The Journal of Systems and Software 83 (2010) 915–936

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Dealing with variability in context-aware mobile software

Andrés Fortier *,[1], Gustavo Rossi [1], Silvia E. Gordillo [2], Cecilia Challiol [1]

LIFIA, Facultad de Informática, UNLP, La Plata, Argentina

ARTICLE INFO

Article history:
Received 2 December 2008
Received in revised form 10 August 2009
Accepted 1 November 2009
Available online 4 November 2009

Keywords:
Context-awareness
Mobile software
Architecture
Software variability

ABSTRACT

Mobile context-aware software pose a set of challenging requirements to developers as these applications exhibit novel features, such as handling varied sensing devices and dynamically adapting to the user's context (e.g. his or her location), and evolve quickly according to technological advances.

In this paper, we discuss how to handle variability both across different domains and during the evolution of a single application. We present a set of design structures for solving different problems related with mobility (such as location sensing, behaviour adaptation, etc.), together with the design rationale underlying them, and show how these sound micro-architectural constructs impact on variability. Our presentation is illustrated with case studies in different domains.

1. Introduction

Mobile, context-aware software pose new and hard challenges to developers. Some of these challenges arise from the specific requirements of this kind of software, in which we need to process implicit input data from non-accurate devices (such as sensors), and make critical decisions based on that data (e.g. adapting the application's behaviour to the user's location). Some other constraints such as security and privacy are also difficult to fulfil, as many times private data such as location and preferences must be shared by different applications.

Mobility is a common feature of different kind of applications, encompassing diverse application domains. Examples of "standard" (i.e. initially context-agnostic) applications that can be upgraded into mobile applications are health-caring systems (Bricon-Souf and Newman, 2007), guided tours (Cheverst et al., 2002; Abowd et al., 1997) and transit information (Google Mobile Maps). On the other hand, any mobile application is susceptible to be enhanced with context-aware behaviour, such as providing location-based services (Hodes and Katz, 1999; Rao and Minakakis, 2003) or dynamic resource management (Sousa and Garlan, 2002). As a result, mobile context-aware applications not only present a set of complexities related to their underlying application domains, but also those related to the kind of context-aware adaptation desired, namely its *adaptation domain* (e.g. a guided tour that recommends different routes according to the user preferences and location). As we show throughout the paper, a clear distinction between the *application domain* and the *adaptation domain* is useful in order to maximize flexibility and reuse.

Understanding how to improve flexibility is important, because there is a growing trend towards extending "old" applications, not only to allow mobile access to information and services, but also to adapt "legacy" behaviours to the user's actual location (or in general, his or her context), which implies, in most cases, much more than support for mobility, requiring some kind of "wrapping" of such existing behaviours.

Additionally, as wisely indicated some years ago by Abowd (1999), the evolution of this software adds new difficulties as "Ubiquitous applications evolve organically. Even though they begin with a motivating application, it is often not clear up front the best way for the application to serve its intended user community. The best approach is often to prototype a solution rapidly. . .and solicit feedback from the user population. . . .This result in the need to modify the application".

Surprisingly, while many of these issues have been successfully tackled and reported in the literature (Harter et al., 2002; Schmidt et al., 1999; Strang and Popien, 2004), there has been little emphasis on how to systematically support variability and evolution in mobile and ubiquitous software. One possible reason for this is that many applications are built "just" as prototypes, and the community has not still faced the problem of maintaining legacy mobile software.

In this paper we present a software architecture aimed at serving as the foundation for the construction of applications and frameworks, targeted at different product families of mobile,

* Corresponding author.
E-mail addresses: andres@lifia.info.unlp.edu.ar (A. Fortier), gustavo@lifia.info.unlp.edu.ar (G. Rossi), gordillo@lifia.info.unlp.edu.ar (S.E. Gordillo), ceciliac@lifia.info.unlp.edu.ar (C. Challiol).
[1] Also CONICET.
[2] Also CICPBA.

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.11.002

context-aware software (see for example Myllymäki et al., 2002); the architecture provides a set of fine-grained abstractions which ensure a high level of flexibility in two different axes: application evolution and support for different adaptation and application domains. The main contributions of the paper are the following:

- We identify and make explicit the most relevant variant features and their associated variations points in the design of mobile context-aware software for different domains.
- We present a set of micro-architectures which ensure modularity by decoupling different application concerns. By doing so we can later instantiate different products by designing and implementing its unique features.
- We show how these constructs are integrated in a sound architecture and provide some proof of concepts both on application development and variability support.

The structure of the paper is the following: in Section 2 we will survey some relevant work in the field of context-aware applications, frameworks and architectures and briefly review the work made in the area of product-lines. In Section 3 we will characterize variability and evolution on mobile context-aware software, describing the main variant features and their associated requirements. In Section 4 we will describe in detail the variant features together with their associated requirements. In Section 5 we will give an overview of the whole architectural approach and in Section 6 we will present the most important micro-architectural abstractions. In Section 7 we will illustrate our ideas with two concrete examples, including a memory-aid application and a framework for location-based services (LBSs). In Section 8 we will conclude the paper and present some further work. Though our ideas are shaped in the broad field of context-aware software systems, we focus our discussion on those issues which are related to mobile software (i.e. when context refers to the user's location).

2. Related work

Weiser's (1999) pioneer work at Xerox Parc was followed by Schilit's (1995) system architecture and the Stick-e Notes (Pascoe, 1997) framework. These first approaches to build context-aware systems helped to understand many of the challenges we are facing today and some interesting ways of tackling them. In that regard, an important design decision in Schilit's approach was to decouple those independently-changing blocks of the system to support scalability, which is a way of anticipating future changes. Also, Schilit emphasized that sensing devices should be abstracted, so that the application logic does not have to get involved with the burden of connecting to hardware devices and sensing information, which is also a way of thinking of variability at the design and implementation level. However, these applications and prototypes were aimed at particular, restricted domains and not conceived to aid in the development of product families.

The next milestone in the history of context-aware architectures was Dey's Context Toolkit (2000), which is a pioneer framework aimed at building distributed context-aware applications, based on a peer-to-peer architecture. The framework is built around the notion of context widgets, which are used to access context data (gathered by physical or logical sensors) using the widget metaphor. The main task of a context widget is to encapsulate the communication protocol and internal state of a sensor to simplify the interaction with other software components. Context interpreters are used at a later stage to abstract the widget's information, which may also be composed with context aggregators. The toolkit is finally completed with services (which implement the required behaviour) and discoverers, which know the state of the application in terms of available components (i.e. widgets, interpreters, aggregators and services). Even though Dey's toolkit was a breakthrough in terms of creating a generic architecture for context-aware systems, we consider that there are two main issues that should be addressed:

- There is no explicit notion of a context model. The designer is in charge of modelling context based on the widgets and aggregators available. This has the advantage of being extremely flexible, but that flexibility comes with a price. First, we cannot think in general terms about context, since there is no structure to reason about. In order to define features and variation points we need explicit models, even basic ones. Second, even though Dey stresses the separation between sensor data and context information, deriving a context model from the widgets' information can be equally harmful. This problem is not new and a parallelism can be made with the combination of RAD tools, inexperienced programmers and a lack of designing guidelines, where applications are written by placing widgets in a window and then writing their action handlers (e.g. *onClick* actions). This style of programming has yielded an uncountable set of programs where there is no explicit domain model, but a collection of procedures scattered through the application and intertwined with GUI logic[3]. As we will show later, we propose taking the inverse approach: first designing the context model (based on simple guidelines and an existing architecture) and later decide how it will be connected to the sensors.
- There is no architectural support for separating application logic from adaptation logic. A service in Dey's toolkit is in charge of performing actions based on context changes that can encompass both adaptation and application logic. As we will describe later, we stress the difference of the base application domain (e.g. a university information system) and the adaptation domain (e.g. providing the students location-based services in the campus) to improve scalability.

In a recent work on mobile services by Pederson et al. (2008) we have found many interesting similarities with our approach, especially in the context model. One of their first statements is that different applications cater for different aspects of the user's context, which means that a unique context model for different applications is not a viable solution. They also stress the sense that the context may be on a server, on a client or distributed between both of them. We clearly share this view and this is one of the main reasons why our design splits context in a set of small-grained features. However, we consider that these approaches have a serious drawback since all their context aspects are grouped in a general context model without any semantic cohesion (e.g. we can find a user's location and the amount of students in a university room as part of the same model).

Another recent approach, but at the language construct level, is Context-oriented Programming (COP) (Hirschfeld et al., 2008), which uses the notion of layers as their main language construction. These layers define classes and behaviour and can be activated and deactivated while a program is running, effectively changing a system's behaviour in run-time. However, since it is not part of its aim, COP does not provide higher level constructs to model context-aware applications. Thus, while the layer concept and implementation are useful as a language feature, we consider that a higher level support is needed for systematically building context-aware applications. We believe that

[3] Of course we are not against GUI builders or RAD tools. However we do claim that many times the people get the false feeling that there is no need for a well thought domain model and that mistake is paid later when the system needs to evolve or incorporate new requirements.

the context model and the idea of separating application and adaptation domains are central to the process of building this kind of applications. However we acknowledge that, once identified the main features of a product line for a set of context-aware applications, using COP may greatly help to solve run-time variation points.

An interesting idea of constructing a UbiComp application by using a set of well defined building blocks (which is conceptually related to our approach) is presented in (Modahl et al., 2004). In the article, the authors make an analogy between building a UbiComp application and a telephone patch board, where the software layer should define a set of main abstractions which are later connected to build new systems. In their paper the authors classify separate subsystems in five different groups, like Registration and Discovery, Service and Subscription, Data Storage and Streaming, etc. The bottom line of the article is that UbiComp applications typically use one or more components to fill these groups, and that having a widely accepted taxonomy and standard interfaces would improve the engineering of UbiComp systems. Even tough we agree in the general idea of defining building blocks, we do not think that it is possible at such high level of granularity (i.e. subsystems) and this is why our approach uses small-grained building blocks to solve concrete variation points.

In the variability area, we based our research on works like (Coplien et al., 1998), where the author specifies common and variable issues according to an object-oriented approach, showing a set of mechanisms to solve them. Most of the terminology and concepts we use are presented in Van Gurp et al. (2001). In Bachmann and Bass (2001) a classification of variability is described together with the types of variation which are similar to the three recurring pattern presented in Van Gurp et al. (2001). An interesting taxonomy of variability realization techniques is described in Svahnberg et al. (2005), where the authors show how to implement different types of variation points.

The area of product-family and context-aware architectures is not a prolific one and we feel that there is yet a lot to be done. In the work presented by Salifu et al. (2006) they propose to extend the variability mechanisms of product lines to accommodate context-aware specifics, but at a very coarse-grained level. Apel and Böhm (2005) present a product-line approach for ubiquitous middleware. Even though the paper uses ubiquitous computing as a motivation, the article only describes a configurable communication layer to connect clients and servers, so that messages can be sent to remote objects, showing no architecture or model that can be related to our work (i.e. context models, dynamic behaviour adaptation, etc.). Finally, Fernandes et al. (2008) propose a notation called UbiFEX, for representing context information. In some aspects we find their approach to be related to ours, since they think in terms of entities and the context properties of each entity (in contrast to having a generic context model) and argue for specifying context feature separately from the application features. This last separation can be roughly correlated with our decision of separating the application model from the context model. Finally, the interaction between the context and the application is achieved by rules. Even tough we found some conceptual commonalities with our approach, their work stops at the notation level and provides no clues on how this notation can be mapped to a specific architecture. Also, we consider that managing context adaptation by using rules works fine for simple prototypes but gets unmanageable for complex applications.

The aim of this article is to solve a set of issues that, to our knowledge, have not been worked out yet. In particular we will:

- Present a product-line characterization of context-aware systems that describes the main variability issues found in these applications.
- Define the variability features and variation points associated to most context-aware applications
- Present an architecture, in the form of a set of small-grained building blocks that can be used to build mobile context-aware applications avoiding most of the problems we have mentioned in this section.
- Show how the variation points can be addressed solved with the proposed architecture in real examples.

3. Variability and evolution in mobile, context-aware applications

In order to define what we consider the critical aspects of any mobile context-aware application we will briefly review some concepts related to variability in software product lines. The definition given in Svahnberg et al. (2005) states that "*Software variability is the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context*". Van Gurp et al. (2001) defines variability as "*...the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability*". In short, variability is an outstanding issue which should be considered during the software building process to guarantee that change and evolution can be successfully managed. As we will see later, this is particularly important in mobile context-aware applications, since we must be ready to deal with a myriad of options (e.g. incorporating new services) and constant changes (e.g. new sensing technology).

To successfully manage variability we must achieve a balance between software flexibility and realistic, concrete needs; whereas a system that can accommodate any new requirement by just replacing a few components would be ideal, experience has shown that this is hardly feasible. On the other hand, constraining a system to the initial requirements without thinking about evolution means that greater costs will be paid later. Thus, we must find a way of explicitly stating which parts of a software product line are open for change and which are assumed to remain stable, at least for some period of time. By doing so designers can concentrate on creating flexible architectures for a family of products, constraining themselves to concrete requirements but without compromising evolution.

In the approach presented in Van Gurp et al. (2001) and later extended in Svahnberg et al. (2005) variability is made explicit by identifying variant features. These features can be of different types (mandatory, optional, variant or external) and follows a lifecycle based on the feature's state (defined, implicit, introduced, added and bounded). In this work we will concentrate on variant features. Variant features are represented in a software system by means of variation points, which can in turn appear in a specific state (implicit, designed or bound) and which can be open or closed for adding new variants. Open variation points are associated with set of options where new variants can be added while closed variation points have an immutable set of options. Finally, variation points are generally characterized according to the variant pattern used: in the *single variant* case only one variant is chosen from a set of options. The *optional variant* pattern states that there is only one variant, but using it is optional. Finally, in the *multiple parallel variants* case the set of variants to choose which variant to use is re-evaluated each time the variation point is accessed.

At the implementation level we would like to have an architectural infrastructure that provides the most common abstractions (such as sensors) and behaviours (such as different types of

adaptation) of this kind of software, leaving the application designer with the task of developing "only" the application's specific functionality. In other words, our aim is two-folded:

- Design an architecture, which identifies the main variant features and its associated variation points.
- When possible, provide the realization of the most common variants, so that standard behaviours can be achieved by selecting existing components. When necessary we leave the variation points open so that new variants can be added.

