A Modular Architecture for Context Sensing

Julián Grigera *, Andrés Fortier * §, Gustavo Rossi * ‡, Silvia Gordillo * †
*LIFIA, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina.
email: {juliang, andres, gustavo, gordillo}@lifia.info.unlp.edu.ar
§DSIC, Universidad Politécnica de Valencia, Valencia, España.
‡CONICET
†CICPBA

***Abstract*— Due to the technological evolution, context-aware computing is slowly moving from dream to reality. These applications heavily rely in sensing the user's environment and abstracting this information to perform high-level adaptation. While technological issues have been mostly addressed, sensing software is usually built in a handcrafted way, which turns into building ad-hoc solutions for every new application. To solve this problem, we consider that context sensing should be regarded as a software engineering problem and not a simple implementation issue. In this paper we present a software architecture for dealing with context-sensing aspects in a high-level modular way. We show that by using this approach, evolution issues typical of mobile wireless software can be managed easily by replacing or composing software modules without compromising performance.**

I. Introduction

Developing context-aware application involves a set of software engineering problems, many of which have been often reported in the literature [1] and some of them solved [2], [3].

While high level issues, such as context modelling and context-aware adaptation have deserved much attention in the literature, context sensing remains as a low-level handcrafted task. The main consequence of this, is that the impact of software evolution tends to be harder to manage. For example, the introduction of new sensing devices might imply changing the acquisition policies (push vs. pull), adding new information formats (latitude/longitude vs. code bars or symbols) or changing existing low-level algorithms. If the application logic is coupled with the sensing mechanisms, as these modifications occur, the application is overwhelmed by changes. In lower levels of implementation, such changes usually impact in other application modules; for example we may need to build different mapping algorithms for abstracting the data gathered by a sensor into a higher-level format used by the context model. In other words, while sensed data is represented as strings, numbers or pairs, applications are built in terms of objects. Therefore, changes in the specific format provided by a sensing device pose a new challenge on data translation.

In this paper we show that, in order to simplify wireless software evolution in mobile context-aware applications, context sensing should be considered also a software engineering problem. To tackle this issue, we present a high-level approach realized by an application framework that allows flexible evolution of sensing features.

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

II. Sensors and System Evolution

If we think in terms of Weiser's vision [4], in a ubiquitous environment both the services provided to the user and the devices used to gather context information will be constantly changing. We should also notice that information about a specific context feature is usually gathered by different sensors. Consider location as an example of user's context. If the user is moving outdoors, location can be determined by a GPS receiver connected to his PDA. In the moment he enters a building, the system must switch from GPS to an indoor sensing technology, such as Bluetooth beacons [5]. This scenario shows that, depending on the user's situation, the information about his location can be obtained using different kinds of hardware sensors.

As a second example consider a user moving inside a building. To detect where the user is standing, we can use Bluetooth beacons. 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 a digital graffiti). In order to do this we can use an infrared beacon [6] to capture the user's location, this time with a finer granularity than the one provided by the Bluetooth beacon. As a result, location is sensed by two different devices at the same time, one giving more detailed information than the other. Additionally, the following requirements arise in systems that use sensing devices:

- As sensing mechanisms change, the policy needed to gather values may change as well.
- Sometimes filters must be placed in order to avoid some information to reach the application.
- Sensed values are generally low-level data, which must be abstracted in order to make sense to the application.

In this paper we present a flexible architecture that can be easily extended to accompany changes in the sensing mechanisms.

III. Architecture and Context Model: An Overview.

In order to provide a scalable architecture we need to establish a clear separation of concerns. To do so, we devised

21st International Conference on
Advanced Information Networking and Applications Workshops (AINAW'07)
0-7695-2847-3/07 $20.00 © 2007 IEEE

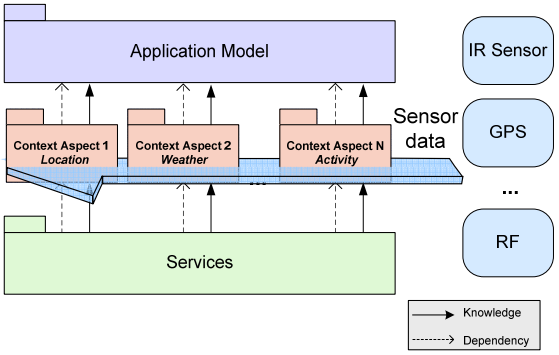

Fig. 1. The Architecture of a Context-Aware Application

a three layered architecture, making an important use of the dependency mechanism as presented in the Observer pattern [7]. A diagram of our architecture is presented in Figure 1.

