Adaptation as an Aspect in Pervasive Computing

Awais Rashid, Gerd Kortuem

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{awais | kortuem} @comp.lancs.ac.uk

Abstract. Adaptation is one of the key characteristics of pervasive computing applications. However, implementing adaptation using conventional development techniques is challenging as adaptation requirements tend to affect multiple elements of a pervasive environment. In this paper, we present our experience with implementing adaptation using aspect-oriented programming (AOP). We argue that the use of AOP not only provides an improved modularisation of the adaptation concern but also makes it easier to adapt the pervasive application to both changes in data and reorganisation of the pervasive environment itself.

1. Introduction

Pervasive computing aims at realising the vision of the information society where computers are embedded within the environment and applications seamlessly interact and exchange information with each other and the users. Though most modern day software systems, especially those servicing volatile business domains such as banking and e-commerce, need to be adaptable to changing requirements, adaptation is an even more crucial characteristic of pervasive computing applications. New interaction mechanisms, devices or services may be added to a pervasive environment requiring them to be adapted to the specific characteristics of the environment. Similarly, the existing elements may be reorganised or adapted on the fly to react to changes in user behaviour and data/information imparted or manipulated by the pervasive environment. This last characteristic of pervasive computing environments is often referred to as *context-awareness* [4, 5] and is a major focus of research in pervasive computing.

Implementing adaptation in a pervasive environment is a challenging task as the adaptation concern affects multiple elements (devices, services, etc.) in the environment. The problem is further compounded by the fact that the elements are geographically distributed and in many instances there is no central node controlling the operation of the pervasive environment.

Aspect-oriented programming (AOP) [3, 7] has been proposed as a means to effectively modularise such crosscutting properties, i.e., properties that have a broadly scoped effect on a system. AOP techniques provide abstractions and constructs to modularise such properties, specify their composition relationships and compose them with other elements of a system. The composition relationships are specified using a *join point model* which identifies well-defined points within a system where an aspect may be composed. The composition process is often referred to as *weaving*.

In this paper we present our experience with using AOP to modularise adaptation in a pervasive environment supporting users to navigate their way to destinations and events across the Lancaster University campus. We have chosen to use AspectJ [1], an aspect language for Java to implement our application. Our choice is driven by the

maturity of the language, its compiler and availability of effective tool support[1]. Section 2 in this paper describes the pervasive navigation environment in more detail. Section 3 discusses the aspectisation of the adaptation concern using AspectJ. Section 4 reflects on our experience and concludes the paper.

2. Navigation Application

The pervasive environment we are developing involves a set of display devices (e.g., flat LCD panels, PDAs, etc.) to be deployed at strategic public locations across the Lancaster University campus. The environment is aimed at supporting a range of applications including, but not limited to, displaying news, disseminating information on upcoming events and assisting visitors (and also staff and students) in navigating their way around campus. We have chosen to focus on the navigation application for the purpose of the prototype aspect-oriented implementation discussed in this paper.

Visitors, staff and students often need to find their way to various destinations around campus. The destination can be a physical location such as a building, department or a lecture theatre or it can be an event such as a conference being hosted in a particular building. The destination is often dynamic as a particular event may have been moved to a different building or various sessions relating to the same event might be taking place in multiple buildings or the event may be held in different buildings on different days of the week. Similarly, though less dynamic than navigation information relating to events, a department may move to a different building or expand to take up additional space in another building. Similarly, alternative routes may need to be displayed in case a particular path is blocked due to building or renovation works or when the navigating person has special requirements such as wheelchair accessibility.

Furthermore, each new display added to the environment must adapt its specific properties to those of the environment. Displays may also be moved as the environment expands or new applications, usage scenarios and services are added.

The UML diagram of the environment is shown in Fig. 1.

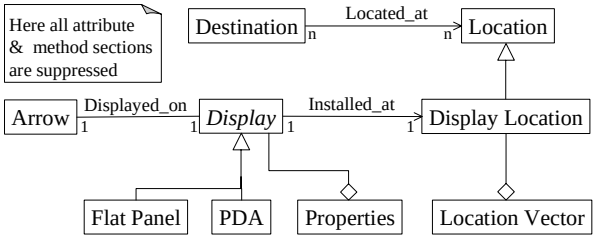

Fig. 1. UML diagram of the pervasive environment

The objects represented by the various classes in Fig. 1 are as follows:

- *Destination:* A destination on campus, e.g., building, department, event, etc.
- *Location:* A location based on coordinates on the campus map.

