An Object-Oriented Approach for Context-Aware Applications*

Andrés Fortier[1,2], Nicolás Cañibano[1], Julián Grigera[1], Gustavo Rossi[1,3], and Silvia Gordillo[1,4]

[1] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
[2] DSIC, Universidad Politécnica de Valencia, Valencia, España
[3] CONICET
[4] CICPBA
{andres,cani,juliang,gustavo,gordillo}@lifia.info.unlp.edu.ar

Abstract. In this paper we present a novel, object-oriented approach for designing and building applications that provide context-aware services. Our approach emphasizes a clear separation of the relevant concerns in the application (base behavior, context-sensitive properties, services, sensing technologies, etc.) to improve modularity and thus simplify evolution. We first motivate the problem with a simple scenario of a virtual campus; we next present a new context model, which emphasizes on behavior instead of data. We next show the main components of our architecture and a simple approach to achieve a clear separation of concerns. We analyze the most important (sub) models in which we decompose a context-aware application and explain the use of dependency mechanisms to achieve loosely coupled relationships between objects. We also show how to take advantage of a reflective environment like Smalltalk to adapt the application's behavior dynamically and to provide transparent object distribution. We finally compare our work with others and discuss some further work we are pursuing.

1 Introduction: The Challenges of Context-Awareness

Context-Aware (and in particular Location-Aware) applications are hard to build and more difficult to maintain due to their *organic* nature [1]. For this reason, improving modularity and reducing tightly-coupled relationships between objects is extremely necessary when designing this kind of software.

Among the many issues involved in developing and maintaining context-aware systems, we consider the following as the main difficulties:

– **Context-aware systems integrate knowledge from different areas** such as HCI, artificial intelligence, software engineering and context sensing to produce the final application. Due to the extent of this discipline, we consider that the next generation of context-aware systems will need an integrating platform rather than a single application [6].

* This paper has been partially supported by the SeCyT under the project PICT 13623.

W. De Meuter (Ed.): ISC 2006, LNCS 4406, pp. 23–46, 2007.

- **Context information is acquired from non-traditional devices and distributed sources**, and must then be abstracted and interpreted to be used by applications [4].
- **Abstracting context means more than changing representation**. While interpreted context data is usually dealt as string data, applications are composed of objects, and many times those objects represent contextual information as well. There is a certain impedance mismatch between context and application data, even when they refer to the same concept.
- **Adapting to context is hard**; design issues related with context-aware adaptation are not completely understood and thus handled incorrectly. For example, the rule-based paradigm has been over-used in the last few years 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), we claim that more elaborated structures are necessary to improve maintenance and evolution.
- **Context-related information is usually "tangled" with other application data**; 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 context-aware software. In this paper we describe an architectural approach for dealing with the problem of providing context-aware services. Our approach is based on a clear separation of concerns that allows us not only to decouple context sensing and acquisition (as in [14]), but mainly to improve separation of application modules, to ease extension and maintenance. For this purpose we make an extensive use of dependency mechanisms to provide context-aware services and take advantage of the reflective nature of the Smalltalk environment to dynamically change objects' behavior and achieve transparent distribution.

The main contributions of our paper are the following:

- We show how to separate application concerns related with context awareness to improve modularity. This approach can be used to build new applications or to extend legacy ones by adding location and other context-aware services. A concrete architecture that supports this approach is presented.
- We show how to objectify services and make them dependent of changes of context; in particular we emphasize location-aware services.
- We introduce a behavioral point of view to deal with contextual information (instead of the current data view of most existing approaches).
- We show how to use the reflective capabilities of Smalltalk to model our conception of context.
- We show how to take advantage of transparent distribution mechanisms in mobile environments.

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 [16]. Our system already provides information about careers, courses, professors, timetables, etc. We now want users carrying their preferred devices to 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. Even though we use the user's location as an example, different contextual information such as the user's role or activity might also shape the software answer. For example, we could infer whether the student is attending to a lecture by sensing his position, checking his timetable, verifying the teacher's position and sensing the noise level in the classroom. As a consequence, the student's smarthpone could be switched to silent mode so that it doesn't interrupt the teacher in the middle of his lecture.

The first design problem we must face is how to seamlessly extend our application in order to make it 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 contextual parameters such as current time or 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 context-aware behavior.

When working with context-aware 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 location model (from symbolic to geometric [11]), and so on. While most technological requirements in this scenario can be fulfilled using state-of-the art hardware and communication devices, there are many design problems that need some further study:

- During the day the user is involved in a set of activities that constantly change what is contextually relevant. How do we model context so that it can follow this changes and effectively adapt to the user's needs?
- How can the system adapt to appearing (previously unknown) sensing devices? For example, in the moment the user enters in the range of a wi-fi access point new sensing devices may be available through the network. This new sensing information can be used to improve the user's context and therefore improve the system response to user's needs.
- Context information can be provided by a variety of sensing devices during a short period of time; for example, the weather conditions can be acquired from a software sensor (a web service when an Internet connection is available) or from a hardware device (a simple weather station installed in the

building accessed through a sensor network). Thus, even thought context is mainly built from sensing information, to easily adapt it to different technologies, it should be as loosely coupled as possible from the sensing mechanisms. How do we model this relationship?
- Context-aware behavior may depend on application functionality, but this kind of functionality tends to be volatile in comparison with the core application. How can the application cooperate with context-ware behavior in a way that changes in the later don't impact on the former?

The aim of this paper is to focus these problems, mainly those that characterize the difficulties for software evolution and to show how combining proven design principles with reflective facilities can yield a scalable architecture to build context-aware software.

3 Our Context Model

Even though context is recognized as having a dynamic nature, it is usually treated as a fixed piece of data upon which some behavior is executed, which we consider neglects its essence. From our point of view, context should be modeled as close as possible to the phenomenological view presented by Dourish [5]. Taking his definition as a starting point, we claim that the following properties must hold when modeling context:

1. *Context is not passive.* The context model must take an active role in the application and shouldn't be considered as a static collection of data. If decisions must be taken according to the context, then the context itself should be actively involved in that process.
2. *Context is not stable and can not be defined in advance.* Since context is inherently dynamic, we can not constraint the context information that will be modeled when designing the application. Since context-aware applications are supposed to adapt to the user's activities, what is contextually relevant to the user is constantly changing. Therefore, our model of context should be able to represent these dynamically varying aspects of the user's context that are relevant in a given situation.
3. *Context is independent of the sensing mechanisms.* Even though context information is usually gathered automatically, the representation of this information should be independent of the sensing mechanism. As we will see in the following sections, if our context model is tightly coupled to the sensing hardware, the system evolution will be heavily compromised.

