Desarrollo de Software Orientado a Aspectos, DSOA 2006
Asociado a XV Jornadas de Ingeniería del Software y Bases de Datos
J. Araújo, J. Hernández, E. Navarro y M. Pinto (Eds)
Sitges (Barcelona), Octubre - 2006

CONTRASTING ASPECTUAL DECOMPOSITIONS WITH OBJECT-ORIENTED DESIGNS IN CONTEXT-AWARE MOBILE APPLICATIONS

Montserrat Sendín and Jordi Viladrich

GRIHO: HCI research lab
Superior Polytechnic School
University of Lleida
69, Jaume II St., 25001- Lleida, SPAIN
e-mail: msendin@eps.udl.es, web: http://griho.udl.es/

Key words: Context-awareness, metadata annotation, adaptation mechanisms

Abstract. *Designing context-aware systems results in a complex task because of the high number of crosscutting concerns inherent to this kind of systems that must be considered and processed, as well as mobile devices constraints. It is essential to follow the most orthogonal approach to provide the necessary pluggability to cope with different contextual needs and adaptation mechanisms. Our solution consists of providing context-awareness to already existing not context-aware mobile solutions in a non-invasive way, using a combination of Aspect Oriented Programming (AOP) and metadata techniques. We pursue minimal-coupling and minimal system-dependent solutions in order to build the generic framework for context-awareness we are working on. We present the design developed for a particular case study in a mobile scenario. By providing three versions of the same application, we compare an aspectual version with a pure object-oriented and a patterned one, evaluating some measures related to the code size, a critical factor in the deployment of mobile applications.*

1. INTRODUCTION

Current interactive systems for mobile scenarios should be prepared to face up and accommodate the continuous and diverse variability inherent to mobility, in which the environmental constraints play a key role. We are referring to the well-known *Context-Awareness* problem[i] [1], proposed about a decade ago. To tackle context-awareness it is needed some kind of software infrastructure to provide proactive adaptive capacities, able to evolve as the *contextual constraints* vary [6]. We refer to a runtime adaptive engine with capacity to detect the context and react appropriately, adapting the underlying system on the fly. We call these kind of engines *Implicit Plasticity Engines* (IPEs henceforth) [5, 6].

[i] Capacity of triggering automatic adaptations caused by changes in the application context.

According to our approach [6], these engines consist of a software architecture divided into three layers: (1) the *logical layer* (the business code); (2) the *context-aware layer*, (the contextual model and sensor controllers) and (3) the *aspectual layer* (adaptive mechanisms).
The major difficulty for building these engines is dealing with the multiplicity of contextual concerns to control and process, which entangle each other. The way to effectively process that information is still a challenging problem for developers of this type of systems. We use a combination of AOP and metadata techniques, which provides a level of orthogonality so high that allows conceiving the different contextual concerns as easily pluggable components, reducing the context-awareness problem to a simple pluggability[ii] mechanism.
In this paper we focus on a case study and we present some design and implementation details of an IPE built to provide a common mobile application with context-awareness. We prove that the IPE is integrated with the business code in a seamlessly way. We compare the aspectual version for this engine with a pure object-oriented one and a patterned solution using GoF, bringing some experimental results according to the extension of the source code and the size of the deliverable code. We have chosen these parameters because the bytecode size is a critical factor in mobile applications, to the extent of being determinant in the deployment in real mobile appliances. The reasons why we have chosen these implementations are: (1) demonstrate that contextual concerns are crosscutting concerns; (2) measure how the contextual concern evaluated scatters with the business code and the effects in the code size in object-oriented solutions; (3) contrast if a patterned version reduces or not significantly these problems, with the aim of definitively demonstrating that an aspectual version is the best design solution for mobile context-aware applications.
Finally, following with our goal of building a generic framework to derive particular IPEs (we call it *Implicit Plasticity Framework* –IPF-), we present the AOP-specific idioms we consider beneficial to be adopted to configure the IPF [5], on the way of learning from experience.

