Aspectual Decomposition in the Design of a Framework for Context-Aware User Interfaces

Montserrat Sendín, Jesús Lorés

GRIHO: HCI research group
University of Lleida, 69, Jaume II St., 25001- Lleida, SPAIN
Tel: +34 973 70 2 700 Fax: +34 973 702 702
{msendin,jesus}@eup.udl.es

Abstract. In mobile computing scenarios, information should be available every time, everywhere. Designing these kinds of systems results in a very complex task because of the high number of *crosscutting concerns* inherent to context-aware systems that must be modeled and automatically processed. The combination of Aspect Oriented Programming (AOP) with new metadata facilities arises as a powerful set of tools that makes possible to map the system's non-core concerns to *aspects* seamlessly. In this paper we propose some design strategies to design a client-side generic framework for context-aware user interfaces aimed at serving the mobile software community.

1 Introduction

Nowadays technology allows users to keep on moving with computing power and network resources at hand. Computing devices are shrinking while wireless communications bandwidth keeps increasing. These changes have increasingly enabled access to information "anytime, anywhere and by everyone". As a result, requirements for dynamic and real time adaptation become more and more demanding. Consequently, the design and development of User Interfaces (henceforth UIs) that obey to changing users' demands has become increasingly complex. Current interactive systems for mobile scenarios should be prepared to face up and accommodate this continuous and diverse variability. They should be provided with adaptive capacities, able to evolve as the *real time constraints* vary [9]. The aim is to provide continuous adaptation to the UI –appearance, contents, and even functionality to be offered.

Nevertheless, the *real time constraints* related not only to resources -server availability, device physical restrictions-, but also others related to the user –incremental preferences and needs-, and even to the environment –position, daylight-, are volatile and require sophisticated adaptive capabilities that today are still challenging.

This problem is well known in the HCI (Human Computer Interaction) community as *Context-Awareness*. Since it was proposed about a decade ago, many researchers have studied this topic and built several context-aware applications. Context-aware systems, however, have never been widely available to everyday users. It is clear that context-awareness field is an old but rich area for research [1].

In particular, it is needed some kind of software infrastructure to support context-aware systems. We refer to a runtime adaptive engine with capacity to detect the context and react appropriately. An engine in charge of applying transparently the necessary adjustments to the UI and thereby adapting to contextual changes on the fly. We have named it *implicit plasticity engine* (henceforth IPE) in our previous works [7,8].

The major difficulty for building this engine is dealing with the multiplicity of real time constraints inherently related that a context-aware interactive system must control and process, which generally entangle each other. Concerns that obey these features are commonly called *crosscutting concerns*. They even present diverse decomposition dimensions, rebounding in a major increase of complexity. Moreover, the way to effectively process that information is still a challenging problem for developers of this type of systems. In [8], a reflection-based approach is presented in order to modify application's behaviour to adapt it to changing network conditions. It allows performing self-modifications of existing behaviour, making possible to separate system and adaptation mechanisms. However, this separation of concerns is far from ideal. We propose AOP as a way to enhance the independence not only between these two issues, but also between the different concerns that need adaptation, obtaining thus the suitable orthogonality to reach reusability. An aspect-based adaptation equally provides self-modifications in the UI, but also reduces coupling and enhances reusability.

For some years, our research group has been involved in a project focussed on the cultural heritage area. We have developed different prototypes to assist the visit of an archaeological site. At present, we are working in an upper prototype to offer higher adaptation and extend its context-aware functionality beyond the localization issue. We are also working in a tele-aid system for high-mountain rescue and in a personalization module to be implanted in a digital newspaper archive. Our final goal is to generate a generic framework to easily derive the suitable IPE for a particular system. With this aim, we are determined to apply the most orthogonal design strategies to solve the most challenging design requirements. Here we outline the approach, general structure, design strategies and main guidelines to generate it.

2 Initial Considerations for Orthogonality

2.1 Design Requirements and General Structure

To generate a generic framework we must obtain the following properties: transparency in adaptation and reusability to different families of systems, different needs of context representation and different adaptation mechanisms. To reach these goals, it is crucial that the adaptive mechanisms and the system core functionality be handled orthogonally. Adaptive capabilities must also be isolated from each other, to avoid conflicts and to evolve individually. As it has been exposed in our previous work [9], we need to divide our framework into three layers. The *logical layer* contains the application core functionality. The *context-aware layer* contains the control and modeling of the real time constraints: the contextual model. This layer carries out the con-

text detection, maintaining information regarding the context for further use. Finally, there is an intermediary layer, responsible for doing the adaptation: an *aspectual layer*.