To understand the variability problems in context-aware applications, we consider very important to establish a distinction between application domains and adaptation domains. Even though the border between these domains is blurred most of the times (especially when the application is devised from scratch), it is clearer when enhancing an existing application. As an example, consider a desktop health-caring application for a hospital that keeps track of patients' records, analysis, current status, etc. Suppose now that we want to extend this application so that it can be used from a mobile device (e.g. a tablet PC) that adds behaviour depending on the doctor's location (i.e. provides Location Based Services (LBSs)) in case of an emergency (e.g. automatically showing the patients information on the tablet PC). A different scenario (in terms of adaptation) would arise if we want to enhance the hospital by providing ambient intelligence (EC-ISTAG, 2001), such as adapting the patients' environment according to their actual status and, at the same time, warning the nurses or doctors in case the status changes abruptly.

What we get in both scenarios (Ambient Intelligence and LBSs) is a mixture of domains: on one hand we have the health-caring domain (i.e. the application domain) while on the other we have the LBSs or ambient intelligence domains (i.e. the adaptation domains). It is evident that both types of domains (application and adaptation) comprise different kinds of abstractions and, as we will show later, one of our main claims is that these two domains should be engineered separately.

We next characterize the main variability issues we have found in mobile context-aware applications:

- **Dynamic context model**. Since context-aware systems are developed in very dissimilar domains, it is almost certain that each application will require a different context model. To make things more complex, certain applications may need to change the context model in run-time (e.g. adding new types of context data according to the available sensors). Thus, we consider the context model to be an important variant feature to take into account when developing context-aware applications. Also, this variant feature is interesting since the associated variation points appear at two different stages: at design time, when the possible components that will compose the context model must be defined (e.g. if our context model will encompass location and activity, we must design how these pieces of context will be modelled). At this stage each context aspect follows the *single variant* pattern. However, this variant feature also changes at run-time, when we should decide which context aspects are active (e.g. we may not have information about the user's location and thus the location part of the context may be temporally suppressed).
- **Sensor support**. A critical aspect of a context-aware system is the ability to adapt its behaviour to the current context. To do so, context data must be automatically acquired from external sources, such as hardware sensors or software sensors (e.g. web-services). Also sensors may be added (or removed) after the application has been installed in the target device. As in the previous case, this variant feature maps in many variation points at the architectural level. Those variation points are open, since new sensing devices can be added as technology evolves. The binding is done at run-time and may change while the application is running, thus a *multiple parallel variants* pattern is followed.
- **Domain-specific adaptation**. In a very simplified way, a context-aware application takes into account contextual information in order to adapt its behaviour. Even though this behaviour is generally seen as intertwined with the application's underlying themes, it has its own particular domain. As we mentioned before, a health-caring system has its own domain model, which includes tracking patients medical histories, blood tests, nurses schedules, etc. If we now want to provide context-dependent behaviour to doctors carrying a mobile device (e.g. displaying the patients record when the doctor approaches the patient's bed) we need to address the LBSs domain, which has a completely different problematic (such as sensor management, location models, etc). A different set of problems arise in the Ambient Intelligence domain (e.g. dealing with actuators, issuing alarms, etc) and even some of them might be shared by both domains (i.e. LBSs and AmI). Thus a main variability issue in context-aware applications is the type of adaptation that the application will provide. At design time the associated variation point is open, since is the time where the context-aware application-specific functionality is built. At run-time however the user may decide to temporally suspend it (e.g. cancelling work-related notifications when the user is at launch), yielding an *optional variant* that can be enabled or disabled a run-time.

In the following section we will present the main requirements associated with each variant feature and their variation points. We will then outline the main packages of our architecture and describe them in detail, based on their requirements. At the end of the paper we will review these main variability issues and see how the proposed approach deals with them.

4. Identifying variability in context-aware applications

In this section we will explain the main places where variability is required, based on the variant features presented in the previous section. To complement the information of each item we will also explicit its associated requirements. As we briefly did in the previous section, to characterize each variation point we will base our description on the articles presented by Van Gurp et al. (2001) and Svahnberg et al. (2005).

4.1. Dynamic context models

The context model is a central part of any context-aware application and efforts has been put forward in different areas to create context models that are both expressive and flexible to accommodate a myriad of requirements. In our experience with context-aware applications we have found the following mandatory requirements that a context model should comply:

- If required, the context model should be shared among applications, independently of the adaptation (or application) domain. As an example, suppose that we have modelled a complex location application for providing LBSs, both for indoor and outdoor settings. Sharing this infrastructure and the context information with another application (e.g. memory-aid application) should be straightforward.
- On top of the previous requirement, each application may have its own set of context requirements. Thus, even tough the context schema is normally shared between applications of the

same adaptation domain (e.g. LBSs applications need to know the user's location), each application may require its own particular contextual information (e.g. the user activity, the traffic state or the current weather). This means that the context model may be extended in a particular way to suite each application needs.
- Context is not a single entity. Even though we generally speak about "the context", we are omitting *whose* context we are talking about. Treating the context as a whole single entity, yields models that are difficult to understand and modify, since different aspects of the context belonging to different entities may have different lifetimes and engineering requirements. As an example, consider the user's location and the available bandwidth of a network connection. While it is clear that the location is part of the user's context, it is arguable to classify the bandwidth as part of the user's context. A much more clear (and flexible) model would emerge from representing the bandwidth as part of the application context.
- Context is entity-based. By extending the previous requirement, we may face the scenario where different entities (objects) of the same kind (class) need different context models. As an example consider the different rooms in a smart home: in a room we may have a sound sensor and thus we can consider the noise level as part of the context of the room. However, other rooms may not have this facility and so we cannot consider the noise level as part of their context. Thus we consider that the context model should be engineered and applied in an instance basis, not as a whole entity or even as a class-based concept.
- The Context schema can be modified at run-time. Context-aware applications are expected to accompany the user for long periods of time, even an entire day. In this scenario, certain context features may be available or not (e.g. the user's location may be available as he or she moves outdoors since the device has a GPS, but may not be available indoors). Thus, the context model should support changing its shape in run-time.

From these requirements we can easily see that the context model should be flexible and manageable in small-grained pieces, representing the context of individual application entities (such as the user or a room in a smart home). We must also remark that the context model should be defined in an instance (and not class) basis, since different objects of the same class might need to consider different context aspects. At this stage we are ready to express the main guidelines of our context model, which we will use to further specify the associated variation points:

- Context is defined in an instance basis.
- The context of an object is split in a set of *context features*[4], each one focusing on a specific domain (e.g. location, activity, weather, etc.).
- The context shape of an object is defined by the context features associated to it. This shape can change in run-time.

With these guidelines in mind we can now describe the context model in terms of variant features: when an application is designed we must decide which objects will be enhanced with context information. For those objects we must model the associated context features and decide if (and how) they will be activated/deactivated in run-time. Thus, we consider each context feature as a variation point, which is closed in the implementation phase, but that behaves as an optional variant in run-time since it can be disabled and re-enabled at any time.

4.2. Legacy bridge

The context model guidelines presented earlier do not suppose any existing application or previous domain model. While this assumption might be correct when a mobile context-aware application is developed from scratch, we may also be in the situation of extending an application with an already existing application model. In this scenario, the existing application is called the *base* application and its associated model the *base* model.

As we previously stated, we consider that the application and adaptation domain should be engineered as separately as possible; the context model, being fundamental for the adaptation components, is not an exception. Thus, on top of the requirements enumerated in the previous section we add the following ones:

- The context model should be transparent to the base application. In other words, the fact that we are adding context information to existing application objects should not imply changing the application model, re-coding part of it or changing existing test cases.
- It should be possible to reify, in the context model, information available in the base model. In the same way that languages reify certain implicit concepts (e.g. continuations in Scheme or classes as objects in Smalltalk) we would like to be able to use the information present in the application model and treat it as part of the context model.

A way of characterizing these requirements is to think of a closed variation following a single variant pattern. In the variant set two "types" of context models are available, one that assumes that there is no previous model and other that takes an existing model into account. The decision of which type of context model will be used is made at design time.

We can now see how this variant feature relates to the one defined in the previous section: the first step when building a context-aware application is to decide, at design time, if we are extending an existing application or creating one from scratch. After that step the set of required context features are designed, implemented and tested, defining the possible context shapes for each domain object. Finally at run-time different combinations of context features may arise depending on whether each context feature is present or not.

4.3. Sensor support

A key aspect of most context-aware applications is how context information is gathered from the environment and fed into the context model. The most common way of achieving this is by using sensors, either physical (e.g. a GPS) or logical (e.g. a weather web service). Thus, sensor management is a delicate issue in context-aware systems and becomes more delicate when using mobile devices. In our experiences we found the following requirements to be critical when handling sensing devices:

- Support for embracing new technology. Technology evolves fast, constantly introducing new hardware, and sensing devices are not an exception. Indoor location systems are a clear example of applying many different approaches and technologies to solve a specific problem (Davies et al., 2001; Priyantha et al., 2000; Feldmann, 2003; Patel et al., 2006; Hightower and Borriello,

[4] Unfortunately the word "feature" clashes with the variability concept of feature. To avoid misunderstandings we will always refer to "context feature" when talking abut the context model and to "variant feature" (or just "feature") when talking about variability features.

2001). Thus, an architecture for mobile context-aware applications should be able to incorporate new technologies in a fast and easy way.
- Support for every sensing stage. The sensing process involves many stages, like configuring sensing policy (push vs. pull), filtering unwanted signals (e.g. because their quality is below a certain threshold), transforming low-level data to semantically rich objects and feeding the acquired information into its corresponding model (e.g. the location of an object). Thus, the steps involved in the sensing process must be decoupled so that a change in one of them does not impact on another. In this scenario, modularization of each stage is extremely important.
- Run-time support. Even though some sensors can be carried in the mobile device (e.g. GPS), other sensors are placed in the environment and accessed wirelessly (e.g. a Bluetooth-enabled weather station). Thus, support for run-time discovery and reconfiguration is a very important feature.

Similarly to the case of the dynamic context model we can see that this variant feature has two stages: at design time we must define what types of sensing devices will be supported and how the information gathered by these devices will be mapped to the context model. At run-time, depending on the active context features, the configured sensing devices will be activated. As a matter of fact we can go one step further and even use alternate sensing devices for the same context information. At design time we can specify many sensors for gathering a specific context feature and sort them using some predefined order (e.g. by the quality of service). At run-time the system searches for the first available sensor according to the given order. If that sensor later becomes unavailable or a new sensor can be used, the search is performed again. As we will show in Section 6 the sensing stage comprises many low-level variation points associated with this variant feature.

4.4. Domain-specific adaptation

Domain-specific adaptation focuses on the behaviour used to adapt an application to the current context (e.g. the user's context). Even though it is sometimes difficult to separate the adaptation domain from the application domain, doing so greatly increases the architecture (and the application) flexibility. Also, by doing so we can encapsulate the adaptation-related behaviour and think of it as a new variant feature. As with previous sections, we will now define the main requirements for the domain specific adaptation:

- Different domain-specific adaptation should coexist. As a general rule, adaptations should be modelled and developed without taking care of other adaptations present in the system at the same time. Of course, if resource sharing is required (e.g. accessing a database) some kind of synchronization will be required, but this should be the exception and not the rule.
- Domain-specific adaptation should be completely decoupled from the actual application domain. Whether we are building an application from scratch, or improving an existing one to support context-aware features, the application model should be independent of the proposed adaptation, since the forces that drive the changes in each case are orthogonal.

From the point of view of the context-aware functionality, the domain-specific adaptation is the core behaviour and must be specified in design time. Thus, this variant feature must be closed at that stage and remain unaltered through the life of the application. However, as we will show in Section 6, some sub-components may be suppressed at run-time which is why a variation point is created for each of them.

5. Our architectural approach: a coarse-grained view

As we stated in the previous section, mobile context-aware applications exhibit many variation points where an application can be customized to satisfy a set of particular requirements. In a first approach, one could think that a carefully designed object-oriented framework would be enough to manage these variation points. However, our experience shows that this endeavour should not be dealt with naively. Frameworks capture the knowledge and good design decisions that emerge as a result of building many similar applications in a particular domain (Fayad et al., 1999); however, context-aware applications can be built on a myriad of domains, at both application and adaptation level (e.g. LBSs can be applied to tourist or health-caring applications, whereas on top of a tourist application we can build LBSs and memory-aid functionality). As a result of this, it is not clear *which* would be the underlying domain abstracted by the framework. As an alternative, we could think of composing two frameworks (one for each domain); however, this approach presents many disadvantages (as reported by Mattsson and Bosch (1997)). Thus, instead of heading towards a one-for-all framework, our aim is to build a more abstract context-aware architecture which "only" supports the concepts which crosscut all domains, but that can be configured to be the infrastructure of a domain-specific framework. The application-dependent parts, which are treated as variant features at the design level were described in Section 4 and will be further specified as variation points in Section 6.

In our research we have undergone a two-phase process: the first one consisted in defining a set of micro-architectural abstractions that are present in most context-aware applications (across domains). These abstractions are domain independent (both in terms of the application and adaptation) and as a result, reusable in many different situations. In other words, they represent the building blocks that help in the construction of any context-aware system, independently of its application or adaptation domain.

With these basic abstractions defined we can move to the second phase, which is developing concrete applications in a specific adaptation domain. From the architectural point of view, this second phase involves the binding of the variation points with a specific variant. It is interesting to note that this phase can be engineered as any "standard" application, where usual techniques and methodologies are employed (Fayad et al., 1999). In particular, after building many applications in the same adaptation domain (e.g. LBSs) we can derive a concrete framework. As a result we get a flexible platform that eases the development of applications and frameworks. In Fig. 1 we show a coarse-grained schema of the platform's architecture.

The building blocks of our architecture have well defined roles and can be used and extended in different ways. We next outline the four main packages shown in Fig. 1:

- **Core abstractions**. This package encompasses the basic structures to handle context and to coordinate the communication across different packages. Even though these abstractions can be extended by specialization, they are used most of the time as black-boxes, i.e. by means of composition.
- **Legacy bridge**. This package provides an extension to the core package to connect it to existing applications; it provides facilities for enhancing "standard" applications with context-aware behaviour. It is worth noticing that, while all our architectural abstractions can be implemented in most object-oriented languages, this particular package depends strongly on the reflexive capabilities of the chosen programming language.