2. TECHNICAL APPROACH: ASPECTS AND METADATA COMBINATION

As it is well-known, one of the main new language constructions introduced by AOP is the *joinpoint*, which defines an interception point in the execution flow. If we base the definition of these points on a simple method signature, we will turn our aspectual design into a system-specific high-coupled one. This would be to the detriment of reusability.
To capture *joinpoints* in a generic manner we use the metadata-based signature. It consists of capturing as *joinpoints* the methods in the core application that carry just a simple metadata annotation expressly supplied. Thus, associating annotations strategically in methods and classes needing contextual management and defining *joinpoints* based in these annotations, we manage to avoid knowing about the business code details. However, the impact in classes can be overwhelming -a phenomenon known as annotation clutter [4]. This inconvenient is solved using a special kind of *aspect* called *annotator aspect* [4]. Its goal is to encapsulate and extract the annotations to be supplied in the core system, relieving so the latter from having to embed expressly these annotations.

[ii] Facility to generate different versions of the same application depending on changing contextual needs.

With this strategy, the only "glue" to attach the adaptation mechanisms to the base system is reduced to a declarative section that gathers all the required annotations for the business code (the *annotator aspect*). Definitively, the *logical layer* does not need recoding at all. This approach allows reaching two opposed goals: removing the impact in the core system, and minimizing coupling between the *aspectual* and the *logical layer*. This contributes to gaining reusability and to obtain system-independent designs. As a result, we can say that this approach is not invasive. At the moment this combination of programming techniques is available in the Java world (J2ME and AspectJ5.

3. DESIGNING AN IPE FOR A CASE STUDY

3.1. Our Case Study: the NewsReader

The main goal of this application is to extend the media to deliver daily news from a local digital newspaper provider to a considerable sector of people, through the mobile phone.
The main functionalities for this application are the downloading, management and storage of the set of news that match the particular needs of the final user.
In order to provide this application with context-awareness support, we have chosen a universal contextual concern: the brightness, thinking in mobile and changing scenarios. The aim is dynamically adapting the screen brightness to different external daylight levels whereas reading news on the move.

3.2. Design and Implementation Details

First of all, as we need to adapt the brightness level to each user interaction (commands and widgets events), the methods that need brightness management are specifically the `commandAction` and the `itemStateChanged` methods of the `NewsReaderMIDlet` main class. So, annotations (*Lightable* in Listing 1) have to be supplied to these methods to guarantee an appropriate brightness level when the user interacts with the application. Listing 1 shows the *annotator aspect*, which associates, encapsulates and extracts these annotations.

```
public aspect NewsReaderAnnotator {
   declare annotation: public
      NewsReaderMIDlet.commandAction(..):@Lightable;
      NewsReaderMIDlet.itemStateChanged(..):@Lightable;}
```

Listing 1. Lightable annotations for the NewsReaderMIDlet class

The *aspect* that treats the concern implemented is called `Brightness` *aspect* -a metadata-based *aspect*. Listing 2 shows a sketch that defines the management and adaptation mechanism related to our case study. Let us explain it with some detail.
Following the metadata-based signature approach, there is a *pointcut* referring the *Lightable* annotation (*LightedOps;* line 4) that captures methods supplied with this annotation. The corresponding *advice* (line 9) is responsible for capturing a new sensor reading and processing the updating. The `BrightMonitor` class models the brightness sensor controller.
In order to assure a periodic monitoring of the daylight sensor and so contribute to

proactivity, we use a thread (`BrightnessThread`). It starts running at the beginning of execution. This is the goal of the *LaunchingMonitor pointcut*, intercepting the midlet constructor (line 5). Changes detected by this thread are expressed as another *joinpoint* by the *set* designator (*pointcut* LightedWholly; lines 6-8). The corresponding *advice* (lines 12-13) is in charge of doing the appropriate adjustment in the screen brightness, according to the current external daylight. `DeviceControl` is a class from the Nokia UI API[iii] (line 13).

```
1 public aspect Brightness {
2   private static BrightMonitor BrightContler = new BrightMonitor;
3   private static int prevBness = 0;
4  pointcut LightedOps(): execution(@Lightable * *.*(..));
5   pointcut LaunchingMonitor(): execution(* NewsReaderMIDlet());
6   pointcut LightedWholly(int curBness):
7     set (private int BrightContler.lvlBrightness) &&
   args(curBness)
8     && if ((curBness>prevBness+THRES)||(prevBness<curBness-
   THRES));
9   before(): LightedOps() { BrightContler.updateBness(); }
10  before(): LaunchingMonitor() {
11    Thread t = new BrightnessThread(); t.start();  }
12  after(int curBness): LightedWholly(curBness) {
13    DeviceControl.setLights(0,curBness); prevBness = curBness;  }}
```

Listing 2. Brightness *aspect* code

Figure 1 depicts a sketch of the IPE described for our case study.

