Implicit Plasticity Framework: a Client-Side Generic Framework for Context-Awareness

Montserrat Sendín

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; everyone. Designing these kinds of systems results in a very complex task due to the high number of concerns inherent to mobile systems that must be modeled and automatically processed. Furthermore, due to the manner in which these concerns interact each other, they are sentenced to hopelessly get mixed and at the same time crosscut the system core functionality –they are commonly called *crosscutting concerns*. The combination of Aspect Oriented Programming 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 present some design strategies to develop a client-side generic framework for context-aware user interfaces aimed at serving the mobile software community. We call it *Implicit Plasticity Framework.*

1 Introduction

With the advent of ubiquitous and mobile computing the design and development of User Interfaces (henceforth UIs) that really conform to changing user demands has become increasingly complex. Although much progress has been made in terms of technological innovation, many mobile interactive systems are difficult to use, lack robustness and, above all, lack flexibility with dynamically changing contexts of use. In addition, systems' design for mobile scenarios covers a wide range of issues; from mobile networking to mobile devices UI design. Moreover, mobile devices present significant graphical and interaction differences. Considering all the issues involved in mobile scenarios can be overwhelming. To master the diversity of contexts of use in an economical and ergonomic way, the *plasticity* property has been introduced.

The term *plasticity of UIs* was introduced by Thevenin and Coutaz from the IIHM group in 1999 [15], along a framework and research agenda development work. Today this framework is being object of continuous revision within the CAMELEON project. Some of the releases are the Revised Reference Framework [2], the CAMELEON Reference Framework [3] –a general tool for reasoning about adaptation, covering both recasting and redistribution-, and more recently the work in [4].

This last revises the notion of software plasticity, which is applied at the widget level in terms of "comets" (COntext of use Mouldable widgETs[1]).

Despite these advances in plasticity, any definition regarding a clear distinction between the static (design time) and dynamic (runtime) points of view of plasticity has been given to definitely isolate plasticity goals into two sub-concepts, as we propose in our *"dichotomic" view*[2] [12]. This distinction has only been noticed, although not exploited, in the CAMELEON Reference Framework. Later, in [1], it is recognized but not clearly specified the necessity of an external mechanism when the adaptation to the context of use can not be addressed autonomously.

What seems a close approach is the notion of *open* and *close adaptiveness* introduced by Oreizy et al. [11]. According to it, when the system includes all of the mechanisms and data to perform adaptation on its own, it is said to be *close-adaptive*. Consequently, *open-adaptiveness* implies that adaptation is performed externally to the system –total or partially. On the contrary, the two concepts from our dichotomic view fit well under a *close-adaptiveness* perspective; we assume self-adaptiveness.

Under our dychotomic view, each sub-concept -*explicit* and *implicit* plasticity- has a different goal clearly identified and delimited. They need to be studied and dealt with separately. They require different tools and modeling techniques. In this way we combine two different infrastructures framed in a client/server architecture to manage both issues alternative, iterative and complementarily in order to feed a process of plasticity without discontinuities (*close-adaptiveness* perspective). The *implicit plasticity* issue is to be solved in the mobile device (client-side) and the *explicit plasticity* in the server. See more details in [12]. Anyway, each one of these issues can also be exploited autonomously. This is the case when we have an only type of requirement of plasticity: adapting the UI either statically (design stage) or dynamically (runtime stage). That corresponds to an *open-adaptiveness* perspective. We call each of these frameworks *explicit plasticity engine* and *implicit plasticity engine*, respectively. This paper deals with the *implicit plasticity* problem, which is associated with a spontaneous adaptation to contextual changes on the fly. See more detail in [13]. It is required an engine capable of dynamically accommodating a specific UI to a continuous variability of contexts of use. We call it *implicit plasticity engine* (*IPE* henceforth).

However, it is clear that each system needs its particular engine. Hence, it is not too much significant to outline the IPE for a particular system. This would only show the approach we propose, being able to reuse little or none line of code. We want to go beyond. What we pursue is to develop a generic framework to easily derive a suitable IPE for each particular system. This is what we have named *Implicit Plasticity Framework* (*IPF* henceforth). This aim requires gathering the experience from different domains of application. Hence, our research group is involved in different projects. One of them is focussed on the cultural heritage area. We have developed different prototypes to assist the visit of an archaeological site called "Els Vilars". At

---

[1] An introspective interactor mold for adaptation that publishes the quality in use it guarantees for a set of contexts of use.