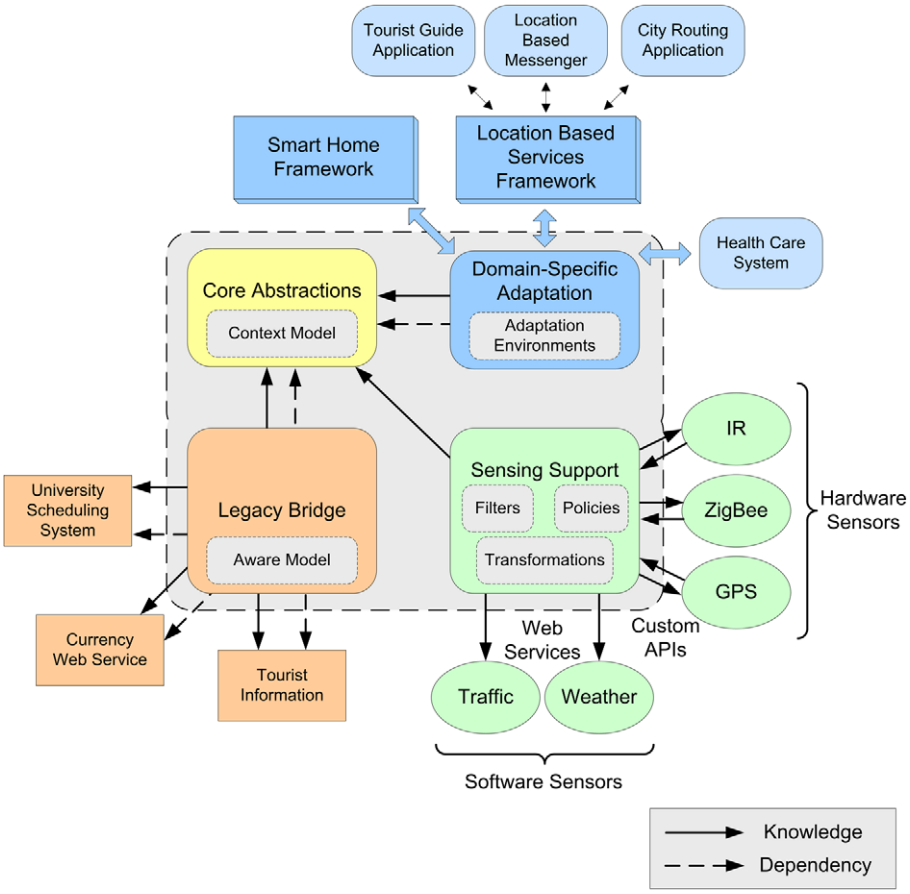

Fig. 1. A schema of the platform's architecture.

- **Sensing support**. This package is in charge of connecting sensing devices with the core abstractions in such a way that the sensing process becomes transparent to the context model. This package handles sensing policies, signal filtering and transformation between low-level data and high level context information.
- **Domain-specific adaptation**. The abstractions in this package provide the skeleton for defining domain-specific behaviours; developers concentrate on this package when devising a new application or framework. In this case, they extend existing classes, by using the abstraction in a white-box style (i.e. by specializing components).

6. Micro-architectural abstractions

In this section we will revise the variant features presented in Section 4 and we will show how the micro-architectural abstractions depicted in Fig. 1 can help designers and developers to effectively realize their associated variation points. In addition we will characterize each variation point based on the taxonomy of variability realization technique detailed in Svahnberg et al. (2005) and mechanisms detailed in Coplien et al. (1998).

6.1. Building dynamic context models

Defining a context model is a major issue in most context-aware applications, especially in those where we expect to incorporate new context features at a relatively constant rate. In Section 4 we stated the requirements for the context model and we defined three main guidelines:

- Context is defined in an instance basis.
- The context of an object is split in a set of *context features*, each one focusing on a specific domain (e.g. location, activity, weather, etc).
- The context shape of an object is defined by the context features associated to it. This shape can change in run-time.

As an example, consider a friend finder application (Schilit et al., 2002). In this application we would need to know the user's location in order to provide the friend finder service, which we assume is gathered form the user's GPS device. However, some users may also have a compass embedded in their handheld, which could be used to enhance the application by giving navigation hints. Thus, two different instances of a user class may have different context models. Also, the context model must be configurable enough to share schemas across different adaptation domains, but allowing each application to add new context features to a specific entity. Finally, the context model should support run-time changes.

Based in the requirements presented in Section 4 we devised a fine-grained context model that works at the (object) instance level. This model is based on two basic concepts: *aware objects* and *context features*.

An *aware object* represents an object whose context is important for the application. As an example, suppose that we want to provide a temperature management service for a smart home, so that the temperature remains low while the user is at work and rises automatically when he or she arrives at his or her home. For this purpose we must track the user's location and the temperature of the house, which in our approach means that the house and the user must be represented with aware objects.

At the design level an aware object is modelled with a class (AwareObject) and used in a black-box fashion (i.e. we do not expect a developer to subclass it but to instantiate and configure it). In order to actually define the context shape of a given aware object, we use the notion of a *context feature*. A context feature is a reification of an aspect of the object's context that we want to track. In its simplest form, a context feature is similar to a Value Holder (Woolf, 1994) that holds the current value of a context property and informs any interested part whenever that situation changes. Thus, to create a context model we must instantiate an aware object and a context feature for each context aspect that we must track. The connection between an aware object and its context features is provided by a set of messages defined in the AwareObject class (in particular addFeature and removeFeature).

Finally, when a context feature is added to an aware object, a connection is established between them so that the aware object knows when the context feature has changed. To do so an aware object is registered as an Observer (Gamma et al., 1995) of its context features, receiving a notification when any of them has changed. In Fig. 2 we show a simple example of a user object, configured with a location context feature, which is modelled as a (latitude, longitude) pair.

With this small-grained description of the context model we can see that what was presented as a variant feature in Section 4 (i.e. context models) is actually realized with a set of variation points associated to context features. At design time the objects that will be aware of their context are defined and their context shape is defined by designing and implementing each context feature. However, while the application is running a context feature may not be active (e.g. because there is no sensing device to gather information about it) and thus the context shape may need to be redefined. For this reason we consider each context feature as a variation point of an aware object; each context feature is closed at design time and is treated as an optional variant that is re-evaluated in run-time. At the implementation level the mechanisms associated with this variation point are inheritance and polymorphism (as mention by Coplien et al. (1998)).

6.2. Towards a context feature library

Even though characterizing each context feature as a variation point that is open for adding custom variants is a flexible approach we found out that some types of context features appeared in a recurrent fashion and could be implemented for later reuse. In particular we found three types of distinctive context features:

- Model-based features: In many cases the current value of a context feature is meaningless without a supporting model. Let us suppose that we are tracking the user's location in a symbolic map (Leonhardt, 1998), and for that purpose we have a context feature that holds the symbol that identifies the room where the user is standing. Without the map itself, the symbol is not of much use, since we can not connect it to other locations. A model-based feature extends a context feature by adding a reference to a model.

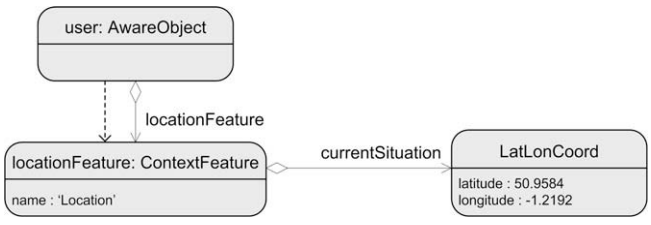

Fig. 2. A simple context model.

- Tracked feature: Some applications require a history of the values which a given context feature has been assigned to. Using again the location example, a program could predict the user path to provide LBSs (e.g. comMotion (Marmasse, 1999)). For this purpose a tracked feature extends the basic context feature with the capacity to keep a log of the different values which have been assigned to it.
- Derived feature: Context features represent small-grained pieces of context information. In some cases we can combine that information (e.g. applying a transformation) to produce derived context information. As a simple example, suppose that we want our smart home to warn us if the weather is suitable for going cycling. If we have context features like temperature, pressure and wind (e.g. because we have installed a small weather station at the house's roof), we could use a decision tree (Quinlan, 1986) whose inputs are the current weather conditions and the output is the suggestion made by the program. An important characteristic of a derived feature is that it automatically updates its current situation (and triggers a change event) when any of its input features change. To make this object as generic as possible, the transformation used to derive the situation from the input features is expressed by a block closure.[5]

Finally, we should notice that tracking the changes of a context feature or requiring an underlying model are not mutually exclusive. For these reason the extensions enumerated earlier are implemented as Wrappers (Gamma et al., 1995) and, if needed, the developer can add its own wrappers. In Fig. 3 we show a class diagram depicting the ContextFeature hierarchy.

Note that what we are doing in this case is providing a set of common used context features to designers and developers. This does not mean that the variation point associated with a context feature will not be open until the implementation phase, since new requirements may arise and new subclasses may need to be added. Thus, our aim is to allow designers and implementers to incrementally add new variants to the collection of variants associated with the variation point.

We next show a more complete example depicting how different context features can be used to create more sophisticated context models. Fig. 4 shows the context for a simple memory-aid application (like the one presented by Lamming and Flynn (1994)) that records the user's actions (e.g. through software sensors) and his or her location. Since we need to log the user actions, the *action* context feature is wrapped to become a tracked feature, keeping a history of the functions performed by the user. A second context feature (*location*) is used to place the user inside a building; notice that, since we are using a symbolic model (Leonhardt, 1998), symbols alone (like 'Room 102') are not sufficient. In order to perform meaningful computations we need a reference, which is the feature's model (i.e. the symbolic map).

Finally, notice that in some situations the user may not be able to sense his or her location (e.g. because the location was sensed inside the building and the user has left it). One possible approach for handling this case is to stop taking into account that aspect of his or her context (i.e. if we can not measure it, then we can not provide services based on it). In our model this is achieved by removing the context feature from the aware objects by means of the removeFeature message.

As a result, by assigning context in an object-basis and splitting each object's context in a set of context features we are able to

[5] A block closure is basically an expression whose evaluation can be deferred to a later time, pretty much like an anonymous function can be created and further evaluated.

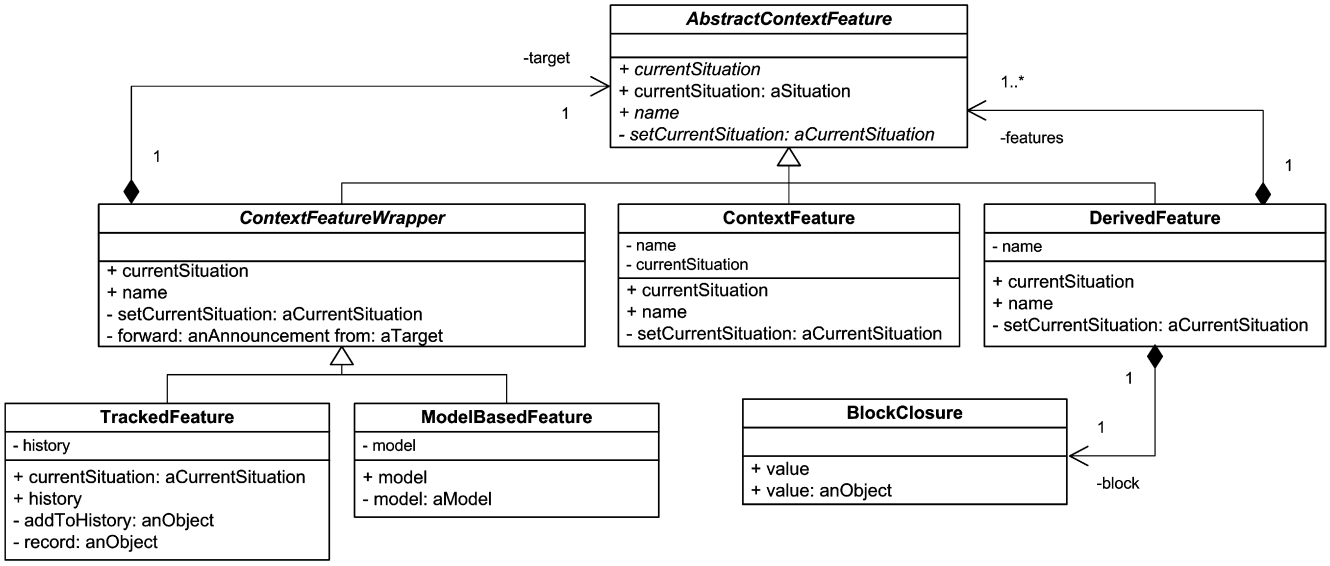

Fig. 3. The ContextFeature hierarchy.

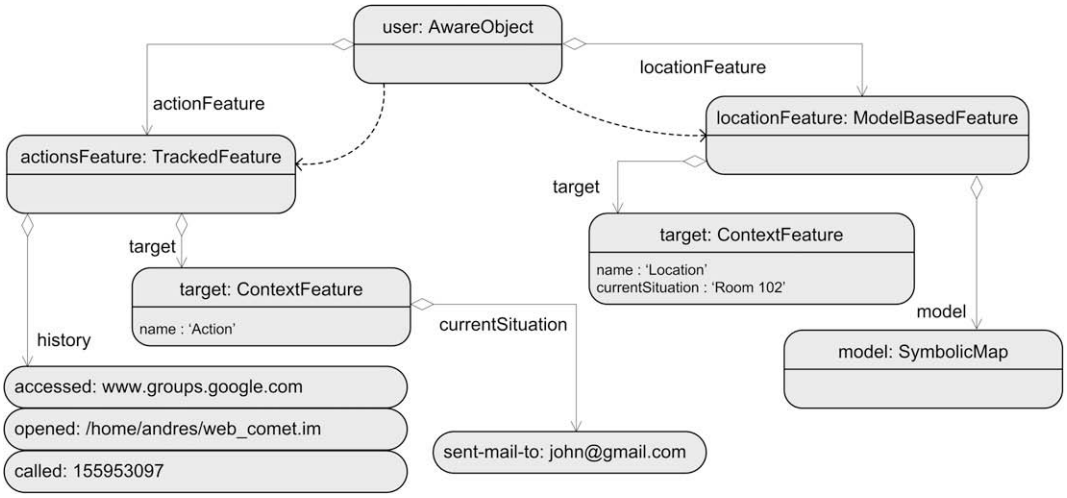

Fig. 4. A reduced context model for a memory-aid application.

engineer each feature in a separate way, therefore reducing the context model overall complexity. This is extremely important since some features may require non-trivial underlying models (e.g. location models as described by Leonhardt (1998)). Also, since aware objects provide the required messages to manage their context (i.e. addFeature and removeFeature messages), context features can be added or removed at any time, in particular at runtime. This is why we treat each context feature as a variation point with an optional variant pattern that can be re-evaluated in runtime.