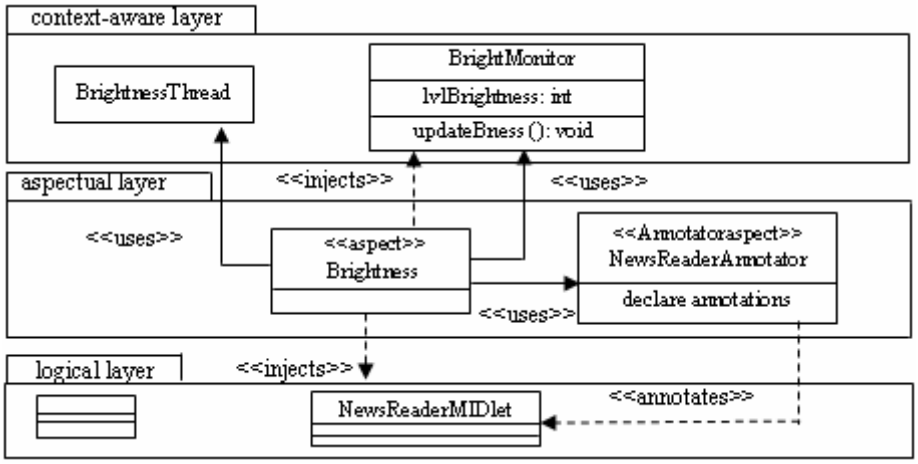

Figure 1. IPE described for the NewsReader application

[iii] forum.nokia.com/java. In particular, the DeviceControl is a class from the com.nokia.mid.ui packet.

In the code in Listing 2 we still can detect some dependences and particular features that need to be removed to obtain a generic version of the *aspect*. Let us to see them:

- the name of the main class –`NewsReaderMIDlet`- in line 5. As the dependence is only localized in the *joinpoint* definition, we apply the *abstract pointcut* idiom [3].
- the reference to the particular class of the Nokia UI API (the class `DeviceControl` in line 13) makes this code runnable only on Nokia devices and emulators, ruining our purpose of genericity. As the device variability is in the *advice*, we apply the *template advice* idiom [2], encapsulating the special feature in specialized methods.
- the conditional check *pointcut* in line 8. To abstract the code from this condition we apply the *pointcut method* idiom [3], encapsulating the decisions in a boolean method.

4. COMPARATIVE STUDY

4.1. Analysis goals, method and approach

In order to evaluate the impact of *aspects* in context-aware applications for constrained devices, we have developed three different brightness-aware versions for the same non brightness-aware application, the J2ME news reader above presented. These versions are: (a) the pure aspectual version proposed in the previous section (*Aspectual* version henceforth); (b) a pure object-oriented version –without any design pattern- (*OOriented* version); (c) another object-oriented version using the Observer and the Singleton patterns (*Patterned* version). The three versions present identical behaviours.

As mentioned previously, the metrics we have used to compare and analyse the three approaches are: (1) the bytecode size (the size of the jar version to be distributed in mobile phones) and (2) the number of lines of source code, paying special attention in which ones entangle the business code or either conform new classes in order to evaluate the impact in the business code. We also compare other quality considerations.

4.2. Experimental results

As it was expected, in the *Patterned* and *OOriented* versions the brightness concern is scattered throughout the code, tangling core functionalities. As expected, that is more relevant in the *OOriented* version. Trying to measure this, we have collected the size of the source code destined to the brightness concern, taking into account both the new package and the code scattered along the rest of the application to reach the behaviour pursued. These results are presented in Table 1. To help in the comparison, we also show these measures for the original application.

	Original version	(a) Aspectual	(b) OOriented	(c) Patterned
Total source code	5429	5746	5806	5783
brightness package	—	317	264	279
scattered code lines	—	0	113	75
% of scattered code	—	0	35,3	21,1
Nº of affected classes	—	0	20	19

Table 1. Comparative in the extension of the source code and tangling effects among versions

The second and third rows refer to parts of the code added for the new non-core functionality (neat and scattered respectively). The fourth row is the percentage of scattered code in relation to the wholly new code. The fifth row presents the number of classes coupled with the `BrightMonitor` class. Regarding the aspectual version, we can see that any class in the *logical layer* has been affected with the addition of the new functionality. There is no dependence –no coupling- due to all the management of the brightness concern is encapsulated in the *context-aware layer* (the brightness package) and in the *aspectual layer* (the *annotator aspect* and the `Brightness` *aspect*).