[2] View from the *plasticity* process as a dichotomy, that is to say, a separation into two sub-concepts of plasticity closely associated with the stages of design (static) and runtime (dynamic). We have named them *explicit plasticity* and *implicit plasticity*. They were defined and presented as an extension to the Thevenin and Coutaz term in [12].

present, we are working in an upper prototype to offer higher adaptation and extend its context-aware functionality beyond a naive localization management. 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. Taking advantage from all the experience collected, we are determined to apply the most orthogonal design strategies to solve the most challenging design requirements in order to gain the necessary reusability. In this paper we outline the approach, general structure, design strategies and main guidelines to develop it.

2 Initial Design Considerations

2.1 Design Requirements and General Structure

To develop the IPF we pursue, we must guarantee the following properties: transparency in adaptation and reusability to different families of systems, different needs of context representation and different adaptation mechanisms. It is quite common to find a set of concerns recurrent in a lot of application domains. To reach these goals, it is crucial that the adaptive mechanisms and the system core functionality be handled orthogonally, so that they can evolve individually. Adaptive mechanisms should also be isolated from each other to avoid conflicts and promote reusability. Orthogonality is especially important in mobile software, where a lot of dimensions are present.

As it has been exposed in our previous work [14], we conceive an *IPE* as a software architecture divided 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 context detection, maintaining information regarding the context for further use. Finally, there is an intermediary layer, responsible for doing the adaptation: the *aspectual layer*.

As it has been mentioned, the real time constraints constitute *crosscutting concerns*. According to the approach presented in [10], and as it has been presented in previous works [13,14], we use Aspect-Oriented Programming (AOP henceforth) [8] to integrate adaptation mechanisms for real time constraints in the system operation. We model them as *aspects* that intercept the operative of the core system to apply the suitable adjustments to the UI, according to the current state of the context. The *aspectual layer* acts as a transparent link between the other layers, reflecting the contextual state in the UI along the system performance.

2.2 Aspect-Oriented Programming

A crosscutting concern is a concern that is inevitably spread along most of the modules of a system. AOP [8] is one of a lot of separation of concern technologies resulting from the effort to modularize crosscutting concerns. It was proposed by

Kiczales et al., (Xerox PARC) in 1990, in an attempt to surpass the increasing complexity of software systems, and was named AOP in 1996. The intuitive notion of AOP comes from the idea of extracting and encapsulating problematic extra-functional concerns (*crosscutting concerns*), providing modularization to their whole treatment in different program units called *aspects*. This approach is especially applicable to Mobile Computing, Ubiquitous Computing and Context-Awareness, where multiple factors, which become tangled each other, should be handled.

AOP includes a set of additional concepts like:

Join Point: An identifiable and well-defined point in the program flow. For instance, a method call.

Pointcut: Programming construct to establish the *join points* that require some type of treatment. *Pointcuts* capture them in the program flow and collect their context.

Advice: The code to execute when *join points* are reached and captured by the associated *pointcut*. We refer to the code responsible for handling crosscutting concerns.

2.3 Why Aspects and Metadata Are a Good Combination?

As stated above, we use *aspects* to integrate the mechanisms of adaptation for real time constraints in the system operation. However, a naive use of *aspects*, i.e. using *pointcuts* (see section 2) 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*. We need to capture *join points* (see section 2) in a generic manner.

We propose to use a metadata-based signature to capture the required *join points*. These *join points* are the methods in the core application that carry a simple *metadata annotation*[3] expressly supplied. We propose this combination (*aspects* and metadata) as the most suitable to reach two opposed goals: minimizing the impact –need of recoding- in the core system, and minimizing coupling, promoting that way reusability.

At the moment, this combination of programming techniques is currently available in the Java world, which is the one we have selected in our implementation.

3 Putting Aspects and Metadata to Work

In this section we want to present the approach, general structure and design strategies we propose for a specific IPE, paying special attention to the *aspectual layer*. This step will allow us to lay the foundations in order to make an effort of abstraction towards the final IPF. Is in the next section that we collect the guidelines of abstrac-

---

[3] Metadata are annotations that mark particular fields, methods, and classes as having attributes that have no direct effect on the execution of the code and should be processed in special ways by development tools, deployment tools, or runtime libraries.

tion we are considering to build it. To illustrate all that, we will consider an elemental application that only has a typical context-aware concern: the localization. We will refer to it as the `Localization` *aspect* (a metadata-based *aspect*). Furthermore, we assume that this application has adaptive (customization) requirements. Supposing the underlying core application encapsulated in a unique class: the `coreAppl` class, every method needing localization management would be supplied with a *Localizable* type annotation. The `Localization` *aspect*, thus, would define a metadata-based *pointcut* to capture all methods in classes carring this kind of annotation. See figure 1.