In order to fulfill these requirements, we decided to split context in a set of *context aspects*, each one responsible for modeling the behavior of a specific context feature. As a result, the context is not seen as data provided by external acquisition mechanisms, but as the emergent behavior provided by the interaction of the different context aspects. This behavior, encapsulated in every context aspect, varies as the application runs, allowing to provide different

adaptation to the user according to his context. By using this scheme, what we are actually modeling is the behavior that is contextually relevant to the user in a given moment of time.

4 The Object-Oriented Architecture

As previously mentioned, to achieve flexible context-aware applications a set of main concepts must be identified and clearly separated. Other approaches for building context-aware applications have also identified concerns that must be separated to achieve modularity: sensing (implemented for example as Widgets in [4]), interpretation (also mentioned as context management in [9]) and application model. Layered architectural approaches [9], or MVC-based ones like [14] provide the basis for separating those concerns using simple and standard communication rules. However, applications 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.

As a case study, let's analyze the location aspect of the example in greater detail. Context-aware applications are supposed to aid the user during his daily activities, which means that they must be able to cope with location models of different granularity; for example, if we want to offer a set of services in the user's office, small distances are important and a fine-grained location model should be used. On the other hand, if we want to offer services when the user is moving in his car along the country, the granularity of the location model must be larger, so that adaptation can be provided according to the user's activity. Unfortunately, there is no silver bullet for location models and we must use different location representation according to the specific requirements.

Symbolic models [11] that represent inclusion, adjacencies or distances between locations characterized as symbols are generally well suited for indoor location. These models represent in a clear way those relationships between the locations, can represent very fine-grained structural arrangements and are easy to understand by humans. On the other hand, Geometric models [11] are well suited for representing large areas where accuracy does matter or when it is necessary to calculate distances between objects with an important level of correctness. As a drawback, creating geometrical models for large maps (for example, a city) is a hard task (when not impossible) without the aid of a specialized organization. Also, geometrical models don't add semantics to the regions they represent, they just provide the boundaries for performing calculations; so, if we want to tag a specific polygon as being the *Plaza Hotel*, we must do it manually.

In this area it's worthwhile mentioning Google Earth and it's Community. This application uses different levels of detail of geometric locations, allowing to easily compute distances between two places, get 3D views of the cities and print maps with country borders. When routing information is available, we can also ask for the shortest route between two places. Although Google Earth doesn't handle symbolic locations, the Community users can tag latitude/longitude points with descriptions in order to give semantic to geometric points. Descriptions can vary from

simple strings explaining what the place is, to a complete description with photos and links. Even though tags give semantic to geometric places, they cannot be consider symbolic locations since they don't offer any location information; symbolic locations (and locations in general) must provide meaningful information in order to compute at least one operation by themselves, which is not the case (a tag without the geometric point where it's attached to can't perform any operation).

As can be seen from the example above, a single context aspect (such as the user's location), can turn out to be a whole world in itself. On top of this, the forces that may impose changes in a context aspect have nothing (or very little) to do with the application model or with the services that are provided. For example, when adapting the user services to his location, the services subsystem should only care about knowing *where* the user is (for example, in the library) and not about the details of *how* the location model determines the user's location. Also, we should remember that context is essentially dynamic and that context aspects can't be fixed at design time. For this reasons, in the same way that MVC [10] architectures argue for a clear separation between the model and the view, we argue that a similar separation must hold between the application model, the context model, how the context is sensed and the way services are offered to the user.

4.1 Main Components

In order to tackle the issues previously mentioned, we decided to decompose a context-aware application in two orthogonal views: the application-centered view and the sensing view. The application-centered view is concerned with context and service management and is in turn separated in three layers. The sensing view (which will be described in Section 5) is concerned with the mechanisms used to get external data and how to feed this data, as transparent as possible, into the context aspects. In Fig. 1 we show the layers that comprise the application centered view.

In the **Application Model** layer we specify the application classes with their "standard" behavior (i.e. those application classes and methods are not aware of the user's context, neither they exhibit context-related information). In our example this layer would have classes like `Room`, `Schedule`, `Teacher`, and so on. The **Context** layer is in turn composed of a set of context aspects, each one creating a high-level representation of a specific part of the user's context. As can be seen in Fig. 1, we take a different approach when it comes to establishing the relationship between the application and the context model: in most approaches, context is presented as data that is somehow processed by the application, thus creating a knowledge relationship from the core application to the context representation. As explained in the previous section, we consider that this approach is rather limited and that a behavioral point of view should be taken to handle context. In our architecture the application does not fetch data from the context, but it is the context itself who extends the application functionality. By separating context in independent aspects we are able to add the behavior that is contextually relevant in a given moment, so that it is the aspect itself the one who decides how the adaptation is made.

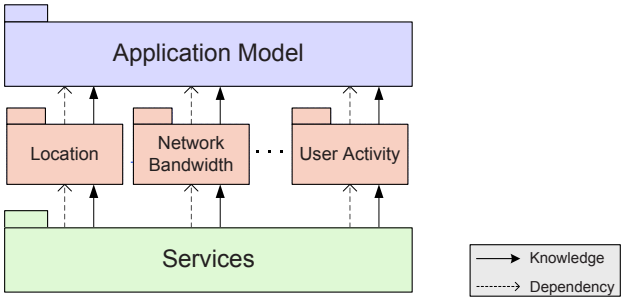

Fig. 1. A layered approach for the Application-Centered view

The **Services** layer models the concepts related to a context-aware application from a services point of view. This layer has the responsibility of deciding what services are available to the user in a given situation. For that purpose, four main abstractions are modeled: the user, the services, the services environment and the service providers. As expected, the user object represents the physical user who is carrying a mobile device and which is the focus of the system's attention. The user in this layer has attached a set of context aspects that can vary dynamically, according to what is considered contextually relevant in a given time. When there is a change in the user's context (either because a context aspect changed or because a context aspect is added or removed) the user's available services are updated according to the system configuration. This configuration basically establishes a relationship between the services and the different contexts in which the user may be, generally represented as a constraint (i.e. a certain service is provided if a constraint is satisfied). As an example, when working with location-dependent services, each service will be associated with a set of geographic areas represented in the location aspect; when the user changes his location, new services may become active or former ones removed.

Each service is associated with a service provider, which is in charge of creating the service when required, acting as a Builder [7]. Besides, every service provider defines a set of constraints to determine when it is allowed to give its services to the user. Upon a change in the user's context, each provider is asked to reevaluate his constraint in order to determine if a new set of services should be added to the user or existing ones removed. Finally, services, providers and users are immersed in a service environment, which reifies the real-world environment from a services view. The service environment acts a Mediator [7] between service providers, services and users.