The **Application Model** package contains the basic application behaviour, which shouldn't be cluttered with context dependent details; in this way the layer defines a set of classes specifying only application objects and their relationships. Thanks to this separation, the context model can change dynamically without impacting on the application.

In previous papers [8], [9] we argued that, due to its constant-changing nature, context shouldn't be formalized and modelled as a static object. Instead, we decided to model context as a collection of **Context Aspects**, each one modelling a specific feature of the current user context. The first implication of this model is that each context feature can be modelled and implemented in an independent fashion, making it easy to change the context model for a particular application. As a side effect, we also obtain two valuable features:

- Context aspects can be reused. For example, if we devise a context aspect that models the network bandwidth, we can reuse it in another application, since it is modelled as a feature independent of a concrete application.
- Context can change at run time. Our context model is based on the composition of different context aspects, allowing the system to redefine what is contextually relevant at any time by dynamically adding or removing such context aspects.

Finally, the **Services** layer reacts to a context change by adding or removing specific services. When the environment is set up, the available services are associated with specific providers, which will be available only when the user's context fulfills a set of requirements.

IV. Sensing Architecture

A common approach to model context-aware applications is to devise a layered architecture in which sensors represent the lowest level of abstraction. Although this kind of layered approach works well enough for prototypes and simple applications, we consider that it is not well suited for applications that evolve and increase their complexity over time. In the architecture we propose, the sensing mechanisms are decoupled from the context model, since both aspects will evolve independently from each other. This issue is seen in two important aspects of location-aware applications:

- On the context model side there are many ways of modelling the location aspect depending on the application's needs: we can use pure models (like geometric or symbolic)[10] or hybrid and enriched ones (like an extended symbolic model, which adds distances between nodes).
- On the sensor side, there are many ways to gather location data, independently from the location system used by the context model. Bats, Access Points and Bluetooth can be used to triangulate and calculate the position for a geometric indoor model, or they can be used as input for a symbolic model. In the same way, a GPS signal can be used for external geometric models, or mapped to a simple symbolic system.

For this reasons we claim that context must be modelled without considering the sensing mechanisms; the context model should only represent the user's context and not the way it is sensed. On the other hand, since sensors can be used in many different ways, they should only be concerned about their specific implementation details. In the rest of this section we will explain how we address these problems by showing how to connect these two layers.

A. A High-level View

To decouple context modelling and acquisition, our architecture adds a layer between the hardware that gets context information and the context model. The basic entities of this layer are the *sensing aspects*. A sensing aspect is an object that constantly watches over a set of sensors, and reacts when they produce new values, notifying the context model.

In Figure 2 we show a simplified class diagram of the sensing aspects layer, and how it relates to the other layers.