With these basic mechanisms we are able to manage the second and third requirements presented in Section 4.1. In Section 6.5 we show how we combine the context model with domain-dependent adaptation behaviours in order to reuse context schemes.

6.3. Legacy bridge

As we explained in Section 4.2 sometimes we need to extend an existing application to add context-aware behaviour rather than building a new application from scratch. Besides the requirements presented in Section 4.2 we would expect to reuse any context feature previously defined, as we shown in the previous section. To achieve this, the AwareObject class is extended (subclassed) in our architecture into an AwareModel. AwareModels are aware objects that reify existing concepts in the application model, and create a representation for them in the context model. To do so, an aware model holds a reference to the application object and acts as a dynamic Proxy (Gamma et al., 1995), forwarding every message it receives to the base object. Also, since an AwareModel is a subclass of an AwareObject it provides support for managing context.

Up to this point, an aware model enables us to represent an existing application object as part of the context model. However this class also adds behaviour to perform more complex behaviours:

1. In its general form, as long as a getter and setter[6] are provided, any aspect of the base object can be treated as a context feature. This means that we can reify objects as aware objects and also their attributes as context features.

[6] In order to keep consistency with the context feature behaviour, we also need the base object to implement the observer pattern for the desired aspect, so that a notification is triggered when the aspect is modified.

2. An aware model can be asked to override a base object's message with a context feature. As a result, instead of forwarding the message to the original object, the aware model will return the context feature's current situation.

Thus, at design time we must decide if the context model of a given entity will be built from scratch or will be based on an existing application model. Since we have only two choices (i.e. aware objects or aware models) this variation point is closed and the decision of which of them will be used must be stated at design time.

As an example, consider implementing a mobile application for a travel agency. Suppose that the agency already has a system (used for booking trips, billing, etc.) and wants to provide a mobile application to aid its customers when visiting a foreign city. Since the agency has considerable information about its customers, it would be nice to reuse it for the mobile version; however, we want part of that information to be used as context, regarding the user's activities. Take for example a fragment of the requirements expressed in the CyberGuide (Abowd et al., 1997) project: *"For example, information should be presented in a way that is suitable given the age and technical background of the visitor and their preferred reading language. Context should also be used to adapt the presentation of information depending upon the information that the visitor has already seen. For example, if a visitor makes a return visit to a landmark then the information presented should reflect this fact, e.g. by welcoming the visitor back."*

Let us suppose that the user of the mobile application is modelled in the tourist agency system with a Customer class as shown in Fig. 5.

By using this class we can now create a context model that:

- Uses the default behaviour with the age message (i.e. just forwards it to the base model). Notice that, even though the client's age may change during his or her trip, it is generally unlikely that we would like to have it as part of the context model (as a matter of fact it may even be disturbing if the user is shown different attractions because his or her age has changed in the middle of the trip). Thus, messages from the base model that are not used to perform dynamic adaptation are just forwarded to the base model.
- Creates a context feature with pending tasks for the trip. For example, the user may be engaged in a trip that includes many cities and may have a hotel reservation waiting to be confirmed. Even though this is recorded internally in the system, we would like to expose it as contextual information, so that the application can provide personalized behaviour to the user while being on the road (e.g. if the hotel is still unconfirmed two days before the arrival, run an LBS that looks for similar hotels nearby). Thus, existing information that will be used to provide context-dependent services is now modelled as context features and thus made accessible to the services.
- Is able to change the desired language to display tourist information. In principle, the language used to display information may be inferred from the user's nationality. This information is useful to the travel agency, since it enables them to communicate with their clients in a personalized way. However, during his or her trip the user may want to see the information in other languages (e.g. because the translations are poorly made or incomplete). This information might vary dynamically (maybe many times a day in a guided tour), and clearly this fact should not be modelled in the tourist agent system. Thus, we can create a context feature that overrides the base object's behaviour but uses, as default value, the systems' response.

In Fig. 6 we show an instance diagram of the derived context model.Notice that a new class appears (FeatureAdaptor) in order to bridge the application model with the context model. Let's now see how these ideas are used to deal with both requirements enumerated before:

- The context model should be transparent to the base application. The aware model and the context features are placed as a layer on top of the base application, pretty much in the same way a view is placed on top of a model in the MVC (Krasner and Pope, 1998) paradigm. Thus, the context model can be engineered and changed without affecting the base application.
- Application information may be treated as context. By using context features derived from attributes and context features that override behaviour we can represent application concepts as context information.

Even though these requirements are satisfied, there are still some issues regarding the implementation of the aware model's behaviour. In particular the dynamic proxy behaviour and the way messages are overridden, heavily depends on the language being used. In our case the architecture has been implemented in

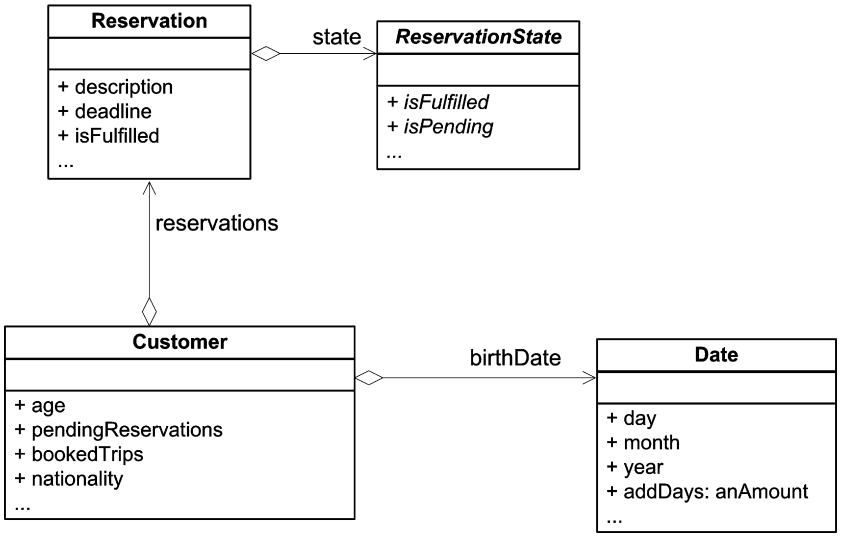

Fig. 5. A subset of the class model for a tourist system.

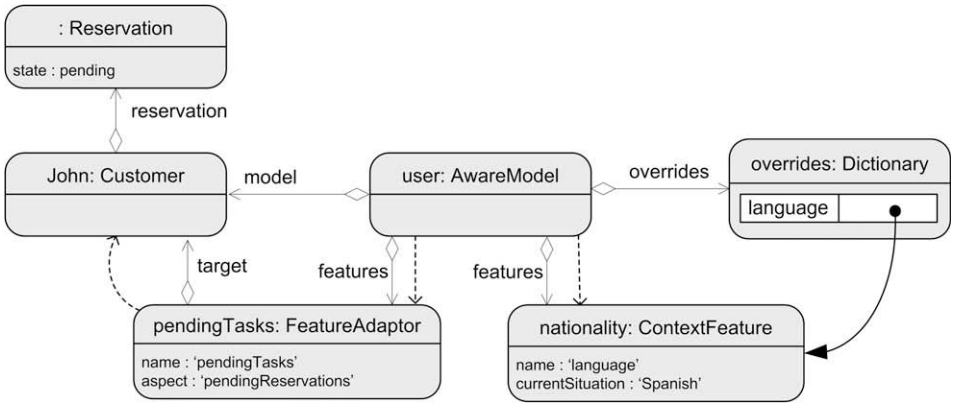

Fig. 6. Instance diagram of a context model extending an application object.

VisualWorks Smalltalk, which is a naturally reflective environment that simplifies the specification of this kind of behaviour.

6.4. Sensor support

Once the context model is created we must define how the external information will be gathered and feed to each context feature, taking into account the requirements presented in Section 4.3. To do so our architectural solution treats the different stages involved in the sensing phase as first-class citizen by reifying them into objects and isolates them by defining clear communication interfaces. As a result, each stage is represented by an interface (and realized by one or more classes), which can be easily changed to adapt to new requirements. As we will show later, this results in many variation points used to configure the sensing package according to each sensor-specific requirements. In Fig. 7 we present an instance diagram, depicting the main objects involved in the sensing phase.

A Sensor object represents an instance used to communicate with the sensing mechanism, which can be a hardware-sensor API or a software sensor, like a web service. This sensor is managed by a Policy, which is used to abstract whether the sensor works in a push way or needs constant polling. As a result of this step we get a common publish-subscribe mechanism to interact with any sensor.

The SensingConcern is in charge of receiving the new sensed values and triggering the pending steps; its first task is to handle the value to a Dispatcher, which is in charge of deciding if the value is acceptable. The meaning of "acceptable" depends on different factors and most of the times it is an application-dependent issue (e.g. a 10-meter error in a GPS reading may be acceptable for driving directions, but not when providing LBSs to a walking user). Once the sensed value gets through the dispatching phase, it must be converted to an object (or a set of objects) that match the model being sensed. For example, a GPS location can be converted by means of reverse geo-coding to a specific address. Once the Transformation phase is done, the sensing concern can update the context model by assigning the new value to a ContextFeature (e.g. the mobile user's location).

In our design the SensorPolicy, Transformation and Dispatcher classes are single variant points, since the developer needs to choose one option when creating the final application. In addition, these variation points are open during the implementation phase because the developer can create new subclasses of them if required. However, as we did with the context features, we have implemented many typical cases of policies and transformations, so that the variant collection can be enriched over time. These are shown in Fig. 8, where a class diagram depicts the main abstractions and some particular specializations that proved to be frequently used, such as lookup transformations (e.g. to map a sensor ID to a room) or error filters, that only accept a value if its error level is under a given threshold.

Finally, to complete the description, we show an interaction diagram depicting the process of dispatching the GPS signal of a mobile user (see Fig. 9). Notice that in the diagram a new dispatcher is presented (DOPDispatcher); this dispatcher only allows GPS signals whose dilution of precision (Langley, 1999) provides a minimum quality.

As a result of separating sensors from the context model and encapsulating each sensing stage by means of well defined interfaces we reduced the impact of changes in two levels:

- In a macro scale, we insulated the context model from technological trends. By using the proposed approach the developer can concentrate in the context model and its design and later manage the sensing related issues. This not only simplifies both tasks (since dealing with them separately splits the complexity) but also enables us to incorporate new sensing devices without affecting the applications.

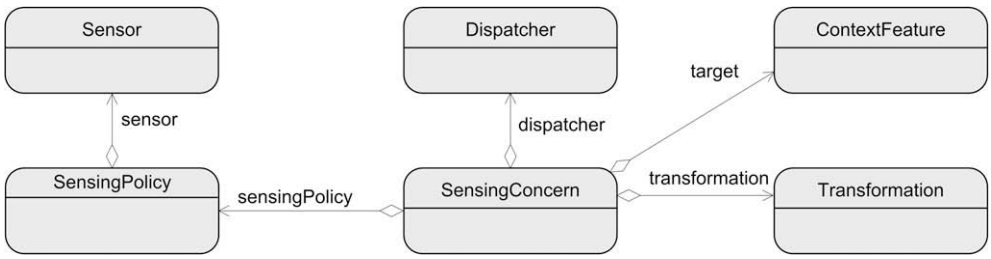

Fig. 7. Simplified instance diagram of the sensing layer.

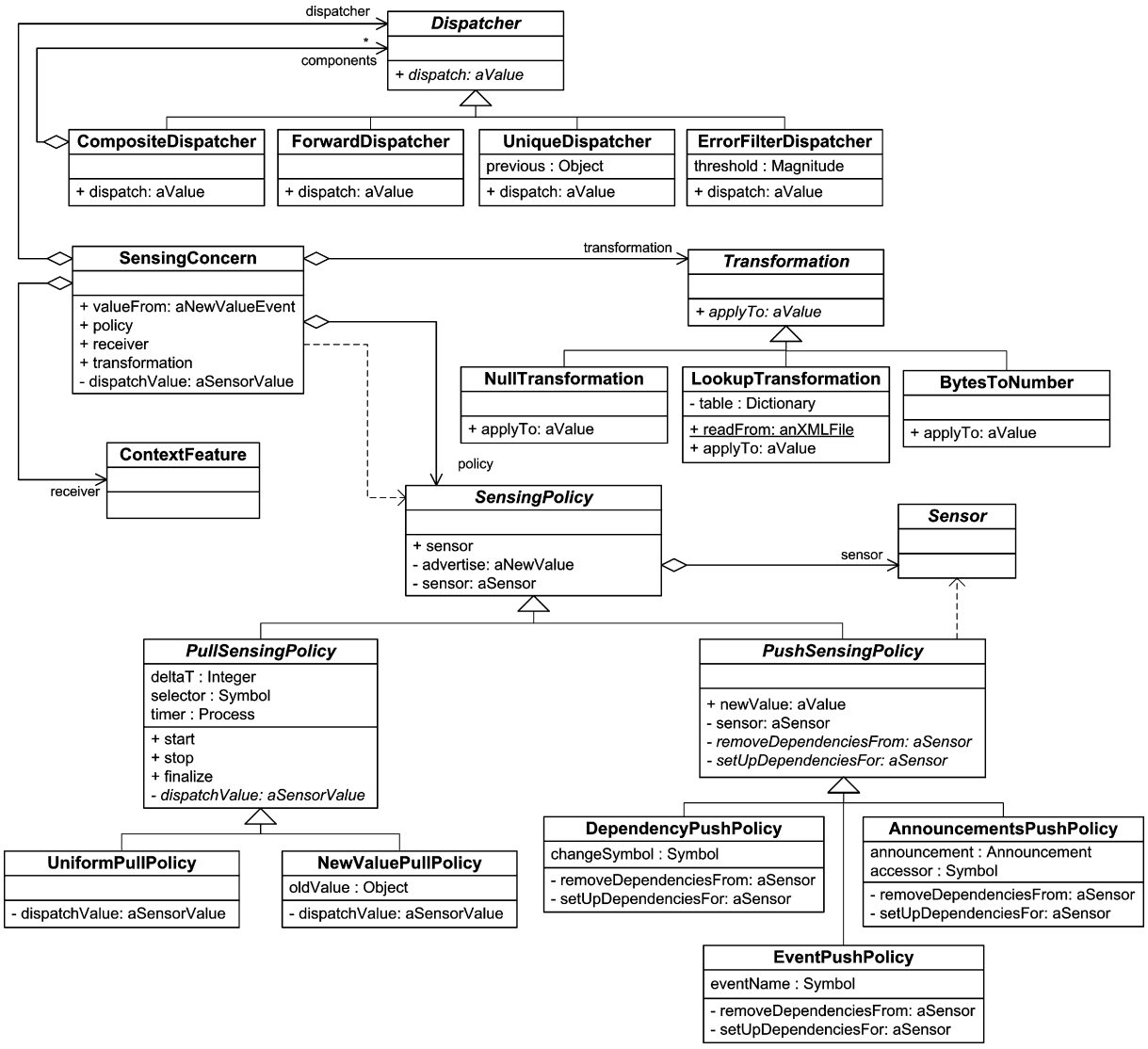

Fig. 8. Class diagram of the sensing architecture.