4.2 Communication Mechanisms

As shown in Fig. 1, we use a layered approach to separate the application-centered view concerns. In this view, context aspects are the basis on which the services layer is mounted. In the services layer, the user object holds the set of currently active context aspects, which are used to determine what services

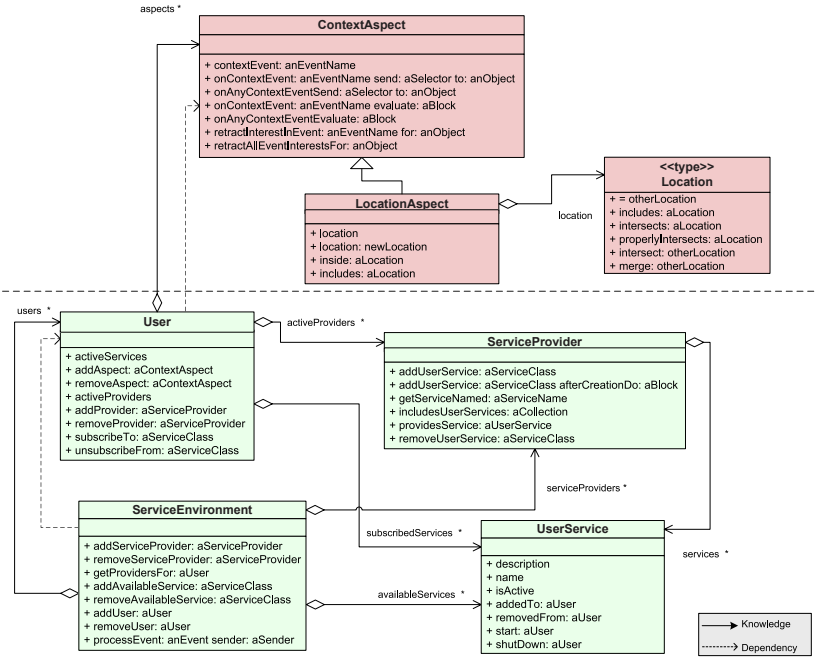

Fig. 2. Relationships between services and context aspects

are available at a given time. As we mentioned earlier, in order to improve evolution, context aspects should be decoupled from the services layer, which is accomplished by making an extensive use of the dependency mechanism. Fig. 1 shows how the Service layer is dependent of every context aspect, so that it can get a notification when there is a context change. If we dig a little inside the services layer we will see that the relationship is established between the user and the services: every user has a collection of context aspects, which he is in turn dependent of. When there is a change in a context aspect the user gets the corresponding notification. This notification is in turn propagated to the environment (again, using the dependency mechanism) which is in charge of triggering the service providers re-evaluation. In order to establish dependencies we originally used the standard notification mechanism, but as the system grew in complexity we decided to implement our own event subsystem. To do so, we modeled a set of events, handlers and adaptors to easily trigger context changes and configure the system response to those changes. In Fig. 2 we show a class diagram depicting these relationships, using Location as a concrete example of a context aspect.

4.3 Creating, Deploying and Activating Context-Aware Services

Creating New Services. New services are defined as subclasses of the abstract class `UserService` and thus a specific instantiation of a service is an object that

plays the role of a Command [7]. The specific service's behavior is defined by overriding appropriate methods of `UserService` such as `#addedTo: aUser` (triggered when the service is made available to the user), `#start` (triggered when the users selects the service form the available services list), and so on. For example, a service that presents the course's material in the room where a lecture is taking place would be defined as a subclass of Service, and the message `#setUp: aUser` would search in the schedule for the appropriate objects to display.

Subscribing to Services. In order to access the services supplied by a service provider, the user must subscribe to those services he is interested in. Once he is subscribed, when the constraints imposed by the provider are met (e.g. being in a particular location at a certain time), the services are made available to the user. In our model, the service environment knows which services are registered, and therefore the users can query the environment for existing services to subscribe. Besides, when new services are added to the environment, a change notification is broadcasted to the users so that they can take the proper action according to the configuration they state. The default action is to show a visual notification (a small icon) indicating that new services are available, but the user may decide to configure the system to ignore notifications or to automatically get subscribed to any new service that appears. It's interesting to notice that this functionality is provided by a standard service which has the environment as its model.

Registering Services in Specific Providers. To provide context-awareness and to avoid the use of large rule sets, services are associated with (registered to) specific providers. When the user's context satisfies the provider's constraints, all services registered to that provider (to which the user has subscribed) are made available. By using the concept of service providers, the architecture also achieves the desired independence from the sensing mechanism, i.e. the circumstances under which the services are made available to the user don't belong to the scope of a sensing device (e.g. receiving a beacon signal) but to logical constraints. These logical constraints can be specified programmatically or interactively: they can be obtained by applying a set of operators to specific context concepts (for example, in the case of working with the location aspect, the operation may involve rooms, corridors, etc) or defined arbitrarily in terms of the user preferences (which can involve any context aspect).

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 as 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; 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 (i.e., changing the restriction

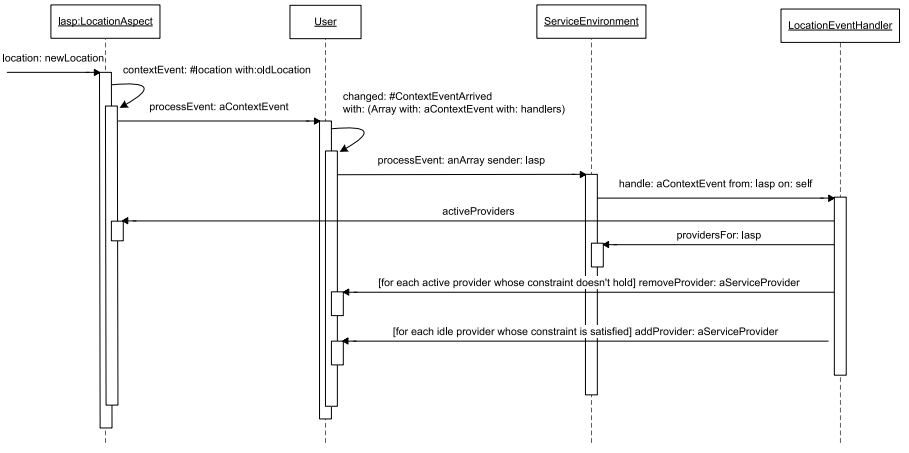

Fig. 3. Updating services as a response to a change in the location aspect

of the service provider), which in case of symbolic location means changing the location "Building" to "University Campus". It is important to notice here that the location constraint is not expressed in terms of sensing information, but in terms of the location model.