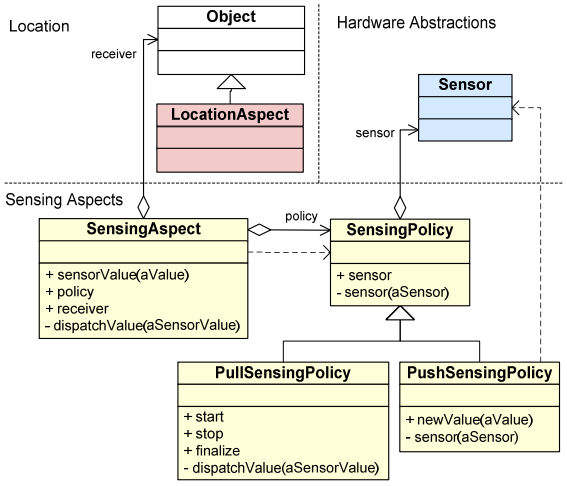

Fig. 2. Simplified Class Diagram of the Sensing Aspects Package

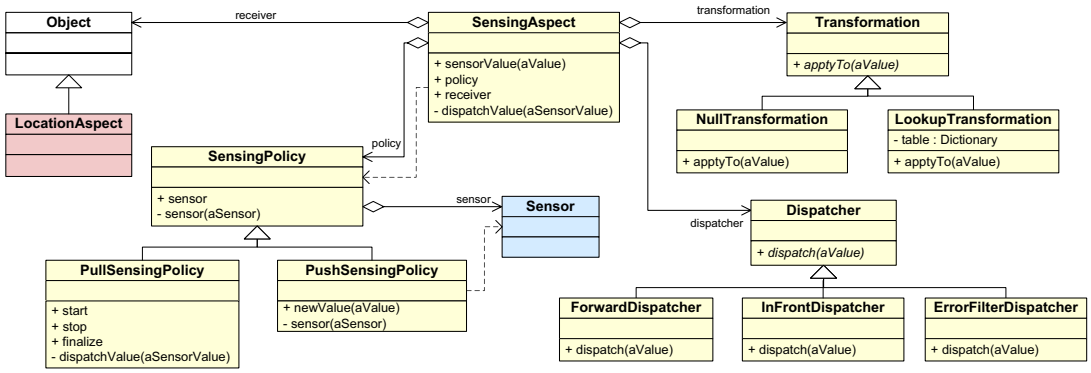

Fig. 3. Class Diagram of the Sensing Aspects Package

The **SensingAspect** class implements the functionality recently described. Each **SensingAspect** instance knows a set of sensors and a policy to acquire data from them. Additionally, when the sensing aspect is created, the programmer determines what message will be sent to the context model for it to update its information. Depending on the programming environment used, this behaviour can be configured by subclassing **SensingAspect** or using reflection. In our example the `location(newLocation)` message is sent to the location aspect of the user.

As shown in Figure 2, a sensing aspect uses a sensing policy in order to isolate the way the sensor works. By using the dependency mechanism, the sensing aspect only "knows" that, when the sensor gathers a new value, it will receive the `sensorValue(aValue)` message.

With only these simple mechanisms we have solved some of the problems mentioned in Section II:

- *The sensing mechanism is decoupled from the context aspect*. In the previous example, the location aspect has no knowledge of how the values are sensed. It only knows that when the user changes his location, it will receive the `location(aCoord)` message.
- *Multiple sensors can be used to gather information about the same context aspect*. Since the context aspect has no knowledge of the sensing mechanism, our approach scales to add sensing devices to the same aspect, all of them running concurrently. Consequently, sensing devices can be added and removed in runtime.
- *Data can be gathered by remote sensors*. Since our architecture is fully distributed, we can use proxies to allow sensing aspects for acquiring context data from remote sensors (i.e. not directly linked to the mobile device) seamlessly.

B. Dispatchers and Transformations

In some cases, we need to filter sensed data to lessen the burden of processing incoming signals we don't need. As an example, suppose an office environment with context-aware services, in which people tend to be sitting in front of their desks for long periods of time. To detect this, we choose to install infrared beacons in front of every desk. These beacons signal an ID representing the specific desk every 2 seconds, and the user's PDA is in charge of translating this ID into an object that represents that desk in the location model. Since users will leave their desks rather sporadically, and and we expect to receive the same ID for long periods of time, even tough the user is still in his desk, we can consider that the position hasn't changed at all. Still, the PDA will process every incoming signal every 2 seconds, dispatching the value to the context aspect. This burden may be eased if we only let the first signal come through the message dispatching mechanism and prevent the following ones from reaching the location aspect. Only when a different signal arrives, or a certain tolerance timeout ends, will the new signal be forwarded. To solve this issue we use a *dispatcher*, which is in charge of deciding, for a given signal, whether the dispatching process should continue or not. This concept is represented by the abstract class **Dispatcher**, which can be extended in order to provide sensor-dependent behaviour. Some example dispatchers, illustrated in Figure 3 are:

- **ForwardDispatcher**: Forwards every signal.
- **UniqueDispatcher**: As explained in the example, only sends a signal when it is first received.
- **ErrorFilterDispatch**: It is used to filter those signals whose noise level is higher than a given threshold.

Once we solved the dispatching issue, we need to address the data transformation procedure. In Section IV-A we presented a simple example to illustrate the basic sensing mechanism. Even though this example captures the fundamental idea, it assumes that data gathered by the sensors can be delivered to the context aspect without any transformations. In real applications this assumption doesn't hold and we need to map atomic values delivered by sensors to full fledged objects which will feed the context aspect. This transformation can take many shapes, ranging from simple table lookups to complex machine reasoning processes. The most common transformations we have found so far are:

- *Table mapping*. A typical example of table lookup is the mapping between sensor IDs and symbolic locations. This is widely used for indoor positioning.