- In a micro scale, we are able to quickly implement new variations and reuse existing components. Since each stage of the sensing process is clearly outlined and its boundaries are well-defined, changing a part of it (e.g. replacing a transformation or changing a dispatcher) is precise task. Also, since the stages are fine-grained and have very clear responsibilities they can be easily reused. For example, a class which has turned out to be very useful is a transformation that maps id's received from sensors to objects in the context model domain. This class implements behaviour from reading files (like XML) and building the corresponding tables. Once implemented and tested, this class can be used to solve different problems, like mapping the id of a beacon to a symbolic location or associating a bar code to an object.

6.5. Domain-specific adaptation

As we previously explained, one of our goals is to separate the adaptation domain from the application domain to get a clear separation of concerns, resulting in better and more flexible designs. To clarify this separation consider the task of providing LBSs to a mobile user: in this scenario we can identify the minimum required context (the user's location) and the expected behaviour (according to the user's location, let him access a set of services). However, the service's domain (e.g. health-caring, traffic monitoring or finances) is, in principle, not relevant. We have aimed at separating the application domain from the adaptation domain, so that developers can build specific infrastructures (e.g. for LBSs) that can be reused across different application domains (e.g. health-caring, finances, etc.). To do so we decided to treat adaptation domains as first-class objects, which are called *adaptation environments*. Each adaptation environment references a set of aware objects in order to "listen" to their context changes. When a context change is triggered, a notification is delivered to the environment so that it can perform a specific action, like showing a new service in the case of LBSs or turning off a light in the case of a smart home. An aware object can be registered to more than one environment at the same time, therefore allowing context sharing between applications.

By combining environments with a notification mechanism each time the aware object's context changes the environment is notified and can perform the corresponding action. For example, if the user moves and approaches to a painting, the environment

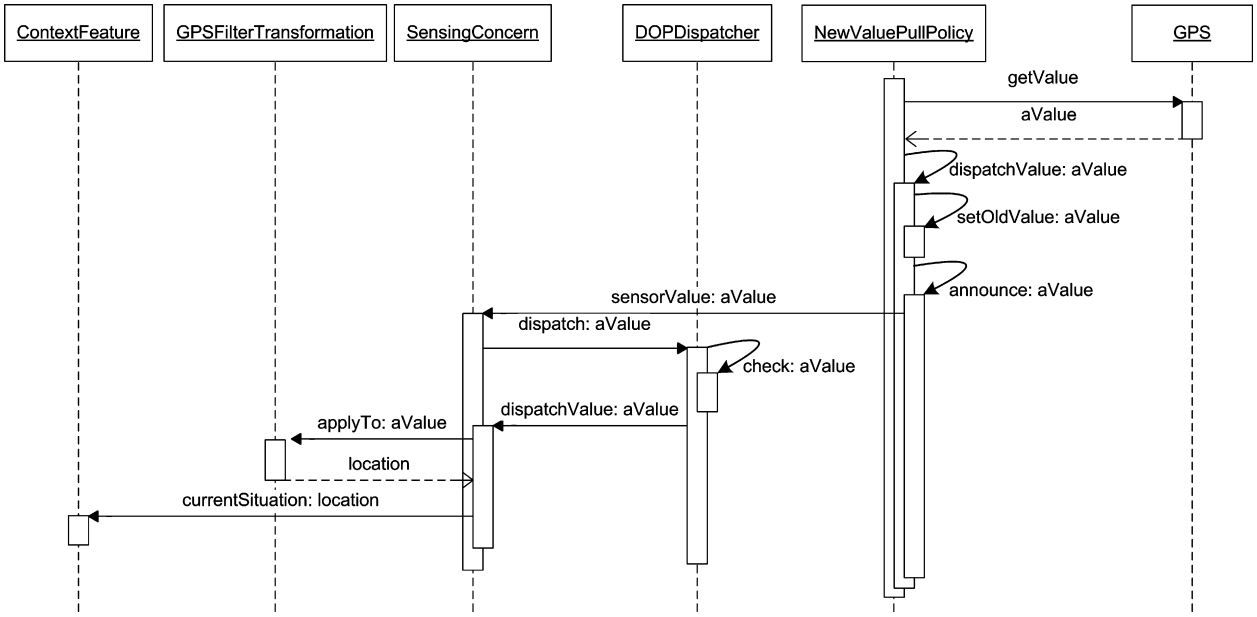

Fig. 9. The process of dispatching a GPS signal.

can react by showing information about that painting in the user's PDA. However, the user may be at the same time registered to another adaptation environment (e.g. context-dependent reminders), which means that different types of functionalities can be provided at the same time (in this case, LBSs and context-dependent reminders).

In order to show a basic example of an adaptation environment, consider the case of a mobile memory-aid application (the context model we use for this example is the one shown in Fig. 4, but with less detail for the sake of clarity). To implement this application a new environment must be created and the aware object registered to it (see Fig. 10). As a result, each time the user performs a new action (e.g. sends an e-mail) the environment gets a notification and can perform an action, like recording the action in a persistent repository with the current timestamp.

Even tough this separation between context and adaptation works well for small environments, as they evolve and become more complex, managing changes turn into a complex task. In particular two important issues must be considered:

- We may be interested in performing many actions for a change in a given context feature (e.g. updating the user's location in a map and checking if a new LBS can be provided).
- Most of the times changes in each context feature must be treated in a specific way.

For these reasons, we decided to refactor the internals of an adaptation environment by separating the way in which each context change is handled. This led to the introduction of *context event handlers*, which are in charge of performing the reaction to a context change. In the resulting architecture an environment acts as a Façade (Gamma et al., 1995), providing a simple access protocol for registering aware objects and handlers for specific changes, and internally coordinating the flow of events. It also coordinates the access to resources shared between handlers (e.g. the connection to a database). To sum up we have decomposed a variant feature (the domain specific adaptation) into a set of variation points, one for each handler. These variation points are closed at the implementation stage and can be optionally enabled (disabled) at

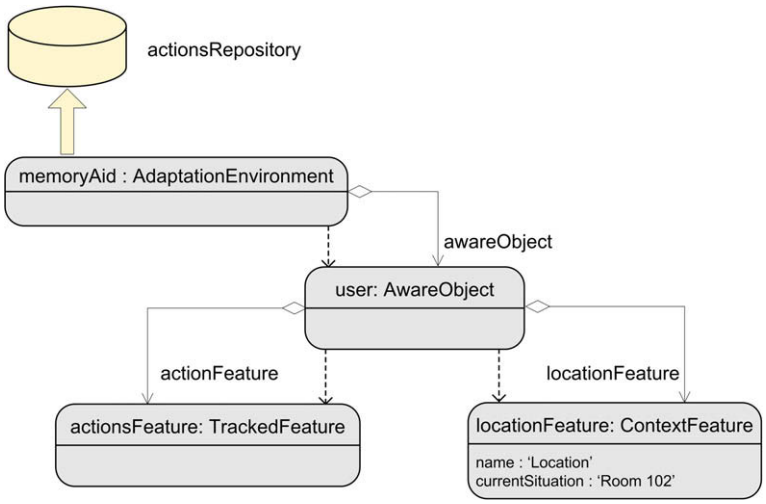

Fig. 10. An adaptation environment for a memory-aid application.

run-time. As mentioned by Coplien et al. (1998) we use inheritance and polymorphism as mechanisms to implement this variation point.

To exemplify let's return to the memory-aid application: when the *action* feature changes, we must store it in a persistent repository. We may be also interested in performing a secondary action, like keeping statistics for further estimation of events (e.g. the *entering a room* action and its timestamp is used to later answer queries about the probability that the user will be in a room at a given time). Finally, we may also be interested in the *entering a room* event in order to check for other present people to implement a context-dependent personal notes (e.g. if *John* is in the room then a personal note related to him may appear in the screen). As it can be easily seen, not only the possible actions attached to a context event are endless, but it is also impossible to foresee them all while developing the application (e.g. the last example can be a requirement added after the first release of the application). To make things more complicated we may even want to enable/disable certain behaviour at run-time (e.g. the user may want his or her actions only to be logged while he or she is at work and not in his or her spare time).

Decoupling the environment itself from the actions to perform on context changes by means of the context event handlers enables us to clearly specify them as variation points and thus manage the software evolution. In Fig. 11, we show an example of an environment with two handlers attached.

Finally, a special issue to take into account is the relationship between an aware object and the environments to which it is registered, since we want changes on either side to have a small (or null) impact on the other. In particular, by using a notification mechanism (i.e. implementing the Observer pattern (Gamma et al., 1995)) to communicate an aware object with its environments and by allowing environments to register to specific context changes we get the following benefits:

- Since an aware object is oblivious to adaptation environments, changes in the environments do not impact on it. This guarantees that already existing software can not be broken by changing or adding a new environment (notice the analogy here with the MVC, where the aware object takes the model role and the environment the view one).
- Adaptation environments can register to listen for specific context changes. Thus, a change in the context schema of an aware object will only impact on the environment if the object is explicitly registered in it. As an example, if we remove the *location* feature only those environments that are registered in it will need to be modified. In particular, since the relationship is created between context features and handlers, only those handlers that expect changes for the location feature (and not the whole environment) will require changes.

It is important to notice that this part of our architecture is expected to be used in a white-box style, since the handlers and the environments must be tailored to suite a specific adaptation domain (sometimes even in a specific application domain). Our experience has shown that by repeatedly building applications in the same adaptation domain, domain frameworks can be constructed (e.g. a framework to manage LBSs). In Fig. 12 we show a class diagram with the base classes provided by the architecture (greyed) and the ones used to implement the memory-aid depicted in Fig. 11.

7. Experience and discussion

The architecture presented in this paper, together with its current implementation, is the result of an iterative learning process. As we developed new prototypes in different domains, new problems appeared and the architecture was upgraded to suit the new requirements. We consider that we have reached a point of stability where variant features and their associated variation points can be defined for most context-aware applications. Additionally we have identified the places where variation points can be realized by using white-box style (e.g. subclassing) and those where black-box style can be applied (e.g. composing existing classes or components).

In this section we briefly outline two case-studies that show how to build context-aware systems using our proposed approach and architecture. In particular we will show a framework for location-bases services and a memory-aid application (this application is based in the one described by Lamming and Flynn (1994)). We contrast these two examples since they expose some interesting differences:

- The memory-aid is a concrete application whereas the LBS framework shows that our infrastructure is also suited to building frameworks for specific adaptation domains.
- Even though other contextual aspects can enrich the user's experience, LBSs are mainly focused on the user location and thus the basic context model is quite simple. On the other hand, memory-aid applications can have very rich context models, including the user's phone calls, mail sent, documents, etc.

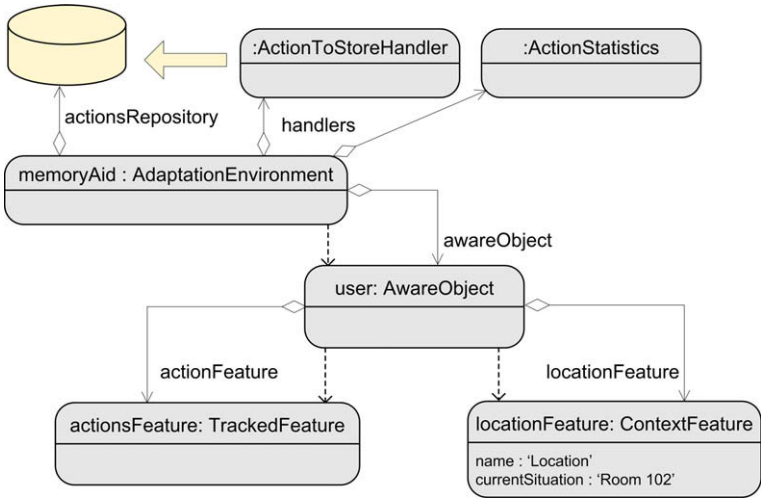

Fig. 11. The refactored environment using handlers.