Service Activation. When the user's context changes (for example, as he moves in a location-aware environment), it triggers a notification event that reaches a `User` instance in the Service layer. Then, this object interacts with its environment to determine if a previously inactive service provider should be active, or if a currently active provider should be removed. In case a new provider is added to the user's active ones, the corresponding services are made available to him according to his subscriptions. As mentioned before, a service is presented to a user if it is available in an active provider and if the user is subscribed to it. A subset of these interactions is shown in Fig. 3 by means of a UML sequence diagram, where the location aspect is used as an example.

4.4 Context as Behavior Added to the User

In Section 3 we presented a context model in which context was represented in terms of behavior and not data. From the application point of view, context aspects were seen as software modules that added new behavior to the core application, using this core functionality when needed. From the services point of view, context aspects are responsible for deciding when a given service can be provided to the user. Since the service model is user-centered, the context aspects are applied to the user itself, which is modeled in the service layer by the `User` class.

As we stated earlier, each context aspect represents a unit of behavior that is contextually relevant to the user in a given moment. From that point of view, when a new context aspect is added to the user, the user's behavior is extended with the behavior provided by the context aspect. So, apart from the behavior

defined in the `User` class, each particular instance of `User` will behave according to his currently available context aspects. Achieving this kind of functionality is easy in Smalltalk, since we can rely on the *`#doesNotUnderstand:`* message to forward the message send to the available context aspects. Of course, as with multiple inheritance, there is a problem if more than one context aspect implements the same message. This conflict can be tested when a new context aspect is added to the user and raise a notification in case of conflict. Although this solution is not optimal it turned out to be quite handy in practice. As a second choice, the sender of the message can ask for a given aspect and specify explicitly who should receive the message, very much like the Role Object [3] pattern.

5 Handling Different Sensing Mechanisms

As explained in Section 4.1, our architecture is decomposed in two views: the application-centered view, which has already been explained, and the sensing view, which is in charge of feeding sensor data to the context aspects. Since the idea of a context-aware application is to give functionality to the user in a transparent way, context information must be gathered automatically (i.e. sensed). To make matters worse, hardware devices used for this purpose are usually non-standard, and the data these devices deliver is often far from what an applications needs: while sensed data is represented as strings, numbers or pairs, our applications are built in terms of objects. For these reasons, we consider context sensing as an important architectural problem. In order to tackle it, we have developed a modular design so that changes in the sensing features don't impact in the application. We based our design on the fact that the context model and the way it is sensed are of different nature. In a context-aware application several sensors may be used for determining information about a single context aspect, and, at the same time, a single sensor's data may be used in many ways for inferring information on different context aspects.

As an example consider a user moving with his PDA inside a building. To detect where the user is standing, we can take advantage of the Bluetooth receiver in his PDA and place a Bluetooth beacon in every room. Each beacon can send different room IDs, which will be interpreted to know what the user's location is. Now suppose there are billboards hanging on the walls inside the building, which we would like to enhance with digital services (for example, showing the web-version of the billboard, or having the chance to leave digital graffiti). In order to do this we can use an infrared beacon to capture the user's location, this time with a finer granularity than the one provided by the Bluetooth beacon and with the added value that we know the user is effectively pointing at the billboard, since infrared signals are directional. In this example, the location of the user is being sensed by two different devices at the same time, one giving more detailed information than the other. As a second example consider the case when the user is not carrying the receiving sensor; for instance, suppose that the user is wearing a Bat unit [8] that constantly sends a unique identifier. This identifier is captured by the receiver units placed on the ceiling of the rooms to

calculate the user's location. In order to use this system we need a mechanism that allows us to monitor the user's location, even though the value is being sensed by an external receiver. This means that the context model should be independent of the physical location of the sensors; it shouldn't matter whether the sensors are attached to the PDA or placed on the room's ceiling.

The examples presented above shows that the way context is represented and the way it is sensed belong to different concerns. This is why, as shown in Fig. 4, we consider sensing as a concern that cross-cuts the context aspects layer.

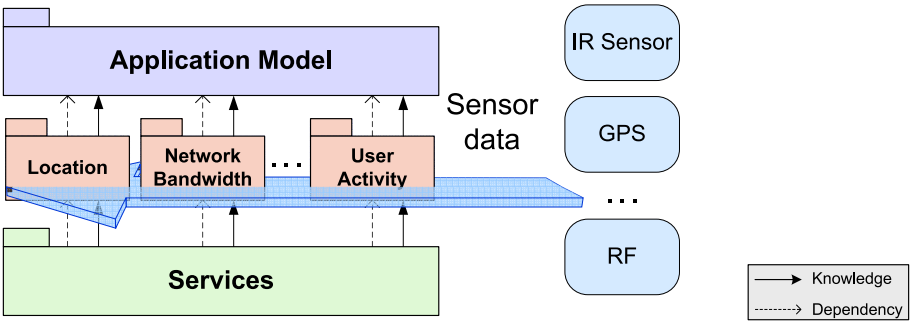

Fig. 4. Sensing as a cross-cutting concern

In order to decouple context modeling from acquisition, our architecture adds a layer between the hardware devices and the context model (see Fig. 5). The basic abstraction in this layer is the sensing aspect (modeled in the `SensingAspect` class), which is basically an object that watches over a hardware sensor and reacts to every new data arrival. To implement this, it has a policy that determines whether the values should be pulled or pushed from the sensor. When creating a new sensing aspect, programmers must provide the message to be sent to the context model when sensed data changes. To improve this task, we have created pluggable sensing aspects that can be configured with a block, and adaptable sensing aspects that take a message to be performed, pretty much like the standard `AspectAdaptor` and `PluggableAdaptor` from the Wrapper GUI framework. In case a more sophisticated behavior is needed, the programmer can create his own `SensingAspect` subclass.