As it has been mentioned, the real time constraints constitute crosscutting concerns. According to the approach presented in [7], and as it has been proposed in our previous works [8,9], we use AOP [4] to integrate adaptation mechanisms for real time constraints in the system operation. We model them as *aspects* that intercept the operative of the core application to apply the suitable adjustments to the UI, according to the current state of the contextual model. The *aspectual layer* acts as a transparent link, reflecting the contextual model state in the UI along performance.

2.2 Why Aspects and Metadata Are a Good Combination?

A naive use of *aspects*, i.e. using *pointcuts* based on the method signature would turn out a system-specific aspectual design. This approach would generate strong coupling between the *logical* and *aspectual layers*. Even if we applied the *participant pattern* [5], we would be able to reduce the coupling from N-to-1 to a minor dependency, but it is still far from ideal. We need to capture *join points* in a generic manner.

We propose to use a metadata-based signature to capture the required *join points*. These *join points* will be the methods in the core application that carry just a simple metadata annotation expressly supplied. We propose this approach as the most suitable to reach two opposed goals: minimizing the impact –need of recoding- in the core system, and minimizing coupling with the system, gaining that way reusability.

At the moment this combination of programming techniques is currently available in the Java world. Particularly, the J2SE 5.0 version brings a new metadata facility allowing a powerful combination of metadata and AOP, mutually beneficial. As AOP language we have chosen AspectJ. It provides the so-called metadata-based *pointcuts*.

3 Putting Aspects and Metadata to Work

To illustrate the design option we propose for the *aspectual layer*, we have chosen a concern typically found in context-aware applications: the localization concern. We will refer to it as the `Localization` *aspect* (a metadata-based *aspect*). Suppose the underlying core application encapsulated in a unique class: the `coreApplication` class. Every method needing localization management would be supplied by a *Localizable* type annotation. The `Localization` aspect, thus, would define a *pointcut* to capture all the methods in classes that carry this kind of annotation. See figure 1.

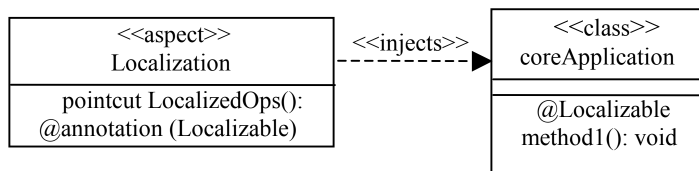

Fig. 1. Localization aspect diagram

Listing 1 shows a sketch of the *aspect* that would define the management and adaptation mechanism related to the localization concern in a particular system.

```
1  public aspect Localization {
2   boolean LocationChanged() {. . .}
3   - other method definitions
4   public pointcut LocalizedOps():execution(@Localizable
5   * coreApplication.*(..));
6   public pointcut LocalizedWholly(coreApplication core)
7     :if (LocationChanged())
8     && target(core);
9   Object around(): LocalizedOps() { -- the associated advice }
10  Object around(coreApplication core)): LocalizedWholly(core){
11    try {
12     loc=EnvironmentModel.getLocation();
13     String st=adaptView(loc); mountView(st); proceed(st);
14    } catch (Exception ex) {. . .}  } }
```

Listing 1. Localization aspect code

`Localization` *aspect* listed above defines two *pointcuts*. *Pointcut LocalizedOps* (lines 4-5) is the one that captures the execution of any method of the core class carrying the *Localizable* annotation. *Pointcut LocalizedWholly* (lines 6-8) is a conditional check *pointcut*. It captures any *join point* occurring after the condition expressed in `LocationChanged()` method[1] (line 2) evaluates to true. The aim is to automatically display the presentation corresponding to the user's location every time the user moves from zone to zone. Thus, the corresponding *advice* (lines 10-14) adapts the view before displaying it. Finally, it leaves the system code to proceed normally. Listing 2 shows a sketch for the core class.

```
public class coreApplication {
  @Localizable
   public void method1() {. . .}
  @Localizable
   public void method2() {. . .}    . . .}
```

Listing 2. The coreApplication class with annotations

The impact in classes is limited only to metadata attached to program elements. However, it is quite common that most of the methods in a class need to carry an annotation. Further, many systems require various annotations, leading to many annotations per method, a phenomenon known as *annotation clutter* [6]. That can be improved doing a refactoring step, using a special kind of *aspect* called *annotator aspect* [6]. Its goal is to encapsulate the annotations to be supplied in the core system. Listing 3 shows the *annotator aspect* for the `coreApplication` class.

```
public aspect coreAppAnnotator{
   declare annotation: public
       coreApplication.*(..):@Localizable;   }}
```

Listing 3. Localizable annotations for CoreApplication

With the *annotator aspect* approach, the only "glue" to attach the adaptation mechanisms to the base system is reduced to a declarative section, which is encapsulated in an *annotator aspect*. If we embedded this *annotator aspect* in each class,

[1] Inside this method, the aspect could check the state of data members expressly introduced by the aspect in the underlying classes. The use of the `if` *pointcut* improves the option of using the *pointcut method*.

following the *participant pattern*, as shown in [6], classes would require recoding. A better option is to extract *annotator aspects* from classes, joining them in the *aspectual layer*. Besides, it makes more sense to define a only annotator aspect for the core system that one per class. Figure 2 depicts a sketch of an IPE for a simple application. The *EnvironmentalModel* component stores for further use the environmental factors.