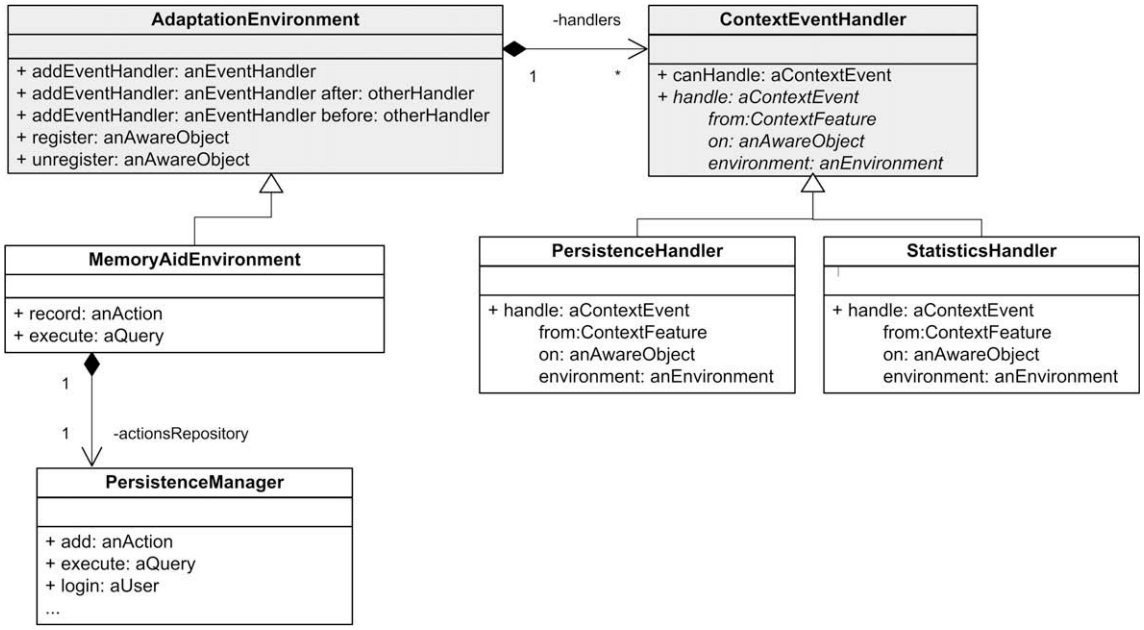

Fig. 12. White-box extensions to the basic toolkit.

- Also related to the previous issue, the main type of sensor used in LBSs is related to location (e.g. GPS), while a memory-aid application can receive information from many devices (desktop computer, other mobile devices, telephone centrals, etc.).
- LBSs are based on an implicit user input (i.e. changing his or her location) in order to trigger a new behaviour. On the other hand, the primary functionality of a memory-aid application is to let the user perform explicit queries (e.g. 'searching for a document used in a meeting with John and Alice last week').

In the next sections we will show the variant features of both systems and we will discuss their design and implementation. In this context two issues are left out of the discussion:

- Graphical user interface. Since we are mainly focused on architectural aspects we will not discuss issues related with the GUI.
- Low-level sensors. We omit discussing issues related to low-level sensor access (i.e. the specifics of the APIs of different vendors). In the case of the GPS for location we used a simulator that delivers an object that maps to the GPS_POSITION structure used in the Windows Mobile Intermediate Driver (2006). For the memory-aid application the simulator delivers directly the user actions, modelled as objects (see Section 6.4).

Finally, the implementation of our architecture and the prototypes were built in VisualWorks 7.4.1, a dialect of Smalltalk developed by Cincom. The product provides a cross-platform virtual machine that includes Windows, Linux and Windows Mobile OSs.

7.1. A framework for location-based services

The aim of the framework is to support the development of applications that provide LBSs. Notice that, as we already explained, these systems can be built on a myriad of application domains. To exemplify its usage we implemented four (relatively simple) applications:

1. Friend finder. The idea is to help the user to locate friends or contacts. We use a symbolic location model with three levels of awareness: a friend is in the same place (e.g. the same room), nearby (e.g. in a room adjacent to the user's room) or far away.
2. Location-based information system. In this case the aim of the system is to present hypermedia information associated to a given area. This system works both with symbolic locations (using a hash table to map symbols with web pages) and with geometric locations (using an r-tree (Guttman, 1984) to map areas with web pages).
3. Location-based messenger. This kind of messenger works as a normal one, except for the fact that a message is delivered to the recipient only if he or she is in a certain physical area or range (generally the same room of the sender). When a user sends a message to a contact outside the sender's area, the message is placed in a queue of pending messages, and only delivered when both the sender and the receiver are in the same location.
4. Location dependent file repository. This kind of repository works like an FTP server that is enabled only in certain locations. As a result we can relate a given location with contents shared (and edited) between users.

The process of building the framework followed well known practices (Fayad et al., 1999): we started by building one application from scratch, and as we developed new applications the commonalities were extracted and factored in specific classes. Having reached a stable design, we can now see which variation points were solved and which new variant features where introduced. At first sight adding new variant features may seem contradictory, since we are building a concrete product from a product line. However, the product we are building is a framework and this means that it is not a finished artefact, but a customizable one. Therefore, two rules should be followed:

1. A variant introduced at the framework level must be used to help solving an existing variant in the underlying architecture. In other words, we cannot introduce a variant at the framework level if it was not conceived at the product-line level.
2. The new variant cannot be more general than the one it is helping to solve. For example, if the product-line variant states that it is closed at design time a framework variant cannot change it to be bound at run-time.

The idea behind these two rules is that if we derive a framework from a product line, the framework cannot be more general that the product line itself. We will now show the resulting framework design and afterwards we show which variation points were solved and which new variant features were introduced.

As a result of using our architecture, the framework was built with 8 classes, 5 of them conceived to be extended by subclassing (i.e. as a white-box framework) and 2 of them to be used by composition. The remaining class was an extension to an already existing class, part of the Smalltalk hierarchy. In Fig. 13 we show a class diagram of the framework, where the original classes (i.e. those implemented as part of the basic architecture's toolkit) are greyed out.

We next outline the main responsibilities of each class:

- **ServiceSpec**. Users that are interested in a particular service must register to its specification. The specification gives information about the service (name, description, etc.), and also indicates when the service can be available for a given user. As an example, consider any kind of LBS: if the user does not have a location context feature, then we can not determine his or her position, and thus we can not decide if we must provide the service or not. By default, the service checks for a (initially empty) collection of required context features which can be extended by the ServiceSpec's subclasses.
- **Service**. The Service abstract class does not provide much behaviour on its own, but defines a set of accessing methods for commonly needed information and a set of methods that will be used as callbacks by the environment. Examples of these methods are start and stop, which are called when a service is activated and deactivated.
- **ServicesEnvironment**. A services environment is an adaptation environment specialized to handle services and users (i.e. services consumers). It adds user management, services specification management and coordinates, by collaborating with a service handler, when a service should be active for a given user.
- **ServiceHandler**. This is a specialized handler that is installed by the service environment for each user and for each of the services he or she has registered to. Since a service specifies which context features it requires to work, the service handler "listens" to changes of those specific context features. When a notification of a context change reaches the handler, it checks whether the service should be active for that user or not.
- **ServiceConsumer**. A service consumer is a wrapper of the standard aware object that adds service accounting information. It keeps track of the registered services, and of those that are currently available. Since we are interested in user-centred services, we will refer to these consumers as users, but it should be noticed that any aware object can act as a service consumer.

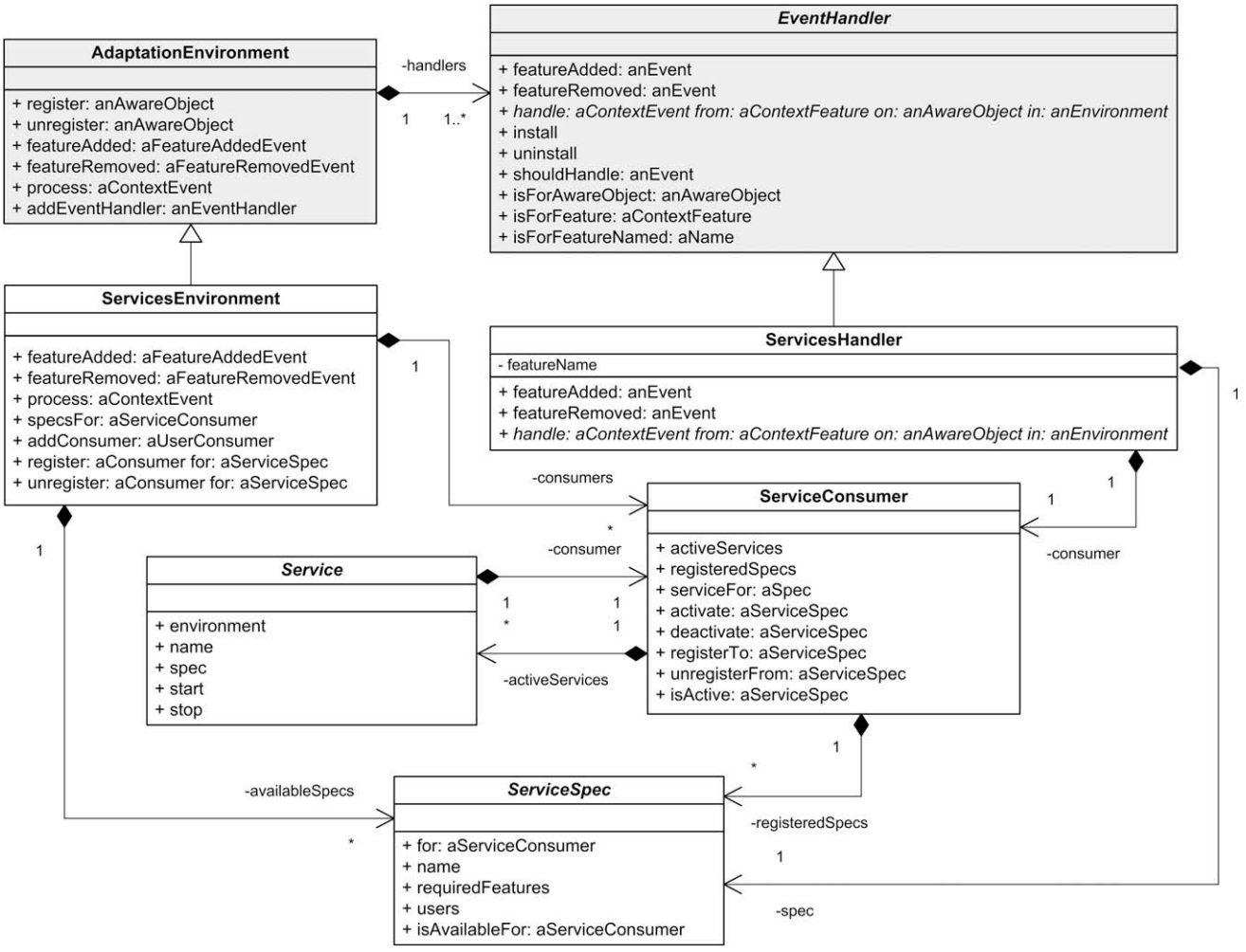

Fig. 13. A basic framework for context-dependent services.

We can now see which variation points presented in Section 4 were solved and which new ones were introduced as part of the framework:

- **Dynamic context model**. The framework constrains this variant by assuming a *user* object and requiring a *location* context feature. Notice however that new context features can be added (i.e. the variant still remains open) in case a particular service requires this (e.g. a service may need the user's current *activity* as a context feature to work).
- **Legacy bridge**. The four applications that we implemented did not require a previously existing application model, thus this variation point was solved at design time by using aware objects.
- **Sensor support**. As we explained in the introduction of this section we have constrained the use of low-level sensors and we tested our framework with a simulator. Thus, the variation points related to this variant feature were solved in design time.
- **Domain-specific adaptation**. This is maybe the most interesting variant feature, since it is partially solved by specifying new classes which have their own variation points. The ServiceEnvironment class specializes AdaptationEnrironment while ServiceHandle specializes the EventHandler one. Thus, the variation points related to this variant features are solved. However, these classes interact with other (abstract) classes that are used as variation points, namely Service and ServiceSpec. The variation points added are clearly more specific than the ones present at the product-line level and are constrained to adding a concrete subclass for the two abstract classes.

Once the framework was finished, building each concrete application required between 2 and 5 classes related to the context-aware functionality (for example, the classes related to the implementation of the r-tree are not considered here). On average, 3 additional classes were needed for each application, with an average of 9 methods per class. As we said before, subclasses of Service and ServiceSpec are needed in every new application.

7.2. A memory-aid application

An important requirement for a memory-aid application is that the sources of information can be configurable and dynamically available (i.e. we may have a connection with a desktop PC that is later lost). To achieve this, each source of information is treated as a software sensor and processed as explained in Section 6.4.

At the context model level, we evaluated different design options, like having a context feature for every source of information, having a single feature of actions and a specific class representing each action, separating context features regarding the types of actions, etc. From our point of view, the clearest model consisted of separating each type of action in a specific context feature; as a result, we have a context feature for email actions, other for phone calls, etc. We also model an action hierarchy, where each specific subclass represents an action performed by the user (e.g. send an email, receive a message, make a phone call, etc.). These design decisions lead us to a very easy implementation of a query approach quite similar to *query-by-example* (Zloof, 1977), where a "template" action is instantiated and only those specified attributes are taken into account to perform the search. For example, issuing a query whose parameter is an instance of UserAction (the root abstract class for any action recorded), would retrieve all instances of any of its subclasses. On the other hand, by creating a MailAction whose device is set to the user's mobile phone and subject is 'Call for papers' will retrieve all the mails sent from (or received) in the user's mobile device, whose subject matches the string pattern. In Fig. 14 we show the hierarchy of actions modelled for the prototype (for the sake of clarity we only show a subset of all the classes).

The next step to build the application is to create a new adaptation environment and a handler, so that when any *action* feature changes, the new action is recorded and persisted[7] (see Fig. 15).

As we did with the LBSs framework, we can now see how the variation points were solved in this concrete application:

- **Dynamic context model**. In this application there is no mandatory context feature, as in the case of the location context feature in the LBSs example. In case no context features are added to a user's context, the application will not be able to record anything and thus would be useless. The power of the application is increased as new context features are added, since there is more information available. However, this variant is constrained in the sense that if we want to add new context features whose information will be used for later retrieval, the values of those features should be instances of UserAction subclasses.
- **Legacy bridge**. As in the previous example there is no previous application to extend and thus no need for using the aware models described in Section 6.3.
- **Sensor support**. Since the data from external applications (e.g. a mail client) are treated as sensor information a new sensing step must be added for each new external application to support. Thus, this variation point remains unchanged until the concrete applications to be supported are defined.
- **Domain-specific adaptation**. This variant is closed at design time and is realized by the memory-aid environment, together with the PersistenceHandler and the actions classes.