Another important issue when using sensors is to decide whether every signal must be sent to the application or not. In some cases we won't need to forward all the information that sensors deliver, avoiding cluttering the application with

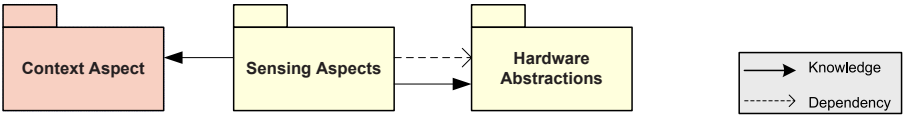

Fig. 5. Layered approach for the Sensing view

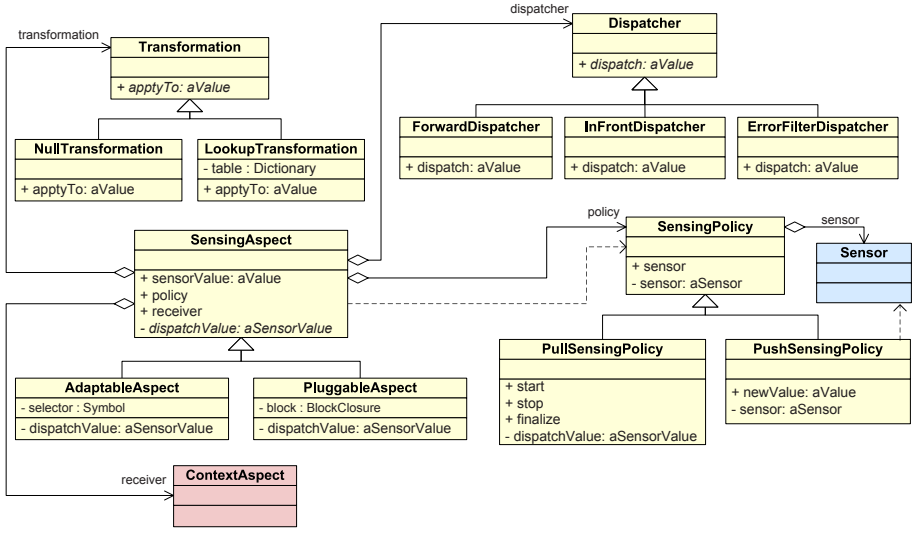

Fig. 6. Class diagram of the sensing aspects package

constant notifications. As an attempt to solve this problem we have introduced the notion of dispatchers (represented in the abstract class `Dispatcher`) that has the responsibility of deciding which signals are let into the system. For example, an `ErrorFilterDispatcher` would be in charge of filtering those signals whose noise level is beyond a given threshold. To complete the process of gathering information from sensors and feeding it to the context model we need to solve the mismatch between sensors' data and applications' needs. To address this problem, we have created the `Transformation` abstract class. Transformations are intended to convert atomic values delivered by sensors to full fledged objects which will feed the context aspect. This transformation can range from simple table lookups to complex machine reasoning processes. Fig. 6 shows the complete class diagram of the Sensing Aspects package.

6 A Pure Object-Based Distribution Scheme

Context-aware applications generally need to share information between a set of devices. This not only encompasses sensing devices (e.g. a server that publishes information gathered by a weather station located in the building) but different kind of services (e.g. location-aware messengers and reminders, shared virtual folders, friend finders, etc.). For this reason we need a mechanism to distribute the proposed architecture between a set of devices and let them share the information they need. Since we can't expect to know the characteristics of every device beforehand or the connection speed between any two devices, the distribution mechanism shouldn't impose a fixed communication schema between the devices; as a matter of fact it should allow objects to *live* in different machines as needed. As an example let's consider again the user and his location aspect and

let's analyze how these objects can be distributed considering that the user emits his unique *id* using the Bluetooth port of his PDA. If we decide to move the whole computation to the client device, we would model the external Bluetooth receiver as a Proxy [7] linked to the machine that is physically connected to the Bluetooth receiver. Each time the desktop machine receives a new signal it will forward the received value to the client, which will then process it as explained in the previous sections. On the other hand, if the client device can't handle this kind of computation, we should move as much as we can to the desktop machine, so that the information arrives as processed as possible. In this example we could move the whole dispatching process to the desktop machine, so that the client only receives an update when his location has changed (notice that this also means saving network bandwidth). Following this idea we could take this a step further and move the whole location aspect to the desktop machine, which would be represented in the client machine as a proxy to the real object residing in a remote machine. This means that each time the user changes his location there would be a remote computation to find out which services are added or removed, and as a result the instance of the `User` class residing in the client device would receive the `#addService:` or `#removeService:` message. Of course this last strategy implies a big risk: if the user gets out of the wi-fi range or the network goes down he loses all location information.

In order to cope with the distribution concern we decided to use the Opentalk [17] framework and extend it to cover our needs. In the next section we will give a brief review of the framework and then explain the extensions we've made so far.

6.1 Opentalk Basics

Opentalk [17] is a distribution framework for VisualWorks that gives flexible support for the development of distributed applications. To achieve this, Opentalk is organized in a series of layers, which includes communication protocols, object services, brokers and remote objects. The Opentalk Communication Layer provides the basic abstractions for developing the connection protocols. These protocols can operate on top of TCP/IP or UDP transport layers. In particular, our work is based on TCP/IP, using the built-in support for Smalltalk-to-Smalltalk interoperability.

In our framework we have exploited the cross-platform nature of the VisualWorks implementation to manage the issue of heterogeneous networks imposed by mobile systems. Thanks to the different VMs available we are able to support several operating systems running over separate hardware platforms (we are currently working on Linux, Windows 2000 and XP on desktops and Windows CE and Mobile on Pocket PC). A VisualWorks image can run almost identically on any supported platform[1], thus allowing the arrangement of a heterogeneous network.

[1] In the case of Win CE or Windows Mobile deployment, we have to take into account also those issues concerning the graphical user interface and display screen size (such as a proper Look-&-Feel and a minimal and convenient layout of visual components). In addition, the processor speed of the target machine and the memory footprint of the deployed image must be taken into an account if we want the final application to run decently.

Request Broker and Remote Messaging. Opentalk provides a complete request broker implementation, which is combined with the Smalltalk reflective capabilities to provide transparent remote communication between Smalltalk images. The communication between objects residing in different environments (images) depends on the functionality provided by those brokers. In order to be remotely accessible, an object must be exported through a local broker, which in turn assigns an object identifier (oid) and registers the recent exported object on an internal table. Once the object has been exported, any remote object containing a proper reference to this object, can collaborate with him by directly sending normal messages. This transparency is achieved thanks to the use of generic proxies (discussed in the following section). However, before any two objects can remotely communicate, it is necessary to resolve the initial reference between the two hosting images (i.e. between both request brokers), which can be easily achieved by exporting a root object with a predefined name. Once this is solved, subsequent communication is done transparently due to the inherent characteristic of navigation by reachability and the use of proxies in conjunction with the underlying machinery provided by the brokers.