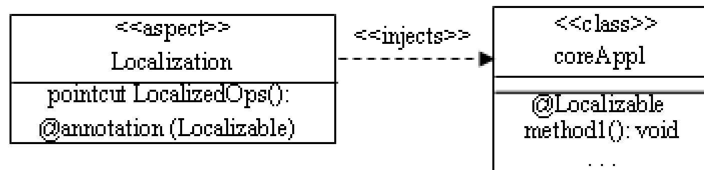

Fig. 1. Localization *aspect* diagram

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

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

List.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 6-7) is a conditional check *pointcut*. It captures any *join point* occurring after the condition expressed in `LocationChanged()`method (line 2) evaluates to true. The aim is to automatically display the presentation that corresponds 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 (line 13). Finally, it leaves the base code to proceed normally (line 14). Listing 2 shows a sketch for the core class.

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

```
@Localizable
 public void method2() {. . .}      . . .}
```

List. 2. The *coreAppl* 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* [9]. That can be improved doing a refactoring step, using a special kind of *aspect* called *annotator aspect* [9]. Its goal is to encapsulate all the annotations to be supplied to a class or system. Listing 3 shows the *annotator aspect* for the `coreAppl` class.

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

List. 3. Localizable annotations for *CoreAppl*

Figure 2 depicts a sketch of the IPE for our elemental application.

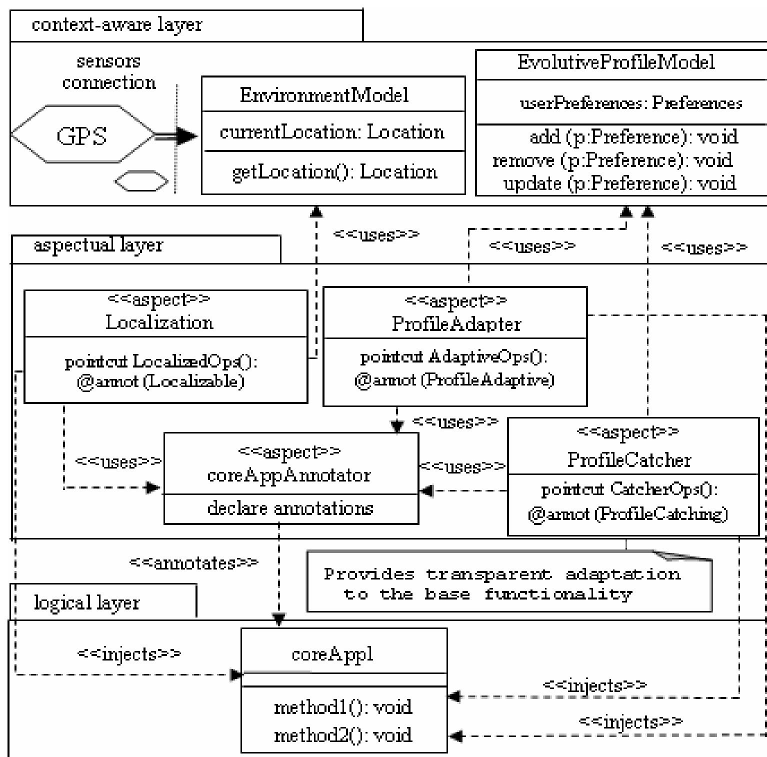

Fig. 2. IPE for an elemental application

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* and put in the *aspectual layer*, as another *aspect*. Hence, the *logical layer* becomes completely free from any annotation and track about *aspects*. Furthermore, the *aspectual layer* is the only responsible for doing the adaptation, and is who takes the initiative. So, the adaptation is carried out transparently.

The *EnvironmentalModel* in the *context-aware layer* stores for further use the environmental factors. In this case it only stores information about the localization.

The main idea around the customization concern is to make user's customization features evolving. We refer to adaptivity. 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*. The `ProfileAdapter` intercepts the core application in order to adapt the UI, according to this user model.

4 Further Guidelines Towards Abstraction

In the design of our IPF, according to the experience extracted from the IPEs built up to now, we have taken some considerations in order to obtain system-independence and reusability. As we intend to adapt our IPF to different adaptation mechanisms, contextual needs and domains of application, let us see them according to each issue.

Adaptation mechanisms. In the *aspect* definition in Listing 1, we can note some system-dependences that considerably limit orthogonality and reusability. To obtain "universal" adaptation mechanisms, we can appeal to *aspect* hierarchy.

Thus, for instance, the system dependence in line 5 (`coreAppl` in the LocalizedOps *pointcut*) 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 a specific class, in spite of annotations have been spread through a set of classes. Another example of specialization could be requiring another *pointcut* type, instead of the "execution" one. Both cases would need to redefine the associated *pointcut* in *sub-aspects*. In short, using *aspect* hierarchy, we can design an abstract *aspect*, and then choose between the following options: define abstract *pointcuts* to be defined by *sub-aspects* –the case just referred-, leave the *sub-aspect* to define new *pointcuts*, refactor *advices*, or any combination of them.

