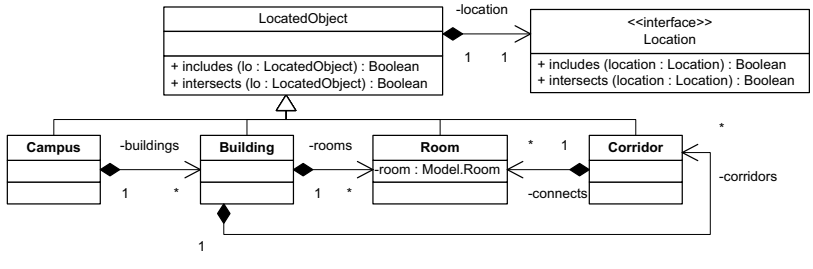

Figure 4. Class diagram of the University Campus Location package

Service Layer

We consider (context-aware) services as possible independent artifacts which are developed individually and do not need to interact with each other. We also view them as extending some existing application behavior and thus they might need to interact with application objects. Also, service users are immersed in a service environment, which reifies the real-world environment. A `Service.User` (i.e. the user considered from a services point of view) is modeled so that we can reflect those services to which a person is subscribed to, which services are currently available and so on. The `Service.User` object is also used to build the whole picture of a user, mediating between every possible context aspect that is relevant to that person. In this layer, the `Service.User` knows (and is dependent of) a `Location.User`, so that the service layer can react to changes in the location layer. In the remaining sections of this paper, each time we talk about a user we will be referring to a `Service.User`.

The service environment is in turn responsible for handling available services, configuring service areas and mediating between users and services. A service may be as simple as an alarm (that is triggered when we enter a place at a certain time) or as complex as a full-fledged application. Services are modeled as first-class objects which share a common super-class (or implement a given interface); this allows our framework to treat them uniformly and simplify the addition of new services. In the following sub-sections we give a brief outline of how services are modeled and implemented using this approach.

Creating New Services. New services are defined as subclasses of the abstract class `Service`, playing the role of a Command (Gamma et al., 1995). 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. In our example, the `CourseMaterial` service is defined as a sub-class, and the message *`activate()`* is redefined so that a graphical interface is opened to display the courses material. Once the service class has been created and its behavior defined, it has to be published to allow users to subscribe to the service; the *`addAvailableService`* message is used to inform the environment about the new service.

Subscribing to Services. Users can access the available services and decide to subscribe (or unsubscribe) to any of them. The details of the subscription mechanism are beyond the scope of this paper; however, is important to mention that a service can be customized by its user.

Once a user is subscribed to a service and provided he satisfies the service's constraints (for example in relationship to the user's role), he can use the service when entering the area associated with the service.

Service Areas. A key aspect in our approach is that services are associated with (registered to) specific areas, called service areas. When the user enters 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. To illustrate the idea, suppose that our sensing mechanism is based on infrared beacons. Since a beacon's signal range is limited, we may need to use more than one beacon to detect the presence of a person in a certain area. As an example, suppose that two beacons (B1 and B2) are placed in the opposite corners of a room (Room A) to detect the user presence. Even when there is a clear distinction between capturing B1's id and B2's id from the location-model point of view, this difference should be transparent to services allocated to the Room (area) A.

Services are not associated to physical areas (in terms of location models) but to logical areas named *service areas*. In this way, we can think of the services that are available in Room A or in the hall, instead of thinking about the services that are triggered by a group of beacons. In Figure 5 we show a class diagram indicating the relationships between the Environment, the Service Areas and the associated Services.

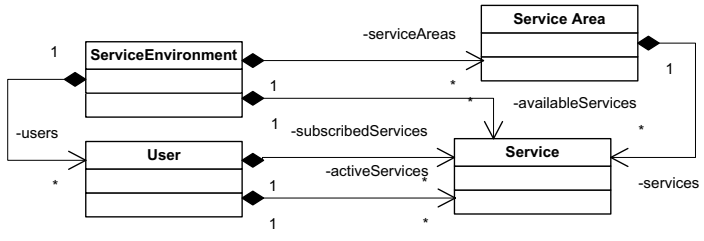

Figure 5. Services and Service Areas

Service Activation. When the person's movement is captured by a sensor, 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, based on the user's old location, if the user left a service area. If this is the case, the user object is told to leave that service area by means of the *leaveArea(aServiceArea)* message, which will remove the services provided by that service area from the user's active services. In a similar way, according to the new user's location, the environment checks if the user has entered a new service area. In that case, the user object receives the

enterArea(aServiceArea) message in order to add the corresponding services.

Sensing Concerns

We introduce the idea of a sensing concern to separate the context model from the way it is sensed. A sensing concern represents the "glue" between the different aspects that are relevant to our context-aware application and the way they are sensed. A sensing concern is created and configured to be an observer of one (or more than one) sensing mechanism. When a sensor indicates that an event occurred (i.e. some context information changed), the sensing concern acts on its subject by sending an appropriate message.