The internal structure of the brokers is a bit complex, since it collaborates with many different kind of objects to effectively allow for remote message interchange. Among the main collaborators we can mention the `ObjectAdaptor`, `Listener`, `Transport` and `Marshaller`. The object adaptor is in charge of registering those objects that have been exported through the broker and to coordinate listeners and transports. The listener is constantly waiting for new connection requirements incoming from other hosts. When one of this requirements arrives, the listener accepts it and creates a new transport (i.e., a new transport is created for every connection between brokers). Every transport is in turn bound to a specific socket and works as the entry point for receiving and delivering remote messages. In order to accomplish this task an encoding procedure must be done, so that remote message sends can be converted to a binary representation (i.e. a byte array). This procedure (known as marshalling) is performed by the marshaller, who is in charge of transforming remote messages into transport packets. Each transport will collaborate with his own marshaller and will use his services to encode and send messages and to receive and decode them.

To mimic the message sends in the local images, brokers use a synchronic communication policy. The broker that is in charge of dispatching outgoing messages (an instance of `STSTRequest`) will send the remote message and wait for the response (an instance of `STSTReply`). This message send will cause that in the image where the actual object is residing a local message will be send, faking the remote invocation as a local one. Once the message has been dispatched, the sending image will be waiting until a response arrives or a time period expires. If the response arrives, the returning object is decoded and control is passed back to the object that originally sent the message.

Remote Objects and Proxies. A remote proxy [7] is a local representative for an object (its *subject*) that resides in a different address space. When a message is sent to the proxy, it automatically forwards it to its subject and waits for its

response. Thus, from the sender point of view, there is no difference between working with local or remote objects[2].

In Opentalk, the `RemoteObject` class performs the role of remote proxy and maintains an indirect reference to its subject by using an instance of `ObjRef`, which is basically composed of a socket address (i.e. an IP address + a port number) and an object identifier (also known as *oid*). When an instance of `RemoteObject` receives a message, it forwards the request to its subject by means of the *`#doesNotUnderstand:`* mechanism; for this purpose the proxy leans on the broker's services, which is responsible for deliver the message through the network. Also, whenever an object is exported (i.e. passed by reference[3]) from one image to another, a new proxy is created to represent the remote object. This instantiation task is responsibility of the local broker.

In Opentalk the concrete `RemoteObject` class is a subclass of `Proxy`, which is a special abstract class that does not understand any messages. The `RemoteObject` class automatically forwards those messages that does not understand, leading to a generic implementation of a remote proxy. This basic implementation allows us to apply seamless distribution to any existing application with relative little effort. Therefore, we can postpone this decision until last moment (or when considered necessary) without worring about distribution issues on early stages of system development. However there are some trade-offs that we must be taken into consideration when working with this kind of distribution technique, in particular when dealing with mobile systems. In the first place, the proxy generally implies an important network traffic, since messages are constantly flowing through the network. Also, this approach needs a constant connection between the images where the proxy and the subject are, not being well suited to distribute objects in networks with intermittent connections. Also, even though we can assume having a continuous connection, in mobile applications we expect the user to be moving around large spaces. As a consequence, accessing the host where the subject resides can be fast in a given time, but extremely slow if the user has moved to a place where the connection to access that host is slow. In these cases, we would like to be able to move the subject to a host that can be accessed with less network delay. Hence, we are motivated to figure out how to take the maximum advantage of this approach and combine it with new alternatives in order to cope with these issues.

Pass Modes and Distribution Strategies. Opentalk provides a fixed set of general purpose object pass modes which indicates how an object will be distributed across images: by reference, by value, by name and by oid. A pass mode can be seen as a strategy for object distribution, because it decides the way in which an object should be distributed when exported to another host; therefore, we will use interchangeably the terms pass mode and distribution strategy. For the sake of conciseness (and because pass by name and pass by oid

[2] There is a subtle issue regarding object identity and pass modes that we will not address due to space reasons.

[3] As we will see in the next section, there are many ways of distributing objects.

modes are not frequently used) we are going to describe the two most relevant pass modes: by reference and by value. The first one is the basic proxy approach: the subject resides in a single image and the other images have proxies that refer to the subject. On the other hand, passing an object by value means that a copy is sent to the requesting image. It is important to notice that this distribution policy does not guarantee that both objects will be consistent; as a matter of fact, passing an object by value is like creating a local copy, and then moving it to another image, so future messages sends may alter their structures without any restrictions or implicit synchronization.

The way an object will be passed can be decided at the class or instance level. In order to indicate the pass mode of all the instances of a given class, the `#passMode` message must be redefined in the desired class (by default all objects are passed by reference). If we want to specify how a specific instance should be passed across the net, the `#asPassedByReference` or `#asPassedByValue` messages can be sent to the specific instance. Sending any of these messages will end up creating an instance of `PassModeWrapper` on the receiver, which will mask the original pass mode defined in the object's class.

As we stated before, we found Opentalk proxy distribution mechanism to be very well suited, especially because of the transparency it provides. On the other hand, in order to accommodate to mobile environments, we found it necessary to enhance the framework to provide new distribution policies, which are explained in the next section.

6.2 Opentalk Extensions

In order to accommodate our needs we devised a series of extensions to the Opentalk framework. These extensions include traceability, migration and object mirroring. In order to clarify the ideas that will be covered in this section we briefly explain the main concepts:

- Traceability is the capability that an object has of knowing all the remote references that have him as a subject. This can be seen as asking an object for all his remote owners.
- We refer to replicas when we talk about a copy of some object that resides in a different Smalltalk image. This doesn't mean that there is any kind of connection between the original object and the replica; a replica is just a copy of the object with no synchronization mechanisms.
- Migration means moving an object from one image to another. This movement must be consistent and rearrange any remote reference to update its address to the new host (note that local references can be easily converted by using the `#become:` message, while remote ones will require a more sophisticated mechanism).
- Mirroring refers to having an object replicated in a way that the replicas are consistent. So, if an object is modified in an image, all the distributed mirrors are modified to maintain the consistency.

As we will see, by adding these features, we can distribute objects in new ways and have a flexible base to dynamically change distribution policies to adapt to context changes.

Extensible Pass Mode Hierarchy. The first task we had to accomplish was a redesign of the way pass modes where modeled. In the original framework, pass modes where represented by a symbol (i.e. #reference, #value, #name, #oid), making it impossible to delegate behavior to the pass modes themselves. To solve this issue, pass modes are now represented as first-class objects and modeled by a class hierarchy rooted at the `PassMode` class. Thanks to this first modification we obtained a flexible way to add new pass modes. The basic pass modes are represented by the classes `PassByReferenceMode`, `PassByValueMode`, `PassByNameMode` and `PassByOIDMode`. Each of these classes redefines the abstract method *`#marshal:on:`* which uses double dispatch to delegate the specific encoding of the object to a marshaller.

In order to facilitate the creation of new pass modes, the `CustomPassMode` class is defined to act as an abstract class. This class redefines the *`#marshal:on:`* message to provide a Template Method [7], so that new pass modes only need to redefine their specific features. This class is the root that we used to define the new pass modes.