Regarding the dependence in lines 6, 7 (`coreAppl` in the LocalizedWholly *pointcut*) and 10 (the corresponding *advice*), we need the second solution: defining a new *pointcut* in the *sub-aspect* that would also encapsulate the associated *advice*.

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, either from different applications or, why not, from different localization managements in the same application. For example, in the second *advice* from Listing 1 (lines 10-14), instead of adapting the view (*adaptView* in line 13), we could require for some classes to send a remote query to receive the view from a server (e.g. an *explicit plasticity* server [12]). In this case, we would only need to redefine 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 [6].

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 [9, 5].

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 a tele-aid system another kind of concerns are required, such as the altitude and ascent speed, in order to assist mountain rescues. We could build an package of *aspects* 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 mobile and ubiquitous applications are exposed to a world where real time constraints change continuously. The design of these systems becomes highly complex and prone to mismatching. The integration of an *aspectual layer* for the IPE is essential to reach an appropriate separation of concerns for these kinds of constraints and to manipulate the underlying UI transparently. 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 that causes no impact over the base application and, at the same time, reduces system-dependence to the minimum, facilitating so greatly its integration with the base system. We assert that the adaptation mechanisms act seamless and transparently.

Finally, the great milestone for obtaining a generic IPF, reusable for any system, consists of removing totally system-dependences in the adaptation mechanisms (*aspectual layer*). This is tackled applying appropriate guidelines of abstraction to give the final framework the flexibility we pursue. We collect in this paper the considerations extracted up to now from experience. Once we have concluded our IPF, with the aim of offering it to the mobile community, we consider essential to arrange and deploy an appropriate hierarchical library of *aspects*, contributing so to the necessary flexibility and reusability and to adapt the framework to different domains.

Acknowledgments

Work partially funded by Spanish Ministry of Science and Technology, grants TIN2004-08000-C03-03.

References

1. Balme, L.: Infrastructure Logicielle pour Interfaces Homme-Machine Plastiques. Proc. of Secondes Rencontres Jeunes Chercheurs en Interaction Homme-Machine (RJC-IHM'04) (2004) 27-32
2. Calvary, G. et al.: Plasticity of User Interfaces: A Revised Reference Framework. Proc. of TAMODIA 2002 (2002)
3. Calvary, G., Coutaz, J., Thevenin, D. , Bouillon, L., Florins, M., Limbourg, Q., Souchon, N., Vanderdonckt, J., Marucci, L., Paternò, F., Santoro, C.: The CAMELEON Reference Framework. Deliverable D1.1 (2002) http://giove.cnuce.cnr.it/cameleon.html
4. Calvary, G., Coutaz, J., Dâassi, O., Balme, L., Demeure, A.: Towards a new Generation of Widgets for Supporting Software Plasticity: the "comet". Proc. of EHCI-DS-VIS, Hambourg (July, 2004)
5. Hanenberg. S., Costanza, P.: Connecting Aspects in AspectJ: Strategies vs. Patterns. 1rst Worshop on Aspects, Components, and Patterns for Infrastructure Software (2002)
6. Hanenberg, S., Schmidmeier, A.: Idioms for Building Software Frameworks in AspectJ. 2nd Worshop on Aspects, Components, and Patterns for Infrastructure Software (2003)
7. Herrmannm, S., Mezini, M.: PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments. Proc. of ACM OOPSLA 2000. Vol. 26, Issue 1, ACM Press (2001) 188-207
8. 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
9. Laddad, R.: AspectJ in action. Practical Aspect-Oriented Programming. Manning Publications (2003)
10. Mesquita, C., Barbosa, S.D. J., De Lucena, C.J.P.: Towards the identification of concerns in personalization mechanisms via scenarios. Proceedings of the AOSD 2002, Workshop on Early Aspects (2002)
11. Oreizy, P., Medvodovic, N., Taylor, R.N.: Architecture-Based Runtime Software Evolution. Proc. of the International Conference on Software Engineering (ICSE'98), Kyoto (1998) 11-15.
12. Sendín, M., Lorés, J.: Plasticity in Mobile Devices: a Dichotomic and Semantic View. Workshop on Engineering Adaptive Web, supported by AH 2004, Eindhoven (2004) 58-67
13. Sendín, M., Lorés, J.: Local Support to Plastic User Interfaces: an Orthogonal Approach. Selection of HCI related papers of Interacción 2004. Springer-Verlag (2005)
14. Sendín, M., Lorés, J.: Towards the Design of a Client-Side Framework for Plastic UIs using Aspects. Proc. of Internat. Works. on Plastic Services for Mobile Devices (PSMD) (2005)
15. Thevenin, D., Coutaz, J.: Plasticity of User Interfaces: Framework and Research Agenda. Proc. of Interact'99, Edinburgh (1999) 110-117