[1] Due to limited space, we do not provide an introduction to the AspectJ syntax. Readers not familiar with AspectJ should refer to [1] for details of the language.

- *Display Location:* The location on the campus map where a display has been installed.
- *Location Vector:* A vector pointing outwards from the display. Used to determine which way a display is facing and whether it has been moved.
- *Display:* An abstract class representing a display in the environment.
- *Flat Panel:* A specific type of display, the flat LCD panel.
- *PDA:* A specific type of display, personal digital assistant.
- *Properties:* The specific characteristics of an individual display.
- *Arrow:* The data to be displayed to assist with navigation (in this case a simple arrow pointing in the direction to be followed).

3. Aspectising Adaptation

When modularising adaptation we need to address three specific facets of adaptation within our pervasive environment. The first two are application independent and relate to any application deployed in the environment while the third is specific to the navigation application:

1. *Display management:* As the environment expands more displays will be incorporated into it. All new displays must have their specific properties adapted for use within the pervasive environment. Furthermore, although the UML diagram in Fig. 1 only shows two specific types of displays, Flat Panel and PDA, it is conceivable that other types of display devices may be added to the environment as they become available.
2. *Content management:* The navigation content (an arrow in this case) is only one type of content to be displayed on the devices. There are other types of content that also need to be delivered to the devices. Furthermore, as new displays are added, the content already being displayed within the environment has to be made available on them as well.
3. *Display adaptation:* As a new destination is added or an existing destination changed (e.g., change of venue for an event), the displays need to be adapted to guide the users to the correct destination. Furthermore, if a display is moved to a different location it should be adapted to display the content in a *correct* fashion based on its new location.

We have modularised each of these facets of the adaptation concern using AspectJ aspects.

3.1 Display Manager Aspect

The `DisplayManager` aspect (cf. Fig. 2) encapsulates all functionality relating to incorporation of new displays or adaptation of their properties to the pervasive environment. The aspect maintains a collection of all `displays` incorporated into the environment and has a public method to traverse the collection (cf. label (A) in Fig. 2). This is useful for other elements of the system, especially the `ContentManager` aspect, which needs to access all the displays in the system from time to time as new content becomes available.

The three inter-type declarations (cf. label (B) in Fig. 2) introduce display incorporation functionality into the abstract `Display` class. Two final static variables representing the two available display types are introduced. As new display types become available, they can be introduced in a similar fashion. The introduced static

`incorporateDisplay` method instantiates the right type of class as a new display is incorporated. If a suitable display type does not exist, a `DisplayTypeNotFoundException` is thrown.
The `displayIncorporation` pointcut (cf. label (C) in Fig. 2) captures all calls to the static method introduced into the `Display` class. An `after` advice then adds the incorporated display to the `displays` collection in the aspect as well as adapts the properties of the newly incorporated display to the pervasive environment.
Note that although the `DisplayManager` aspect affects only a single class, nevertheless it encapsulates a coherent concern. This use of an aspect is, therefore, very much in line with good separation of concerns practice.

```
public aspect DisplayManager {

    private Vector displays = new Vector();

A   public Enumeration displays() {
        // code
    }

    public static final int Display.PDA = 1;
    public static final int Display.FLAT_PANEL = 2;

    public static Display Display.incorporateDisplay(int id,
B                                                    DisplayLocation location,
                                                     int displayType)
                                    throws DisplayTypeNotFoundException {
        // code
    }

    pointcut displayIncorporation():
        call(public static Display Display.incorporateDisplay(..));

C   after() returning(Display display): displayIncorporation() {

        // code
    }

}
```