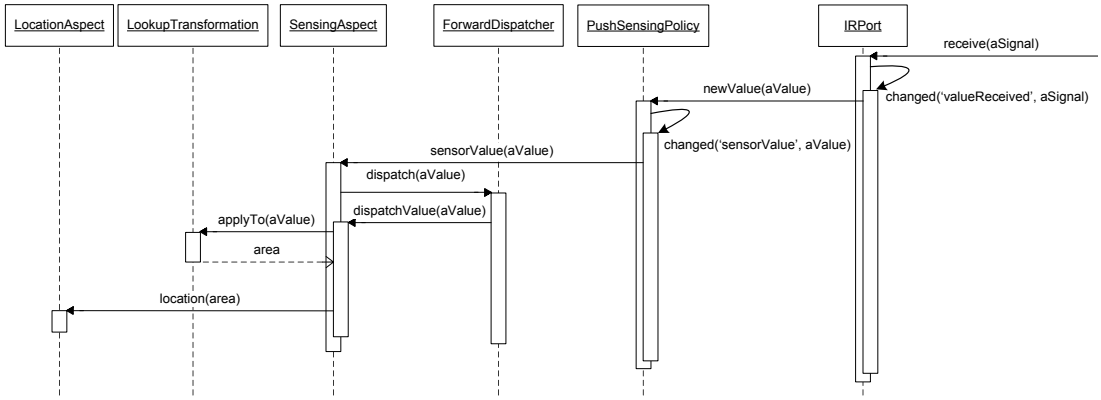

Fig. 4. Complete Dispatching Process

- *Coordinate system transformation*. This kind of transformation is used when data arrives in some coordinate system, and the location system used by the application works with a different one. As an example, we can get data from a GPS in a *(latitude, longitude)* pair, which needs to be mapped onto a specific geometric model used for positioning containers in a dock.
- *Situation abstraction*. In many cases data gathered by sensors is used to infer higher level information. For example, by examining information like location, schedules, the people in the surroundings, noise level and other ambient information, we can expect to make predictions about the user's current situation. Some related studies in this field can be found in [11] and [12].

To isolate these concerns, we modelled the transformation mechanisms in a separate class hierarchy, which is used to configure the sensing aspect. In Figure 3 we show the complete class diagram of the sensing layer.

To clarify the dispatching of a sensed value, we present a complete interaction diagram (see Figure 4). In this example the sensed value is an ID coming from an infrared beacon, which must be translated into a symbolic location by means of a simple lookup transformation.

C. Using Software Sensors

Through the paper we assumed that context data was gathered using hardware sensors, but, in many cases, this information can also be obtained through software modules. For instance, in order to infer the user's situation, we may need to inspect his schedule or see what kind of application he is running. Other examples of external sources of information are web services, which can be used to complement (and even replace) the information gathered by hardware sensors.

Our sensing architecture can be easily adapted to use software information. From the sensing point of view, these information producers can be seen as "software sensors", which provide data in the same way that hardware sensors do. Even the policies used in hardware sensors can be applied: services like e-mail or simple http requests use a pull policy, while there is a new trend to provide services through the internet using a push policy [13].

V. Adapting the Architecture through Sensors Evolution

In this section we will detail how to extend our sensing layer in order to add a new sensing technology. As an example scenario, consider a digital enhanced office, where users may be at their desks, attending to meetings or just walking around. Since we are going to provide location-based services we assume that we have a digital map of the office.

A. Adding Infrared Beacons

Suppose we want to use a set of IR beacons to identify different areas (meeting room, front door, etc). To integrate these sensors into the application we must follow 3 steps:

- Create a new sensing aspect.
- Select a policy to fetch sensor data.
- Create a transformation object to map atomic values into location objects.

The sensing aspect will coordinate the new functionality: it will receive the values coming from external sensors and take an action, which will be typically notifying the context change to its related context aspect. To configure this action we can create a subclass of **`SensingAspect`** and override the *`dispatchValue(aValue)`* method, which is invoked every time a new signal comes from the dispatcher. In our example, we created the **`LocationSensingAspect`** class and overrode *`dispatchValue(aValue)`* to convert the arriving value into a location object and then setting the new location to the location aspect by sending the message *`location(aLocation)`* to it.

```
class LocationSensingAspect
dispatchValue(value){
  newLocation = this.transformation.valueFor(value);
  this.locationAspect().location(newLocation);
}
```

Notice that the conversion is actually performed by a transformation object (as explained in IV-B).