Traceability. Traceability is defined as the capability that an object has of knowing all the remote references that have him as a subject. This is achieved by knowing those places to which it was exported and then tracking those remote proxies that are referencing it. Traceability is implemented as a special type of passing an object by reference and is modeled by the `TraceablePassMode` class.

When a *traceable* object is exported, a local wrapper (`TraceableObjectWrapper`) is created to hold a collection of the remote proxies that are referencing the target object in other hosts; we will refer to this original object as a *primary copy.* When a host requests for a proxy to the primary copy, instead of creating a remote object, an instance of `TraceableRemoteObject` is instantiated in the remote host. This object acts basically like a standard remote object, but adds the necessary behavior to notify the primary copy that a new reference to it has been created and to notify it when the proxy has been garbage collected in order to remove the reference. These notifications are really captured by the wrapper created on the primary copy, which is responsible for keeping the remote references collection.

As we will see in the next sections, by adding traceability we can choose between the interested hosts (i.e., hosts that have a remote reference to the primary copy) to mirror or migrate the primary copy. Also, things like distributed garbage collection by reference counting can be easily implemented by adding the required logic on top of the traceability mechanism.

Migration. As was introduced earlier, an object can be replicated in many other images. Once these replicas are created there is no synchronization between the original object and the remote replicas. To perform this remote copy, a

`Replicator` object is introduced. This object is in charge of coordinating the hosts involved in the copy process, which can be triggered explicitly or by defining a set of events related to the environment. In order to fulfill his task, a replicator uses a Strategy [7] that allows to configure how a replica will be made. The most basic one is the `PlainReplication` strategy, which just makes a copy of the object in another image. A more interesting one is the `MigrateAndRedirect` strategy, which migrates the primary copy to another image and rearranges all the remote references to update their information about the host that now holds the object. Also, during this process, all message sends to the primary copy are temporarily frozen so that no inconsistencies can arise.

The migration mechanism was originally needed in order to give the user flexibility when working with portable devices (such as PDAs or smartphones) and desktop computers. Imagine that the user is working in his desktop using his favorite context aware application. Suddenly, a reminder appears notifying that he must go to the airport to catch his flight. Now the user asks his application to shut down, but before doing so, the application tries to find the user's PDA in the network. In case it does, it launches the application by executing a shell command and migrates all his primary copies. As a result, the user automatically has the same up-to-date information in his PDA, with the additional benefit that any remote reference will be properly updated to reflect the host migration of the primary copy (of course, assuming that the PDA has a global connection to the net).

Mirroring (Synchronized Replicas). In contrast with the plain replication, the mirroring mechanism allows to keep a set of replicated objects in a consistent state (i.e. if an instance variable of one of the objects is updated, the remote replicas are updated to be consistent with the object). Associated to this mechanism, three new distribution policies are implemented[4]: `ForwarderPassMode`, `MirrorPassMode` and `StubPassMode`. An object that is exported under any of these three new pass modes can be dynamically changed to any of the other (e.g. an object passed in *forwarder* mode can be changed dynamically to be exported in *mirror* mode). Next we present a brief description of each strategy:

Forwarder. An object exported under this pass mode will forward every message to the primary copy, behaving like an object passed by reference. The added value of this class is the ability of dynamically changing the distribution policy to mirror or stub my receiving the messages `#becomeMirror` or `#becomeStub`.

Mirror. An object exported as a mirror will create copies of himself in the other images, making sure that all the replicas are consistent with the original object. In order to keep this consistency, a mirror object delegates the synchronization mechanism to a Strategy [7], which can be specified at the class or the instance level. At the moment we have implemented two synchronization mechanisms:

[4] These set of passing modes are inspired in the distribution strategies used by GemStone.

- A simple one, that just forwards the change to every replica. This strategy doesn't check if the update has been done in a consistent way, assuming an optimistic update. Note that this can easily end up in desynchronized objects in the case that mirrors of the same object are updated at the same time in different images.
- A two-phase strategy, that ensures the objects consistency. In the first phase, the object whose internal state has been updated triggers a notification that will cause the blocking of every mirror by asking for his lock. If all the locks can be successfully obtained, the change is propagated and the objects are unlocked. In the case that the lock can't be obtained a rollback is performed.

In order to support mirrors in a transparent way the immutability and modification management mechanism present in VisualWorks is used. This mechanism allows tagging an object as immutable so that a change in his state triggers an exception. We use the **ModificationManagement** package to have a simpler way of handling changes, by creating a subclass of `ModificationPolicy` (`MirroringPolicy`) which triggers the mirror updates.

Stub. A stub can be seen as a special kind of proxy, that waits until someone sends a message to it. When this happens, the stub gets replaced by a mirror, creating in this way the notion of a lazy-mirror. In order to perform this, the stub sends to himself the message *`#becomeMirror`* and then re-evaluates the message that was originally sent to him. As expected, a stub can be sent the messages *`#becomeMirror`* or *`#becomeForwarder`*.

Dynamic Change of Distribution Policies. Distribution policies can be changed in two granularity levels: at the class level, by redefining the *`#passMode`* message and at the instance level by using an instance of `PassModeWrapper`. As an extension to this basic mechanism, a family of distribution policies that can be changed dynamically has been introduced (`Forwarder`, `Mirror` and `Stub`), allowing to change the way that mirrors are synchronized. With these tools at our hand, not only can distribution be made transparent to the programmer, but we can also decide what is the best way to distribute a given object. As an example, consider an object in a context-aware system whose instance variables are constantly being updated. If this object is distributed by mirroring, we should expect to have an important network traffic and system unresponsiveness, since every modification implies a locking and an update. On the other hand, if we distribute this object by using a forwarder, the network traffic will be proportional to the messages sends to the primary copy and not to the instance variable update ratio. Of course, this can in turn become a bottleneck, since many hosts would be sending messages to a single image and asking for those messages to be resolved remotely. In order to overcome this issue, we can even use a mixture of distribution policies to balance the charge in a host: the primary copy can be mirrored in a small number of hosts, and then be distributed to the rest of the hosts by using forwarders. In this way, the load is distributed among the hosts that have the mirrors.

7 Related Work

From the conceptual point of view, we found our model of context to fit quite well the ideas presented by Dourish [5]. While in most approaches context is treated 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 in which 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. This idea is based on the concept of a context aspect, that represents the behavior that is contextually relevant to model in a specific situation.

From an architectural point of view, our work can be rooted to the Context Toolkit [4] 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 we expect to take it one step further, attacking also the application concerns. Hydrogen [9] 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. To achieve this modular architecture we based on the work of Beck and Johnson [2] in the sense that the sum of our micro-architectural decisions (such as using dependencies or decorators) also generate a strong, evolvable architecture.