According to the other metric analysed (the bytecode size), the results obtained are not favourable to the aspectual version. As we refer to the bytecode size, the weaving step is already applied by the compiler. We can see it is very inefficient, because for the shorter programs (the aspectual), the bytecode size is superior. That happens because the AspectJ compiler instruments the java code in order to provide some reflection capabilities, generating new and bigger classes. In particular, *Aspectual* version shows to be 51% bigger than *Patterned* and *OOriented* versions.

Apart from that, in the aspectual version we have to also link de AspectJrt (the AspectJ runtime jar), necessary to make the deliverable code runnable. That results in a considerable increase in the final bytecode size, as it is shown in table 2. To make the interpretation clearer, we present the size of the bytecode both compressed and decompressed, and also the size of the libraries required. The most interested rows are the first and the sixth ones. The latter presents the source code uncompressed, once the weaving is completed. All the values are in Kbytes.

	Original version	(a) Aspectual	(b) OOriented	(c) Patterned
compressed bytecode	124	258	132	132
uncompres. bytecode	219	481	234	234
resources	7,7	7,7	7,7	7,7
kxml lib	66,7	66,7	66,7	66,7
AspectJrt	—	204	—	—
uncomp. source code	144	205	160	159

Table 2. Comparative in the bytecode size among versions

As it is shown, only the AspectJrt takes up 204 Kbytes uncompressed. It has been announced that a version specific for J2ME has to be delivered in short.
Finally, we want to point that the comparative study has been made considering a unique context concern. It is important to notice that the lack of flexibility and the consequent problems in maintainability and so on can be extraordinarily incremented if we need to incorporate other types of contextual concerns in the same application, or if simply the contextual needs vary. That is this way, even for the *Patterned* version, because despite of using patterns, it becomes more difficult to isolate each contextual concern and to understand the code when the number of such concerns increases. However, in the *Aspectual* version, it is an easy operation consisting of weaving the selected *aspects*.

6. CONCLUSIONS

We present a metadata-based aspectual decomposition approach to provide seamlessly non context-aware systems with context-aware capacities. We prove that it causes no impact over the base application obtaining, at the same time, system-independent adaptation mechanisms, resulting in a non-invasive technique. Definitively, AOP is the better approach for our purpose. In particular, the patterned version analysed does not improves significantly regarding the object-oriented one. Furthermore, the reusability and extensibility problems make worse. Incidentally, the approach presented allows producing low resource-consuming components, suitable for compact devices and the pervasive design field, as proposed in [7].
From data obtained, we conclude that the burden on bytecode size while using *aspects* is not in our power. We will wait for a specific J2ME version. However, the aspectual version shrinks the source code. Independently, the aspectual version gains in other added values such as reusability, ortoghonality, pluggability, maintainability, essential for good designs.

ACKNOWLEDGMENTS

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

REFERENCES

[1] G. Chen and D. Kotz, *A survey of context-aware mobile computing research*, Technical Report TR2000-381, Computer Science Dep., Dartmouth College (2000).

[2] S. Hanenberg and A. Schmidmeier, "Idioms for Building Software Frameworks in AspectJ", *2nd Workshop on ACP4IS* (2003).

[3] S. Hanenberg, R. Unland and A. Schmidmeier, "AspectJ Idioms for Aspect-Oriented Software Construction", *Proc. of the EuroPlop'03*, Irsee, Germany, (June 2003).

[4] R. Laddad, *AOP and Metadata: a perfect match*. IBM DeveloperWorks, (2005).

[5] M. Sendín and J. Lorés, "Towards the Design of a Client-Side Framework for Plastic UIs using Aspects", *1rst Intern. Works. on Plastic Services for Mobile Devices* (2005).

[6] M. Sendín and J. Lorés, J., "Plasticity in Mobile Devices: a Dichotomic and Semantic View", *Workshop Engineering Adaptive Web*, supptd. by AH 2004, pp. 58-67, (2004).

[7] M. Sendín, "Implicit Plasticity Framework: a Client-Side Generic Framework for Context-Awareness", *Proc. of the I Intern. Conf. on Ubiquitous Computing* (2006).