Once the new sensing aspect has been created, we must select a fetching policy and a dispatching technique. These choices are closely related to both application needs and hardware features. Since the PDA's IR port will be constantly expecting for new beacon signals, a push policy will be adequate for this case. The following code shows the steps to set up the hardware abstraction and policy:

```
//locationAspect is the related context aspect
ir = IRPort.new();
policy = PushSensingPolicy.new(ir);
sa = LocationSensingAspect.new(policy,
locationAspect);
sa.setDispatcher(UniqueDispatcher.new(sa));
```

The **IRPort** class, included in the Hardware Abstractions package, implements the low-level functionality needed to manage the PDA's IR port. It is a subclass of the abstract class **Sensor**, which defines the abstract method *value()* (intended to return the last sensed value) and the behavior to notify changes to eventual observers. Notice that in the last line of code we set a dispatcher to the sensing aspect. This object, as we explained in Section IV-B, acts as a filter, deciding what signals will ultimately reach the context aspect. An **UniqueDispatcher** will only let in a signal that differs from the previous one, avoiding repeated data.

As we saw earlier on this example, the sensing aspect uses a transformation object to convert beacon signals to location objects. In this case, we can use a simple lookup table (already defined in the **LookupTransformation** class), and load the mappings from an XML file. Following with the previous example, we show how the table is created:

```
t = LookupTransformation.new(''mappings.xml'');
sa.setTransformation(t);

class LookupTransformation
  //constructor
  new(filename){
    mappings = (XMLParser.new()).parseFile(filename);
    for each m in mappings;
      signal = m.signal();
      location = digitalMap.locationFromId(m.value);
      this.addMapping(signal, location);
  }
```

B. Adding a GPS Device

So far we have only represented symbolic location, but sometimes it results useful to manage geometric location (i.e. coordinates systems). In order to add this capability to the application we will need a proper sensing technology to obtain geometric positioning. A GPS receiver attached to the PDA will serve for this purpose, although it will only be useful for outdoor positioning. The data this device supplies is a *(latitude, longitude)* tuple with the user's global coordinates. However, in our application we won't be using geometric positioning in terms of latitude and longitude, but simple *(x,y)* coordinates with respect to a local origin. This can be solved later with a simple transformation object.

The new sensing technique requires a low-level class (analogous to **IRPort**) to communicate with the GPS device; we will assume this is already implemented in the class **GPSDevice**. Just like we did before, we will create a new sensing aspect, this time using a pull policy to ask the device for values with a certain timeout. We can also use a dispatcher to skip repeated coordinate tuples (e.g. while sensing a user who is standing still). The following code shows the set up for the gps device with geometric position:

```
policy = PullSensingPolicy.new(GPSDevice.new());
sa = LocationSensingAspect.new(policy,
locationAspect);
sa.setDispatcher(UniqueDispatcher.new(sa));
```

The only issue left is the data transformation. We can create a new **Transformation** subclass and override the *valueFor(sensedValue)* message to convert the coordinates with a linear function. If we wanted to use the same symbolic location model keeping this technology, we only need a new transformation object that maps coordinates sets into location objects.

C. Adapting Web Services

Some useful context information is not easy to sense, either because it becomes too expensive or simply because it is not possible. Sometimes we can still get this information from others who actually perform such sensing and share the results publishing them into web services.

In our framework, there is a way to create web services query objects, and they can be easily incorporated into the application as software sensors (like we mentioned in IV-C). Consider as an example a weather report web service. We follow the same procedure from the previous examples, but now we will create a sensor class (analogous to **IRPort** or **GPSDevice**) to wrap the web service accessor. In this wrapper [7] class we will set up the parameters for the web service query; a weather report usually needs the global coordinates, which we can set fixed for static applications (like an office environment) or we can obtain from a geometric location sensing aspect when available. We can complete the setup with a pull policy with a suitable timeout, and an **UniqueDispatcher**.

As we have shown through these examples, setting up the sensing view of our architecture involves compounding objects and extending few classes. Having implemented the necessary low-level functionality for the new sensors, setting up the rest of the sensing layer involves little codification.

VI. Related Work

Problems regarding tight coupling between sensors and applications is not a new issue. In his PHD thesis [2], Dey

already refers to it as the source of major application redesign in regard to their experience in the Cyberguide project [6]. The Context Toolkit [14] has been developed to face these problems. In this framework, context information is obtained through external sensors that need to be integrated into the application with a higher level of abstraction. For this purpose, the Context Toolkit uses three different components, namely, *context widget*, *context aggregator* and *context interpreter*. The context widget component is used to get context information from sensors and retrieving it to the application, separating the context information from the acquisition process.