In this layer, the core behavior is modeled in the `SensingConcern` class and its subclasses. A sensing concern is attached to a sensor with a fetch policy suited for it; for example, a GPS system may need a pull policy while a barcode reader a push one. Additionally, we specify the message that should be sent to the object that models a specific context concern in order to update its aspect (in our example the *location(newLocation)* message should be sent to the `Location.User`). Depending on the programming environment used, this behavior can be achieved by sub-classing `SensingConcern` or via reflection.

Continuing with our example, when the student enters Room A the infrared port of his PDA captures B2's id, and the port abstraction (in the hardware layer) reacts by notifying its dependents that a new id has been received. Since a sensing concern has been created to modify the user's location, it receives the notification and reacts by adding the beacon's covering area to the user's active areas. Once this happens the corresponding user object interacts with its environment to find out which new services are active and available.

Putting all things together

In order to clarify the objects interactions occurring in our architecture, in Figure 6 we present an interaction diagram that shows how a change in the location layer triggers the service assignment to a user. From the services point of view, a change can drive the framework to add or remove service areas depending on the user's previous and current location. To keep the diagram simple, we assume that the initial interaction begins with a message sent by an object of the Sensing Aspects layer. When the sensing hardware (whatever it is) detects the presence of a user in a room, the sensing concern attached to it sends the message *location(newLocation)* to the user. The *newLocation* parameter is an object that implements the `Location` interface.

Once the `Location.User` receives the message it triggers a change. Since the user in the service layer is dependent of the `Location.User` it gets an update which, in turn, triggers a change that is captured by the `ServiceEnvironment`.

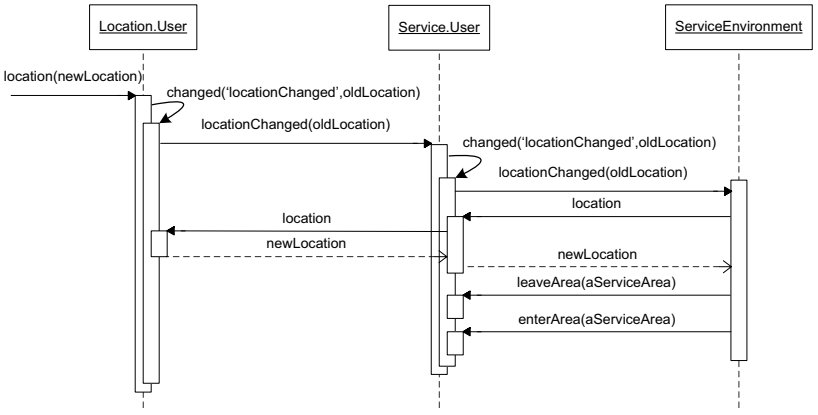

Figure 6. A three layer architecture for Location-Aware Services

When the environment gets this notification it calculates (by interacting with the user and the available service areas) which service areas the user left (if any) and which ones he entered. After the message `leaveArea` (or `enterArea`) is sent to the user, he will end with old services removed (or new added).

Adding other Context Models

In this section we briefly describe how we can add time constraints to our application using the same philosophy described in the paper. Suppose that we want to specify that a service is available at a certain area in a particular period of time. Following with the campus example, we would expect the course material service to be available at the time the course is being held; once the lecture is over, the material shouldn't be accessible as a room service. To implement this mechanism we must first add the notion of time and time events to the context abstractions; in order to do so we add a Time package containing the class `Timer`. The `Timer` class is a Singleton (Gamma et al., 1995) that has two main responsibilities: it can be queried for the current time and it can be configured to send time events in predefined moments.

Once this package is added to the application, we need to configure our services to have a time constraint: the service will only be activated in a predefined period of time. At first glance, the implementation of this constraint seems to be straightforward: as we have seen before, when the user enters a room, he triggers a change that ends with the user asking for the services available in a service area. When a service area is asked for its available services, it checks the service's constraints; a service will be available if and only if the time constraint is satisfied (this can be verified by asking the timer for the current time). Now, suppose that the user enters the room before the course starts;

since the time constraint is not satisfied, the course's material is not presented to the user. After a couple of minutes the course begins, but since the user is still inside the room (i.e. he hasn't left the room and re-entered it again) he doesn't have the course material service. The problem in this case is that, so far, service changes are only triggered by location changes, while time changes should also affect the services available for a user. To solve this problem, we need to be able to configure time events associated with time constraints: when a service is accessible during a specific period of time, time events should be generated at the beginning and at the end of the period, so that the services available for a user are re-evaluated. The events generated by the timer, are captured by the dependency mechanism and dispatched to the environment, which in turn asks the user object to analyze again the services provided by the service area that has just changed.