Schmidt and Van Laerhoven [15] proposed a middleware architecture for the acquisition of sensor-based context information, which is separated in four different layers: sensors, cues, context and application. The sensors layer is where both physical (such as cameras or active badges) and logical sensors (like system time) are located. Data obtained from sensors is processed by cues on the next layer, whose main function is to synthesize and abstract sensor data by using different statistical functions. Values generated by cues are buffered in a tuple space, which provides for inter-layer communication between the cues layer and the context layer; then, the context layer can read this values and take the appropriate actions. In this approach, the use of middleware architectures helps decoupling the sensing hardware from context abstractions. Our approach also places sensing mechanisms into a separate module, but it does not depend directly on any other; it is treated as a crosscutting concern of the context model, what makes it less sensitive to system changes.

Other approaches that have been presented pay closer attention to monitoring resources and consider adaptation in terms of network bandwidth, memory or battery power. Among these works we can mention Odyssey [12] which

was one of the first systems to address the problem of resource-aware adaptation for mobility. In this approach there is a collaborative partnership between the operating system and individual mobile applications, in which the former monitors resource levels and notifies the applications of relevant changes. Then, each application independently decides how to best adapt when notified. This adaptation occurs when the application adjust the fidelity[5] levels of the fetched data. Following a similar path, CARISMA is a middleware model that enables context-aware interactions between mobile applications. The middleware interacts with the underlying operating system and it is responsible for maintaining a representation of the execution context. Context could be internal resources (e.g. memory and battery power), external resources (e.g. bandwidth, network connection, location, etc.) or application-defined resources (e.g. user activity or mood). CARISMA provides an abstraction of the middleware as a customizable service provider, so that a service can be delivered in different ways (using different policies) when requested in different context. On the other hand, MobiPADS presents an event notification model to allow the middleware and applications to perform adaptation and reconfiguration of services in response to an environment where the context varies. Services (known as mobilets) are able to migrate between client and server hosts. MobiPADS supports dynamic adaptation to provide flexible configuration of resources to optimize the operations of mobile applications.

Regarding distribution policies, even though not in the OO paradigm, an interesting work is presented in GSpace [13], which implements a shared data space. This middleware monitors and dynamically adapts its distribution policies to the actual use of the data in the tuple space. The unit of distribution in a shared data space is called tuple, which is an ordered collection of type fields, each of them containing an actual value. Additionally, in GSpace tuples are typed, allowing the association of separate distribution policies with different tuple types. Making an analogy, we use the object as the basic unit of distribution, whose internal state can be seen as the data represented by a tuple. In Smalltalk, an object belongs to a particular class which can be mapped to the notion of type present in GSpace and assign the distribution policy at the class (type) level and change it at run-time. In addition, we provide the functionality to assign distribution policies in an object basis.

8 Concluding Remarks and Further Work

We have presented an architecture for developing context-aware applications that emphasizes in a clear separation of concerns. Also, by using and extending the dependency mechanism to connect different layers we have been able to avoid cluttering the application with rules or customization code that would result in applications that are difficult to maintain.

[5] Fidelity is the degree to which data presented at a client matches the reference copy in the server.

From the context modeling point of view, we have shown a behavior-oriented representation, where context is built from different context aspects. Those aspects provide the behavior that is contextually relevant in a given moment. This model, with the flexibility provided by a fully reflective environment as Smalltalk, provides the kind of dynamic adaptation that we consider context-aware applications need. We have also founded many things in common with the MVC architecture when we look at the way that sensing is separated from the context aspects and context aspects from services. This isn't surprising at all, since the reasons and the aims are basically the same: allow different layers of a system to evolve independently, without propagating changes to other layers. Finally, by extending the Opentalk framework we are able to choose between different strategies to distribute objects, making it possible to accommodate the system to the needs of mobile applications.

We are now working in the following issues:

- As mentioned in the introduction, we consider that next-generation context-aware applications will have such an extent that no single company or development group will be able to handle on its own. To cope with this issue, an integration platform is needed to allow software modules created by independent groups to interact seamlessly.
- Characterize object behavioral patterns, so that we can discover general rules for distributing objects with a given distribution policy.
- Adapt distribution policies to context. For example, a context aspect can be used to represent the network bandwidth, so that when it becomes lower than a certain threshold the distribution policy of predefined objects is changed (e.g. from forwarder to mirror to reduce the network traffic).
- Supporting intermittent network connections.
- We started a research track on Human-Computer Interaction, since we found that designing usable context-aware applications is not an easy task. The inherent limitations of mobile devices, such as small screens, tiny keyboards and lack of resources makes the design of usable GUIs rather difficult, so other non-graphical solutions must be explored, like audio or tactile UIs. Additionally, users of context-aware applications tend to be constantly moving and easily distracted, what makes usability a determining factor.

References

1. Gregory D. Abowd. 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, 1999. IEEE Computer Society Press.
2. Kent Beck and Ralph E. Johnson. Patterns Generate Architectures. In *ECOOP*, pages 139–149, 1994.
3. D. Bumer, D. Riehle, W. Siberski, and M. Wulf. Role Object Patterns, 1997.
4. Anind Kumar Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, 2000.
5. Paul Dourish. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1):19–30, 2004.

6. Andrés Fortier, Javier Muñoz, Vicente Pelechano, Gustavo Rossi, and Silvia Gordillo. Towards an Integration Platform for AmI: A Case Study, 2006. To be prsented in the "Workshop on Object Technology for Ambient Intelligence and Pervasive Computing", ECOOP 2006, 4/7/2006.
7. Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
8. Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. *Wirel. Netw.*, 8(2/3):187–197, 2002.
9. Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-Awareness on Mobile Devices - the Hydrogen Approach. In *HICSS*, page 292, 2003.
10. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.
11. U. Leonhardt. *Supporting Location-Awareness in Open Distributed Systems*. PhD thesis, Dept. of Computing, Imperial College, 1998.
12. Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287, New York, USA, 1997. ACM Press.
13. Giovanni Russello, Michel R. V. Chaudron, and Maarten van Steen. Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System. In *CoopIS/DOA/ODBASE (2)*, pages 1225–1242, 2004.
14. Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *CHI*, pages 434–441, 1999.
15. Albrecht Schmidt and Kristof Van Laerhoven. How to build smart appliances. *IEEE Personal Communications*, pages 66 – 71, 2001.
16. João Pedro Sousa and David Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *WICSA*, pages 29–43, 2002.
17. Visualworks Opentalk Developer's Guide - Part Number: P46-0135-05.