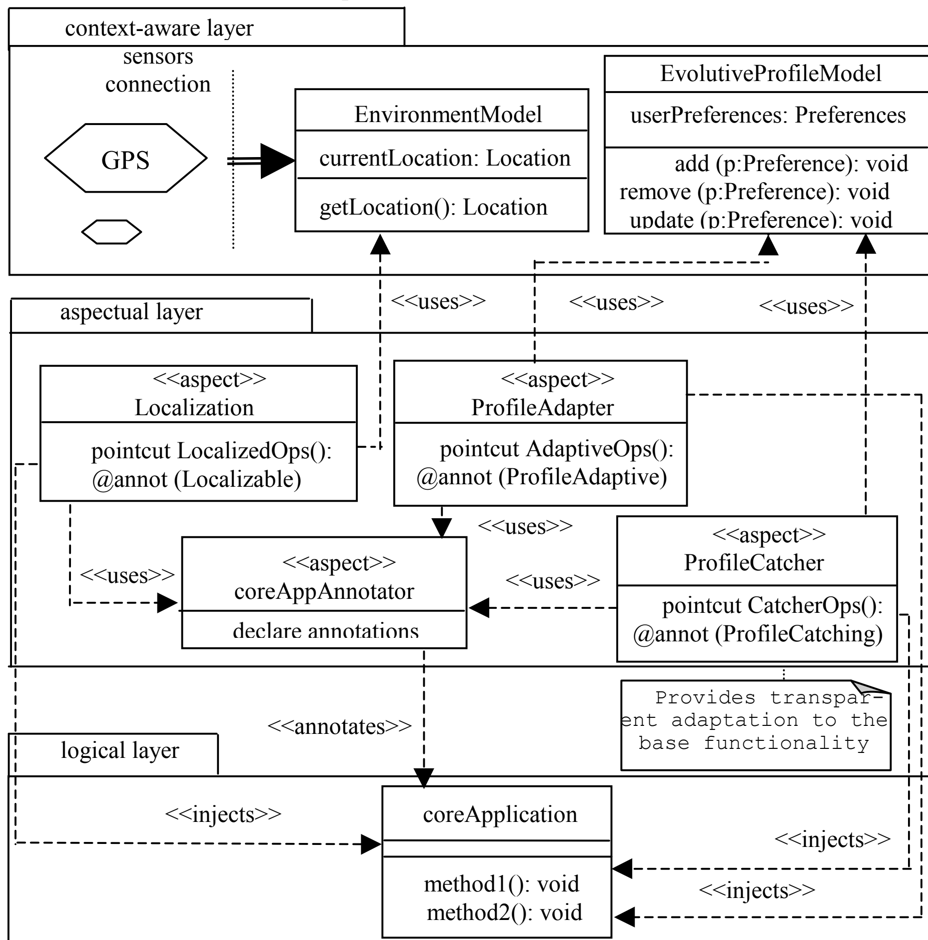

Fig. 2. IPE for a simple application

The main idea around the customization concern is to make user's customization features evolving. To undertake this issue we need a different treatment because the input, which needs to be incrementally stored, is not received from sensors. It comes from the execution of the application. That implies that the *aspectual layer* acts not only as an adaptive means, but also as a listener from the events and actions occurred in the core system. It must capture information about the user's preferences and interests, providing feedback to the dynamic user model: the *EvolutiveProfileModel*. This aim is assigned to the `ProfileCatcher` *aspect*. Then, the `ProfileAdapter` intercepts the core application in order to adapt the UI, according to this user model.

To solve *interaction of aspects* phenomenon we can appeal to *aspect precedence*.

4 Further Guidelines towards Abstraction

In the line of designing a generic and flexible framework to different issues, we can make the following considerations according to each issue.

Adaptation mechanisms. In the *aspect* definition in Listing 1, we can note some system-dependencies that considerably limit orthogonality and reusability. To offer flexibility in adaptation mechanisms, we can appeal to *aspect* hierarchy with abstract *aspects* at the top. Shall we outline some refactoring steps to surpass dependencies.

In particular, the system dependency in line 5 could be solved simply including the wildcard also in the class name. Despite of the core application is composed by a set of classes, this *pointcut* would only affect classes carrying the *Localizable* annotation.