Another approach, based on layered architectures, has been used to provide flexible support for the acquisition of sensor-based context information. Following this style, Schmidt and Van Laerhoven [15] proposed a middleware architecture which is separated in four different layers: *sensors*, *cues*, *context* and *application*. Data obtained from sensors is processed by cues on the next layer. A cue represents a higher-level abstraction of sensed data and it is used to facilitate context recognition in the higher layer. Values generated by cues are buffered in a tuple space, which provides for inter-layer communication between the cues layer and the context layer. The context layer can read these values and infer the current context.

As we have seen in this approach, the use of middleware architectures helps decoupling the sensing hardware from context abstractions. In the architecture depicted above, it is feasible to switch sensors on the lowest layer without having a mayor impact on the cues processing layer. The tuple space between the cues and context layers helps to separate sensed data and context information, providing also a means to distribute information in networked environments.

VII. Concluding Remarks and Further Work

In this paper we have presented a software architecture for dealing with context sensing. We showed that by using simple state of the art software composition mechanisms it is possible to face usual problems that developers have to cope with when sensing technology changes or evolves. We have explained how to modularize those properties which are idiosyncratic of sensing devices and decouple them from application concerns. Format transformations, acquisition policies and signal processing have been objectified and thus made easily interchangeable according to application needs or technological requirements, even at runtime. Also, by clearly decoupling the sensor notion from the sensing aspects we are able to replace hardware sensors with any information provider, like rss or web services.

We are currently researching in three different areas in order to improve the sensing layer. At the lowest level, we are developing an approach for building abstract specifications for sensors. This would allow us to program against a sensor specification without caring about specific low-level details. As an example, we can have a port specification that states that the sensor receives a five digit number, making the system independent of whether we will use the infrared or Bluetooth port of our PDA.

We also are developing a distributed publish-subscribe system to connect sensors residing in different hosts. This subsystem will handle data gathered by sensors and meta-data related to the sensors themselves.

The third (and most challenging) area of research is the transformation step, which now must be done in an ad-hoc manner for every sensor. We are trying to tackle this problem by modelling typical kinds of transformations, in order to reuse them in different applications and by providing a higher level syntax to express transformations, so that non-programmers can express them in a natural way. Our final goal is to be able to design a graphical editor that allows us to build this kind of systems by connecting different modules based on sensor specification, predefined filters and required policies.

References

[1] G. D. Abowd, "Software engineering issues for ubiquitous computing," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 75–84.

[2] A. K. Dey, "Providing architectural support for building context-aware applications," Ph.D. dissertation, Georgia Institute of Technology, 2000.

[3] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang, "A middleware for context-aware mobile services," *IEEE Vehicular Technology Conference*, 2004.

[4] M. Weiser and J. S. Brown, "The coming age of calm technolgy," *Beyond calculation: the next fifty years*, pp. 75–85, 1997.

[5] A. Huang and L. Rudolph, "A privacy conscious bluetooth infrastructure for location aware computing," http://people.csail.mit.edu/albert/pubs/2004-albert-infrastructure-for-location-aware-computing.pdf.

[6] G. Abowd, C. Atkenson, J. Hong, S. Long, R. Kooper, and M. Pinkerton, "Cyberguide: a mobile context-aware tour guide." *Wirel. Netw.*, vol. 3, no. 5, pp. 421–433, 1997.

[7] E. Gamma, R. Helm, and R. Johnson, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] A. Fortier, G. Rossi, and S. Gordillo, "Decoupling design concerns in location-aware services," in *Mobile Information Systems II*, 2005, pp. 187–202.

[9] G. Rossi, S. E. Gordillo, and A. Fortier, "Seamless engineering of location-aware services." in *OTM Workshops*, 2005, pp. 176–185.

[10] U. Leonhardt, "Supporting location-awareness in open distributed systems," Ph.D. dissertation, Dept. of Computing, Imperial College, 1998. [Online]. Available: citeseer.ist.psu.edu/article/leonhardt98supporting.html

[11] E. Mynatt and J. Tullio, "Inferring calendar event attendance," in *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces*. New York, NY, USA: ACM Press, 2001, pp. 121–128.

[12] D. J. Patterson, L. Liao, D. Fox, and H. A. Kautz, "Inferring high-level behavior from low-level sensors." in *Ubicomp*, 2003, pp. 73–89.

[13] M. Hauswirth, "A reference architecture for push systems." technical Report.

[14] D. Salber, A. K. Dey, and G. D. Abowd, "The context toolkit: Aiding the development of context-enabled applications." in *CHI*, 1999, pp. 434–441.

[15] A. Schmidt and K. V. Laerhoven, "How to build smart appliances," *IEEE Personal Communications*, pp. 66 – 71, 2001.