Fig. 2. The Display Manager aspect

3.2 Content Manger Aspect

The `ContentManager` aspect is shown in Fig. 3. It declares that all types of content must implement the `Content` interface (cf. label (A) in Fig. 3). Note that in this case there is only one type of content, `Arrow`, shown but in practice the pervasive environment displays a variety of content. The `Content` interface provides an application independent point of reference for the two pointcuts within the aspect, hence decoupling content management from the type of content being managed. The `contentAddition` pointcut (cf. label (B) in Fig. 3) traps calls to `addContent` methods in all application classes. An `after` advice for the pointcut then traverses all the displays registered with the `DisplayManager` and updates them with the new content. The `pushContentOnNewDisplay` pointcut (cf. label (C) in Fig. 3) captures

the instantiation of all sub-classes of the `Display` class. An `after` advice then pushes the available content onto the newly instantiated display.

```
public aspect ContentManager {

A   declare parents: Arrow implements Content;

    pointcut contentAddition(Content c):
             call(public * *.addContent(Content)) && args(c);

B   after(Content c): contentAddition(c) {

        // code
    }

    pointcut pushContentOnNewDisplay(): call(Display+.new(..));

    after() returning(Display display): adaptNewDisplayProperties() {
C
        // code
    }

}
```

Fig. 3. The Content Manager aspect

3.3 Display Adaptation Aspect

While the `DisplayManager` and `ContentManager` aspects are application independent and handle adaptation facets that span across applications in the pervasive environment, the `DisplayAdaptation` aspect, shown in Fig. 4, is specific to the navigation application. The `destinationChanged` pointcut in this aspect (cf. label (A) in Fig. 4) captures the change in location of an existing destination or the creation of a new destination. An `after` advice for the pointcut invokes the adaptation rules for the displays to adapt the content accordingly.

```
public aspect DisplayAdaptation {

    pointcut destinationChanged():
        execution(public void Destination.setLocation(..))
        || execution(public Destination.new(..));

A   after(): destinationChanged() {

        // code
    }

    pointcut displayMoved():
        execution(public void DisplayLocation.setLocationVector(..));

B   after(): displayMoved() {

        // code
    }

}
```

Fig. 4. The Display Adaptation aspect

The `displayMoved` pointcut (cf. label (B) in Fig. 4) identifies that a display has been moved by capturing the change in its location vector[2]. An associated `after` advice then proceeds to adapt the content of the moved display and any neighbouring displays accordingly.

4. Discussion and Conclusion

The three aspects in section 3 clearly demonstrate that AOP constructs provide an effective means to modularise both application independent and application specific facets of adaptation in a pervasive environment. The use of aspects makes it easier to not only adapt the environment to changes in content but also makes it possible to react to the reorganisation of the displays in an effective fashion. Furthermore, any changes to the adaptation characteristics of the environment or the navigation application are localised within the aspects hence avoiding changes to multiple elements of the system that would have otherwise been required.

There are also interesting observations to be made about design of aspect-oriented systems. Firstly, the use of `Content` as an application independent point of reference makes it possible to decouple the `ContentManager` from application specific content. This is similar to the use of a Persistent Root Class in [8] to decouple the persistence concern from application-specific data. Also, similar to [8], we can observe that the notion of one large AspectJ aspect (or one in any other AOP technique) modularising a crosscutting concern does not make sense. The three aspects and the `Content` interface together modularise adaptation. This further strengthens the argument initially presented in [8] that developers should utilise aspect-oriented and object-oriented abstractions together in a coherent *framework* to modularise a crosscutting concern. While different classes and AspectJ aspects modularise specific facets of a crosscutting concern, it is the *framework* binding them together that, in fact, aspectises the concern in question.

Though our experience and observations in this paper are based on AspectJ, we are of the view that they will apply to other aspect-oriented approaches as well. As such we aim to employ other AOP frameworks such as JBoss [6] and AspectWerkz [2] for implementation of the same application in the future for comparative evaluation.

References

[1] "AspectJ Project", http://www.eclipse.org/aspectj/, 2004.

[2] "AspectWerkz Home Page", http://aspectwerkz.codehaus.org, 2004.

[3] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", *Communications of ACM*, Vol. 44, No. 10, 2001.

[4] H. W. Gellersen, A. Schmidt, and M. Beigl, "Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts", *Mobile Networks and Applications*, Vol. 7, No. 5, 2002.

[5] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, "The Anatomy of a Context-Aware Application", ACM/IEEE MobiCom Conf., 1999.

[6] "JBoss Aspect Oriented Programming Webpage", http://www.jboss.org/products/aop.

[7] G. Kiczales et al., "Aspect-Oriented Programming", ECOOP, 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241, pp. 220-242.

[8] A. Rashid and R. Chitchyan, "Persistence as an Aspect", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 120-129.

[2] Change in location vector is the most appropriate way to identify that a display has been moved as it might not have been physically moved but simply rotated at its current position on the map.