However, it is not always so trivial. We might need to be more selective in the weaving stage –e.g. treating the localization concern only in one class, in spite of annotations have been spread through classes. Another example of specialisation could be requiring another *pointcut* type, instead of the "execution" one. Both cases would need to redefine the associated *pointcut* in *sub-aspects*.

Using *aspect* hierarchy, we can design an abstract *aspect*, and then choose between the following options: define abstract *pointcuts* –the case just referred-, leave the *sub-aspect* to define new *pointcuts*, refactor *advices*, or any combination of them.

- In the first case, *sub-aspects* extend abstract *aspects* specializing their *pointcuts*.
- Regarding the dependency in lines 6, 8 and 10, we need the second solution: to define a new *pointcut*, to also encapsulate the associated *advice* in the *sub-aspect*.
- On many situations, it is not necessary to define the complete *advice* in the *sub-aspect*, but only a method that encapsulates some special needs. Then, we can use *advice* refactoring. *Sub-aspects* would only redefine that method. In general, this strategy is particularly appropriate when the *advice* contains some variabilities from application to application. However, it could also be considered for different localization management in the same application. For example, in the second *advice* from Listing 1 (lines 10-14), instead of adapting the view (line 13), we could require for some classes to send a remote query to receive the view from a server. In this case, we would only need to refactor the *adaptView* method.

That is a smart way to specialize the code corresponding to the adaptation mechanisms. In particular, this idea corresponds to the *Template advice* idiom [3].

Of course, it can be used other types of refactoring, other types of emerging AOP-specific patterns and idioms to make good designed AspectJ applications [3]. For example, a common situation is when you have different *aspects*, which depict redundant code inside the application. We can use the *composite pointcut* idiom [2].

Domains of application. It is possible to deploy libraries of *aspects*. Thus, each particular application -even each particular use- is able to establish the set of concerns it needs to manage. That will determine which *aspects* need to be charged in memory. For example, in an archeological site it is required to consider the daylight constraint to adjust the UI. However, if we want to adapt this framework to an indoors museum guide, this concern is useless. Incidentally, in the tele-aid system another kind of concerns are required, such as the altitude and ascent speed, in order to assist high-mountain rescues. We could build an aspect package related to mountain conditions.

Contextual needs. Equally, we need to adapt the *context-aware layer* to the *aspectual* one, in order to map *aspects* with data stored in the contextual model. This is the reason why it is essential to obtain flexibility also in the *context-aware layer*. Flexibility in the contextual representation can also be obtained by means of classes hierarchy in the components contained in the *context-aware layer*.

5 Conclusions

Self-adaptive context-aware applications are exposed to a world where real time constraints change continuously. The design of these systems is prone to mismatching.

An *aspectual layer* in the adaptive framework is essential to reach a better separation of concerns for these kinds of applications. However, how attaching this layer to the base application is determinant to obtain the decoupling between layers we pursue. We present a metadata-based aspectual decomposition approach. We have proven that it causes the minimum impact over the base application obtaining, at the same time, a system-independent adaptation mechanism. Incidentally, arranging and deploying an appropriate hierarchical library of *aspects* would contribute to the flexibility and extensibility we pursue to adapt the framework to different domains of application.

Acknowledgements. This work is partially funded by the Spanish Ministry of Science and Technology, grants TIN2004-08000-C03-03.

References

1. Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Technical Report TR2000-381. Computer Science Department, Dartmouth College, Hanover (2000)
2. Hanenberg, S., Costanza, P.: Connecting Aspects in AspectJ: Strategies vs. Patterns. 1rst Worshop on Aspects, Components, and Patterns for Infrastructure Software (2002)
3. Hanenberg, S., Schmidmeier, A.: Idioms for Building Software Frameworks in AspectJ. 2nd Worshop on Aspects, Components, and Patterns for Infrastructure Software (2003)
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. Irwin, J.: Aspect-Oriented Programming. In: M. Aksit and S. Matsuoka (eds.): 11th ECOOP'97. Lecture Notes in Computer Science, Vol. 1241 (1997) 220-242
5. Laddad, R.: AspectJ in action. Practical Aspect-Oriented Programming. Manning Publications (2003)
6. Laddad, R.: AOP and Metadata: a perfect match. IBM DeveloperWorks (2005)
7. Mesquita, C., Barbosa, S.D. J., De Lucena, C.J.P.: Towards the identification of concerns in personalization mechanisms via scenarios. Proc. of Workshop on Early Aspects (2002)
8. Periquet, A.I., Lin, E.: Mobility Reflection: Exploiting Mobility-Awareness in Applications by Reflecting on Distributed Object Collaborations. Technical Report 97-CSE-6, Southern Methodist University (1997)
9. Sendín, M., Lorés, J.: Local Support to Plastic User Interfaces: an Orthogonal Approach. Selection of HCI related papers of Interacción 2004. Springer-Verlag (2005) in press