Up to this point we had to code 20 classes (16 of which where used to represent the possible user actions), and 3 class extensions to the Smalltalk hierarchy in order to implement part of the query-by-example style for searches. However, even though this application is clearly ubiquitous and transparent (it is working all the time by gathering information), we considered that we could improve it further in two possible ways:

- If location information is available, add that information to the action. Following our approach, a location context feature was added and a new handler was written to listen to context changes and adding the location information to the actions using a Wrapper (Gamma et al., 1995). It is interesting to note that we could reuse the code from the LBSs framework to dynamically suspend the handler action when there was no location information.
- Let the user decide when specific information should not be recorded. To do this, we had to introduce a small modification to the handlers dispatching mechanism: so far, we were able to decide the order in which the different handlers of an environment where processed, but we have not been able to stop a handle for processing an event. Therefore, we decided to take into account the value returned by the handler when processing a context event; if it is false, we stop the process event. Thus, by putting a FilteringHandler as the first handler, we can avoid "private" actions to be recorded. To express the filters we used the already built support for query-by-example.

It is worth noticing that incorporating these two new requirements was straightforward and was managed as part of the variant features. The first one required adding a new handler, which is the

[7] For the prototype application this was done by serializing the objects to disk using BOSS, a built-in mechanism in VisualWorks. A deployed application would require another kind of persistence mechanism.

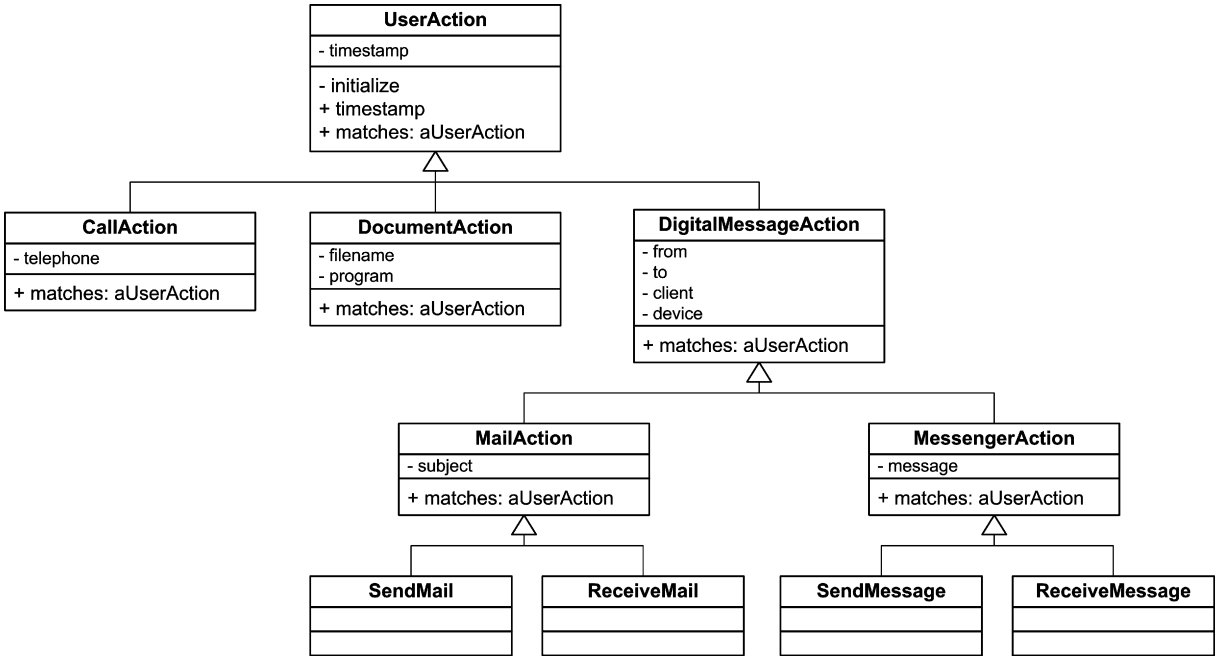

Fig. 14. A subset of the user actions hierarchy.

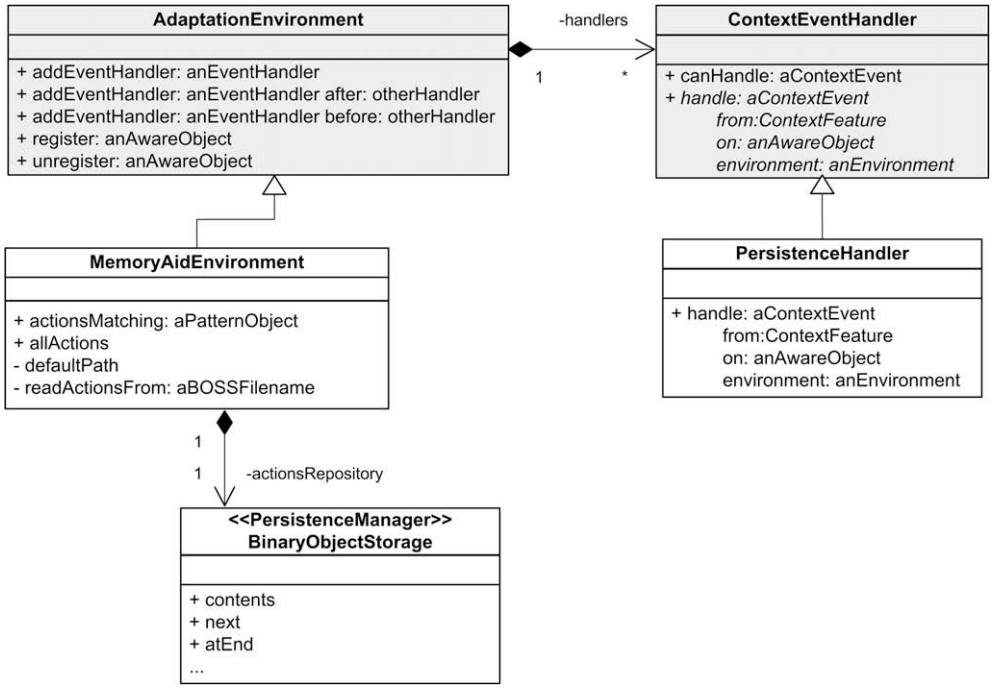

Fig. 15. First step to create a memory-aid application.

standard way of managing the fourth variant feature (i.e. the domain-specific adaptation). The second modification was a bit more difficult, since it required changing the way in which handlers are processed in the main architecture. In the modified version the return value of the handlers are checked for a Boolean result, which decides if the rest of the handlers will be processed. However, once the change was implemented, the new handler could be added as part of the standard variation point resolution. It is also interesting to note that this architecture refactoring did not impact in the LBSs framework implementation. Fig. 16 shows a class diagram of the final model to include the proposed extensions.

7.3. Discussion

In this section we have presented two concrete scenarios where our architecture and the characterization of variant features were tested. Even though the applications presented in the paper have been conceived in controlled environments, our variability characterization and architecture proved to be flexible enough to suit the applications' requirements. For a more elaborated case-study the interested reader can consult (Challiol et al., 2008), where new requirements are incrementally added to a context-aware guided tour.

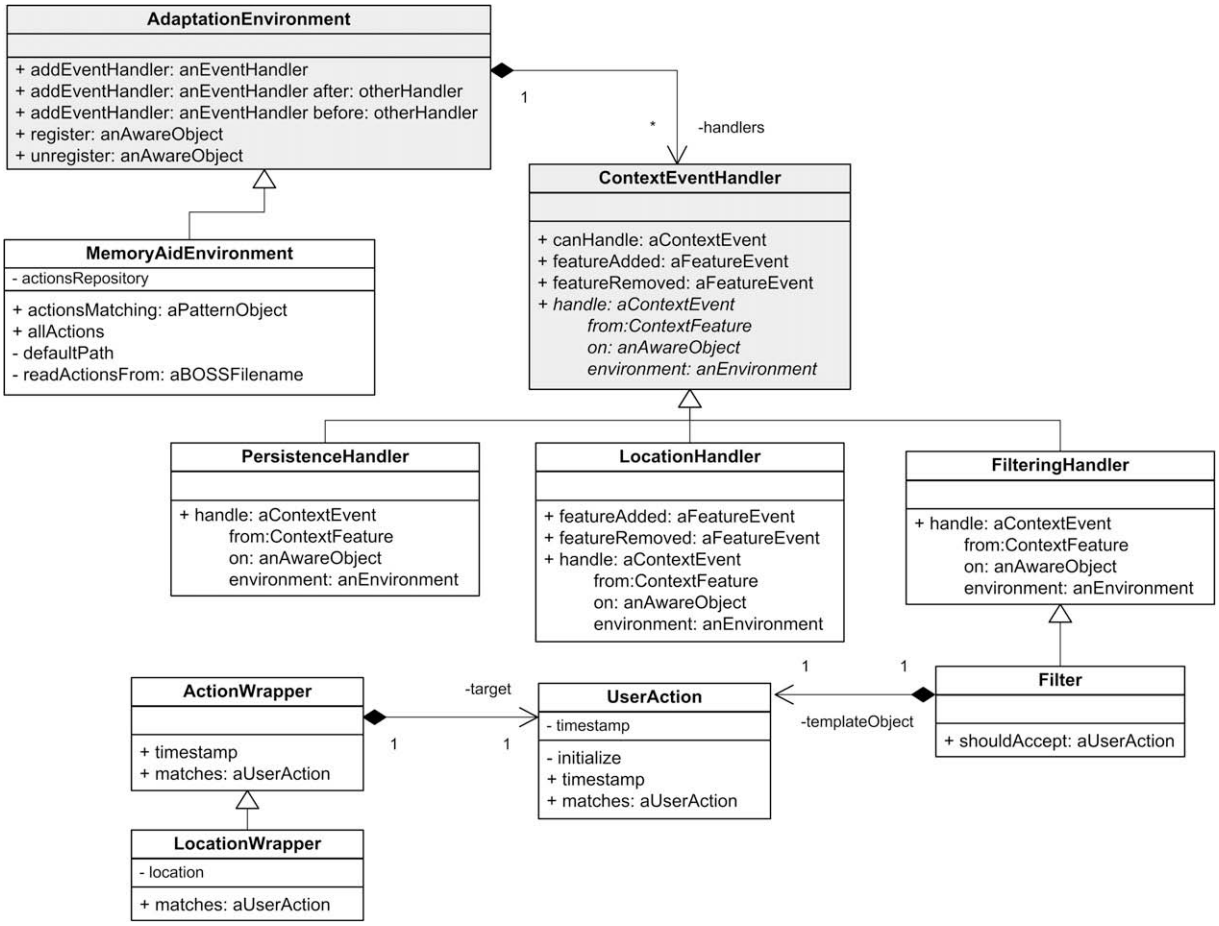

Fig. 16. The extended memory-aid class model.

To test our proposal, based on the identified variant features and a concrete architecture, we chose two different scenarios with distinctive characteristics. As a first example we showed how to build a framework for LBSs with our approach. Frameworks are a concrete representation of the knowledge in a certain domain, generally given by the developer(s) experience; their design captures and documents that experience and allows other developers to create applications in a given domain faster and more secure than if they were building them from scratch. To do so frameworks also have their counterpart of the product-line variation points, which are the hotspots and hooks. The task of building a framework was a challenge to see if the variant features and variation points proposed were flexible enough to accommodate the requirements of a framework for later deriving LBSs. In the process of doing so we realized that, since the framework stood in the middle of the process of the generic architecture and the concrete application, there should be a decrease in the granularity of the variation points identified in the architecture and the hotspots and hooks defined in the framework. In other words, if the framework could not solve a particular variation point, the hooks and hotspots could not be more general or allow for more flexibility than the original variation point. These ideas were later confirmed when we analyzed each variant feature and its associated variation points. In particular, the domain-specific adaptation was a clear example of a variant feature being partially solved by the framework, but where some issues were left open to complete according to the concrete application.

The memory-aid example was different from the previous one in the sense that was a concrete application with a defined functionality. In the finished application the only variant feature left open was the sensing support, since we use a simulated set of sensors that just fed the required types of actions. If we were willing to use external applications as software sensors this variant feature should be analyzed again and the proper sensors added to handle new information feeds.

We will now review the four variant features presented in Section 3 and show how they were solved:

- **Dynamic context model**. In our approach having a defined context model and simple design guidelines is a must. We have shown that both examples fit our aware object/context feature metaphor in a natural way. As we outlined in the requirements, the context model variants were closed in the design phase, but the context shape could be changed at run-time. Thanks to this approach we can build flexible context models in an object basis, updating them in run-time if required (e.g. adding and removing the location feature in the memory-aid application).
- **Legacy bridge**. The examples shown in this section were not extensions of existing applications and did not require an underlying application model. Although our aware model concept and its related functionality have been tested and simple prototypes were built, we still have to extend an existing legacy application to show its real power.
- **Sensor support**. This single variant feature was converted in three main variation points by identifying and encapsulating the different steps involved in the acquisition and transformation of external data into context information. By doing so we can provide a flexible architecture and highly reusable compo-

nents. In our examples, using simulated sensors instead of "real" ones was straightforward; however, we are aware that using physical sensors (e.g. a Smartphone's built in GPS) will raise new issues.

- **Domain-specific adaptation**. As we emphasize through the paper, we consider that the context model and the actions taken based on the context information should be engineered separately. The domain-specific model is conceived as a layer on top of the context model and is treated as a first-class object, where its specific behaviour is defined. As a result, domain-specific adaptations can be engineered independently from the application domain, easing the design and maintenance of the entire system. This issue has been extremely helpful to understand and design new scenarios and to isolate them from other requirements. A clear example of this is the Friend Finder application, built with the LBSs framework; by isolating it from any application model it can be easily reused to (for example) build an application to locate doctors in case of an emergency in a hospital.