5. An archetypical Implementation

We have built a proof of concept of our architectural framework using a pure object oriented environment (VisualWorks Smalltalk) that supports dependency mechanisms and reflection, and where truly transparent distribution can be implemented. To achieve distributed objects collaboration in a transparent way we used the Opentalk framework, which we adapted to support PDAs sockets; we also extended the framework to perform object migration from one device to another. We used HP iPaq 2210 PDAs as user devices; user's location sensing was performed using infrared beacons and we are now adapting the sensing mechanism to work with bluetooth signals.

Our design prototype is not conceived to work on a client-server style, but mainly on a fully distributed environment shared between different devices. This approach promotes an environment where we can find different kinds of PDAs and desktop machines working together in a transparent way. We have filled our expectations so far, since we are interacting with wireless PDAs and wired PCs without any trouble; we also have upgraded our PDA hardware to HP iPaq hx2750 without even noticing it.

6. Related Work

We found our model of context to be quite similar with the one presented by Dourish (Dourish, 2004). While in most approaches, 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, Dourish proposes a phenomenological view of context. In this approach, context is considered as an emergent of the relationship and interaction of the entities involved in a given situation. Similarly, in our approach, context is not treated as data on which rules or functions act, but it is 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.

From an architectural point of view, our work can be rooted to the Context Toolkit (Dey, 2001) which is one of the first approaches in which sensing, interpretation and use of context information is clearly decoupled. We obviously share this philosophy though pretend to take it one step further, attacking inner application concerns. Hydrogen (Hofer et al., 2003) introduces some improvements to the capture, interpretation and delivery of context information with respect to the seminal work of the Context Toolkit. However, both fail to provide cues about how application objects should be structured to seamlessly interact with the sensing layers. Our approach proposes a clear separation of concerns between those object features that are "context-free", those that involve context-sensitive information (like location and time) and the context-aware services. By placing these aspects in separated layers, we obtain modular applications in which modifications in one layer barely impact in others. From an architectural point of view, our work has been inspired in (Beck and Johnson, 1994): the sum of our micro-architectural decisions (such as using dependencies or decorators) also generate a strong, evolvable architecture.

In the Java Context Aware Framework (Bardram, 2005), a Java-based framework is presented for building context-aware applications. Even though the framework presents a behavior oriented structure, it still models context in a traditional way (by means of context and context items) and makes an explicit separation between the entities and their context (in fact, entities explicitly know their context). In our proposal, we think of context as extending the base application behavior instead of viewing context as data to be acted upon. Since the layers are built on top of the application model, there is no need to change the core of the system in order to make it context-aware.

To summarize, in our approach we see *context aspects* as active objects and the context itself as an emerging property of their interaction. To achieve independence between the contexts aspects and the sensing mechanisms we placed a layer between them, so that changes in one model does not affect the other. At the architectural level, and thanks to the increasing power of the mobile devices, we decided to work with distributed objects instead of using a client-server architecture. In this way the applications running on the PDAs are responsible of handling the services of each user and can provide more advanced services than the ones provided by web pages, avoiding at the same time the scalability problems associated with concentrating all the processing in a single server. Lastly, in order to be isolated from lower level details, we decided to implement our framework on a pure object oriented environment as Smalltalk.

7. Concluding Remarks and Further Work

We have presented a new approach for designing location aware services and described how to enhance existing applications with new context-aware behaviors. By using a dependency mechanism to connect locations, services and application objects we have been able to avoid cluttering the application with rules. We have also improved separation of different design concerns, such as applicative, spatial, temporal, sensing, etc. Additionally, we showed how to achieve a finer granularity of design concerns with respect to existing approaches.

Our view 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 behavior. We briefly described a prototype system that we are using as a proof of concept for building context-aware services.

We are now working on the definition of a composite location system that allows symbolic and geometric location models to co-exist 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.

References

Abowd, G. D. (1999). Software engineering issues for ubiquitous computing. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 75–84, Los Alamitos, CA, USA. IEEE Computer Society Press.

Bardram, J. E. (2005). The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In *Pervasive*, pages 98–115.

Beck, K. and Johnson, R. E. (1994). Patterns generate architectures. In *ECOOP*, pages 139–149.

Dey, A. (2001). *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology.

Dourish, P. (2004). What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1):19–30.

Gamma, E., Helm, R., and Johnson, R. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., and Retschitzegger, W. (2003). Context-awareness on mobile devices - the hydrogen approach. In *HICSS*, page 292.

Leonhardt, U. (1998). *Supporting Location-Awareness in Open Distributed Systems*. PhD thesis, Dept. of Computing, Imperial College.

Rossi, G., Gordillo, S., and Lyardet, F. (2005). Design patterns for context aware adaptation, Workshop on Context-aware Adaptation and Personalization for the Mobile Internet.

Salber, D., Dey, A. K., and Abowd, G. D. (1999). The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441.

Sousa, J. P. and Garlan, D. (2002). Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA*, pages 29–43.