As stated in Section 1, this article addresses three major issues, which we review in the context of the two examples presented earlier:

- **Identifying the most relevant variant features and associated variations points for different mobile context-aware domains**. In the examples we have shown four different applications in the LBSs domain and one concrete memory-aid application. Our proposal of variant features and variation points was successfully used to solve all the examples.
- **Define a set of micro-architectures that allows us to instantiate different products by implementing its unique features**. The examples presented earlier were solved by tackling one problem at a time in a given domain. By using our small-grained building blocks (aware objects, context features, adaptation environments, etc.) we could concentrate on the context model, sensing support and adaptation logic in separate stages, which are the distinctive parts of each application.
- **Show how these constructs are integrated in a sound architecture**. Both the LBSs framework and the memory-aid application were conceived by implementing the concerns related to their particular domain (e.g. the UserAction hierarchy in the memory-aid application) and connecting those classes to the proposed architecture. This connection was done by solving the proposed variation points with our building blocks; in some cases the connection was performed by subclassing (e.g. when creating new adaptation environments) while in other cases by composing existing components (e.g. the lookup transformation).

As we stated in Section 3, we consider that variability is managed by achieving a balance between flexible software and concrete needs and we think that our architecture achieves this balance. Of course this does not mean that adding a new requirement (e.g. a new sensing device) does not pose any challenges; accommodating new requirements or changing existing ones will have an impact on the system. The important issue in this situation is that the effort made to accomplish the change should be concentrated in the new requirement (e.g. learning the new sensor's API) and not in "gluing" the new requirement with the existing system (e.g. changing the context model to adapt to the new sensor). The efforts in our variability characterization and architecture were oriented in that direction: isolating the main variant features so that changes will not be spread across the entire system. This helps designers and developers by allowing them to focus on one thing at a time.

We consider that it is not casual that those variant features represent the main concerns that are usually mixed in any mobile context-aware application (context model, sensing and adaptation behaviour). To avoid this mixing our architecture relies on two basic principles: separating the concerns in different packages (so that they can be engineered individually), and using a notification mechanism to connect these packages (so that the models of each package are loosely-coupled). The LBSs framework is a good example of this case, where the required components are designed with very little coupling:

- The context model was designed independently of the services and the sensing devices. The only requirement for the context model is to define a location context feature.
- The location context feature is not aware of how it is sensed; this allows us to change the simulated location events with real GPS records without changing either the context model or the service environment. The sensing stage will of course need to be adapted for using the real GPS signal (for example, by adding a new dispatcher to handle DOP), but the change is confined to the sensing package and not spread throughout the system.
- The only requirement for an object to be registered in a LBSs environment is to have a location feature. Notice that this object can be an aware object specially conceived for a Friend Finder, or an aware model that adds context information to an object of a legacy application. From the point of view of a service, there is no difference between these two objects.
- Moreover, the LBS environment is not affected if the context model of an object registered in it has other context features, or if it is constantly changing its context shape. Environments are only dependent of those context features they explicitly ask for.

Thus, adding a new application that provides LBSs will almost have the complexity of the service itself (which can be a simple location-based messenger or a complex resource-tracking system) since the other parts of the system (like the context model) have already been worked out.

On the downside, our approach was conceived for building context-aware applications that change their behaviour according to the context. This means that, while it can be used for applications focused on context-dependent data retrieval, it may result in an over-kill. For this kind of applications, approaches based on context-dependent data modelling or extended relation models (Grossniklaus, 2007; Roussos et al., 2005) may be better suited. Also there is no current facility for structured reasoning in our context model. As we will mention in the next section, a pending issue in our architecture is to support context ontologies.

8. Concluding remarks and further work

In this paper we have presented the main variant features of mobile context-aware applications together with an architecture and a concrete implementation of those components that are common to most context-aware applications. In our characterization we have made a clear distinction between application and adaptations domains. We have shown that, since different adaptation domains can be applied to the same application domain (and vice versa), a general framework that encompasses any application domain, and any adaptation domain is very difficult to build. Instead of following this approach, we decided to isolate the main concerns in every context-aware application (context modelling, sensing and adaptation) and discovered the set of small-grained, micro-architectures that can be used as standard building blocks. The practical result of this research is an architecture, that allows

developers to concentrate on specific concerns (e.g. context modelling) whose evolution have a low (or null) impact on others since they are loosely-coupled. Even though we used the variability concepts to characterize our architecture, we must remark that we have not yet achieved the maturity level of a configurable product family (Deelstra et al., 2005); we actually consider our architecture to have reached the platform level. In order to reach the next level (i.e. a software product line), we would need a larger set of frameworks for specific adaptation domains, effectively sharing adaptation functionality across applications.

Even tough we consider that the abstractions presented in this paper are powerful, we are aware of the fact that environmental and toolkit support is a must. For this reason, a key research issue we are pursuing is to develop a platform to support the construction of context-aware software. This platform should help the developer to rapidly prototype applications, by manipulating predefined context features, sensing concerns and adaptation environments, organized as reusable catalogues. We are also working on advanced and flexible simulator tools to test our applications.

At the sensing level, we are looking forward to incorporating ontologies to simplify automatic sensor discovery. In this scenario, a context feature would export the kind of information it needs, while each sensing concern would provide a high level specification of the information it provides. Then, a discovery service would be in charge of matching providers and consumers, connecting them automatically when a change in the sensing hardware is detected (e.g. when entering into a new network).

We are currently working on improving the context modelling features; particularly, we are developing an extended version of the context model package that supports shared features between aware objects and "transitive relationships" (e.g. if the user is in the car, and the car has a GPS receiver to sense its location, then the user's location feature is the same as the car's location). We are also tackling the problem of a context feature being private to a specific adaptation environment; in this way we could specify when a context feature can be shared between domains, and when it is only accessible inside a specific adaptation domain.

Finally, we are studying how to deploy a core application as a standalone executable that communicates with the user by means of a web browser. In this case, our infrastructure would run its own dedicated web server, plus a (transparent) persistence engine, letting the developer focus only on the context-dependent behaviour. As a result, we would get both the power of directly accessing local resources (e.g. the GPS module of a Smartphone), and presenting the application in natural and usable way by using a web browser.

References

Abowd, G.D., Atkeson, C.G., Hong, J., Long, S., Kooper, R., Pinkerton, M., 1997. Cyberguide: a mobile context-aware tour guide. Wireless Networks 3 (5), 421–433. October.

Abowd, G.D., 1999. Software engineering issues for ubiquitous computing. In: Proceedings of the 21st International Conference on Software Engineering, pp. 75–84.

Apel, S., Böhm, K., 2005. Towards the development of ubiquitous middleware product lines. In: Proceedings of the ASE Workshop on Software Engineering and Middleware (SEM). LNCS, vol. 3437, Springer.

Bachmann, F., Bass, L., 2001. Managing variability in software architectures. In Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (Toronto, Ontario, Canada). SSR '01. ACM, New York, NY, pp. 126–132.

Bricon-Souf, N., Newman, C.R., 2007. Context awareness in health care: a review. International Journal of Medical Informatics 76 (1), 2–12.

Challiol, C., Fortier, A., Gordillo S.E., Rossi, G., 2008. Architectural and implementation issues for a context-aware hypermedia platform. Journal of Mobile Multimedia, Rinton Press, pp. 118–138, ISSN 1550-4646.

Cheverst, K., Mitchell, K., Davies, N., 2002. The role of adaptive hypermedia in a context-aware tourist GUIDE. Communications of the ACM 45 (5), 47–51.

Coplien, J., Hoffman, D., Weiss, D., 1998. Commonality and variability in software engineering. In IEEE Software 15(6), 37-45.

Davies, N., Cheverst, K., Mitchell, K., Efrat, A., 2001. Using and determining location in a context-sensitive tour guide. Computer 34 (8), 35–41.

Deelstra, S., Sinnema, M., Bosch, J., 2005. Product derivation in software product families: a case study. Journal of Systems and Software 74 (2), 173–194.

Dey, A.K., 2000. Providing architectural support for building context-aware applications. Ph.D. Thesis, Georgia Institute of Technology.

EC-ISTAG, 2001, Scenarios for Ambient Intelligence in 2010, Final Report.

Fayad, M.E., Schmidt, D.C., Johnson, R.E., 1999. Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, NY.

Feldmann, S., 2003. An indoor Bluetooth-based positioning system: concept, implementation and experimental evaluation. In: Proceedings of the International Conference on Wireless Networks, pp. 223–228.

Fernandes, P., Werner, C., Gresta, L., Murta, P., 2008. Feature modeling for context-aware software product lines. In: Proceeding of SEKE 2008, pp. 758–763.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley.

Google Mobile Maps. <http://www.google.com/mobile/default/maps.html>.

Grossniklaus, M., 2007. An object-oriented version model for context-aware data management. Ph.D. Thesis, Swiss Federal Institute of Technology.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM, 1984, ACM SIGMOD International Conference on Management of Data, pp. 47–57.

Harter, A., Hopper, A., Steggles, P., Ward, A., Webster, P., 2002. The anatomy of a context-aware application. Wireless Networks 8 (2–3), 187–197.

Hightower, J., Borriello, G., 2001. Location systems for ubiquitous computing. Computer 34, 8, 57–66.

Hirschfeld, R., Costanza, P., Nierstrasz, O., 2008. Context-oriented programming. Journal of Object Technology (JOT) 7 (3), 125–151.

Hodes, T.D., Katz, R.H., 1999. Compassable ad hoc location-based services for heterogeneous mobile clients. Wireless Networks 5 (5), 411–427.

Krasner, G.E., Pope, S.T., 1998. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming 1 (3), 26–49. August/September.

Lamming, M., Flynn, M., 1994. Forget-me-not: intimate computing in support of human memory. In: Proceedings of Symposium on Next Generation Human Interfaces.

Langley, R.B., 1999. Dilution of precision. GPS World 10 (5), 52–59.

Leonhardt, U., 1998. Supporting location-awareness in open distributed systems. Ph.D. Thesis, Dept. of Computing, Imperial College.

Marmasse, N., 1999. comMotion: a context-aware communication system. In: CHI '99 Extended Abstracts on Human Factors in Computing Systems CHI'99. ACM, New York, NY, pp 320–321.

Mattsson, M., Bosch, J., 1997. Framework composition: problems, causes and solutions. In: Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems.

Modahl, M., Agarwalla, B., Abowd, G., Ramachandran, U., Saponas T.S., 2004. Toward a standard ubiquitous computing framework. In: Proceedings of the Second Workshop on Middleware for Pervasive and Ad-Hoc Computing, pp. 135–139.

Myllymäki, T., Koskimies, K., Mikkonen, T., 2002. On the structure of a software product-line for mobile software. In: Software Infrastructures for Component-Based Applications on Consumer Devices, pp. 85–91.

Pascoe, J., 1997. The stick-e note architecture: extending the interface beyond the user. In: Proceedings of the Second International Conference on Intelligent User Interfaces, pp. 261–264.

Patel, S.N., Truong, K.N., Abowd, G.D., 2006. PowerLine positioning: a practical sub-room-level indoor location system for domestic use. In: Proceedings of Ubicomp 2006, pp. 441–458.

Pederson, T., Ardito, C., Bottoni, P., Costabile, M.F., 2008. A general-purpose context modeling architecture for adaptive mobile services. In Song, I.-Y. et al. (Eds.), ER Workshops 2008. LNCS, vol. 5232, Springer, pp. 208–217.

Priyantha, N.B., Chakraborty, A., Balakrishnan, H., 2000. The cricket location-support system. In: Proceedings of the Sixth Annual international Conference on Mobile Computing and Networking, pp. 32–43.

Quinlan, JR., 1986. Induction of decision trees. Machine Learning 1 (1), 81–106.

Rao, B., Minakakis, L., 2003. Evolution of mobile location-based services. Communications of the ACM 46 (12), 61–65.

Roussos, Y., Stavrakas, Y., Pavlaki, V., 2005. Towards a context-aware relational model. In: International Workshop on Context Representation and Reasoning, Paris.

Salifu, M., Nuseibeh, B., Rapanotti, L., 2006. Towards context-aware product-family architectures. In: Proceedings of the International Workshop on Software Product Management, pp 38–43.

Schmidt, A., Beigl, M., Hans, W.H., 1999. There is more to context than location. Computers and Graphics 23 (6), 893–901.

Schilit, B.N., 1995. A context-aware system architecture for mobile distributed computing. Ph.D. Thesis, Columbia University.

Schilit, B.N., Hilbert, D.M., Trevor, J., 2002. Context-aware communication. IEEE Wireless Communications 9 (5), 46–54.

Sousa, J.P., Garlan, D., 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments. In: Proceedings of the IFIP 17th World Computer Congress – Tc2 Stream/Third IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, pp. 29–43.

Strang, T., Popien, C.L., 2004. A context modeling survey. In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 – The Sixth International Conference on Ubiquitous Computing.

Svahnberg, M., van Gurp, J., Bosch, J., 2005. A taxonomy of variability realization techniques: research articles. Software Practice Experience 35 (8), 705–754.
van Gurp, J., Bosch, J., Svahnberg, M., 2001. On the notion of variability in software product lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture.
Weiser, M., 1999. The computer for the 21st century. SIGMOBILE Mobile Computer Communication Review 3 (3), 3–11.
Woolf, B., 1994. Understanding and using value models. <http://c2.com/ppr/vmodels.html>.
Windows Mobile Intermediate Driver, 2006. <http://msdn.microsoft.com/en-us/library/ms850332.aspx>.
Zloof, M., 1977. Query by example. IBM Systems Journal 16 (4), 324–343.

Andrés Fortier is a PhD student at Facultad de Informática, Universidad Nacional de La Plata (UNLP), Argentina where he also got his degree. He holds a CONICET grant for his PhD and is also teaching assistant at the UNLP. His research interests include Context-Aware Application Development and Object Oriented Languages.

Gustavo Rossi is Full Professor at Facultad de Informatica, Universidad Nacional de La Plata and researcher of CONICET, Argentina. He holds a PhD from PUC-Rio, Brazil. His current research interests include Agile approaches in Web Engineering and Context-Aware Software Development.

Silvia Gordillo is Full Professor at Facultad de Informatica, Universidad Nacional de La Plata and researcher of CICBA, Argentina. She holds a PhD from Université Claude Bernard-Lyon I, France. Her current research interests include Geographic Information Systems and Mobile Computing.

Cecilia Challiol is a PhD student at Facultad de Informática, Universidad Nacional de La Plata (UNLP), Argentina. She got her degree at the same University. She is a teaching assistant at the UNLP. Her research interests are Physical Hypermedia and Mobile Application Design.