Aspect-Oriented Programming for Pervasive Computing: A Comparative Study

Awais Rashid
Computing Department
Infolab21, Lancaster University
Lancaster LA1 4WA, UK
+44-1524-510316
awais@comp.lancs.ac.uk

Gerd Kortuem
Computing Department
Infolab21, Lancaster University
Lancaster LA1 4WA, UK
+44-1524-510324
kortuem@comp.lancs.ac.uk

James Walkerdine
Computing Department
Infolab21, Lancaster University
Lancaster LA1 4WA, UK
+44-1524-510352
walkerdi@comp.lancs.ac.uk

Computing Department, Lancaster University, Technical Report COMP-002-2005

ABSTRACT
Recent years have seen an increasing interest in research into and development of pervasive and ubiquitous computing environments. Such environments require a high degree of adaptability and are often inherently distributed in nature. Adaptation and distribution are properties with a crosscutting nature as the need for adaptation is determined by the context in which various elements of the environment operate. Similarly, the environment often tends to be composed of elements in different locations, which need to communicate with each other and exchange information to meet the needs of the pervasive computing applications supported by the environment. Aspect-oriented programming (AOP) has emerged as a promising candidate to support modularisation of crosscutting concerns, such as adaptation and distribution, in a reusable, evolvable and maintainable manner. In this paper, we discuss our experience of implementing an adaptive peer-to-peer (P2P) display environment using AOP. We compare the AOP implementation with two independently developed OO implementations of the environment, one using a regular client-server model and the other using a P2P application framework. The comparison demonstrates that an aspect-oriented approach is indeed more effective in modularising adaptation and distribution in a reusable, maintainable and evolvable fashion. It also reduces the complexity of the implementation with respect to the above three desirable attributes. At the same time, our experience challenges some of the existing (mis)conceptions about aspect granularity within an application and also highlights the need for development guidelines and idioms.

Keywords
Aspect-oriented applications, evolution, adaptability, maintainability, pervasive computing, reuse, aspect-oriented frameworks

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.

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 often geographically distributed and in many instances there is no central node controlling the operation of the pervasive environment. Therefore, the distribution concern has to be catered for across the various elements forming the environment.

Aspect-oriented programming (AOP) [11, 19] has been proposed as a means to effectively modularise such crosscutting properties, i.e., properties that have a broadly scoped effect on a system. Not only does AOP support improved separation of crosscutting concerns, it promises to provide such separation in a manner that promotes reusability, maintainability and evolvability. However, few application studies exist so far to demonstrate the effectiveness of AOP-based implementations with respect to these quality attributes.

One of the earliest application studies of AOP was ATLAS [18], a Web-based, distributed, learning environment, which investigated the effectiveness of AOP (using an earlier version of AspectJ) from the perspective of maintenance and change and also made some important observations about aspect-class associations. Soares et al. [32] have reported their experience on using AOP for refactoring distribution and persistence in a layered Web-based information system and derived guidelines for incremental development of aspects in a system. A detailed study of modularising the persistence concern in a large-scale bibliography application has been carried out at Lancaster previously [27] and observed that the notion of obliviousness – the base concerns remaining fully unaware of the aspects being applied to them – does not always make sense or is effective. In fact, at times key architectural decisions rely on concerns not remaining oblivious of each other. Similar observations were made by Kienzle et al. [20] who studied the application of AOP to modularise transaction management code. Some studies from IBM [7, 8] have investigated the use of AOP, specifically AspectJ, to reduce complexity in large-scale middleware platforms. Similarly, the DAOP platform [24] has been used to implement coordination in collaborative virtual environments. However, none of the application studies so far have focused on applying AOP in pervasive computing environments or more specifically on modularising two key crosscutting properties in such environments namely, adaptation and distribution.

It is important to investigate the effectiveness of AOP to improve reusability, maintainability and evolvability in a pervasive environment as such environments are aimed at underpinning the next generation of applications. From an AOP perspective, such an investigation would inform the design of AOP languages, frameworks and methodologies to better serve such emerging adaptive, distributed environments. From a pervasive computing viewpoint, such a study would provide insight into a new modularisation technique that promises to provide an effective means to develop, maintain, reuse and evolve crosscutting concerns, such as adaptation and distribution, which are at the heart of pervasive applications.

In this paper we present our experience with using AOP to modularise adaptation and distribution 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 [3], 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. Section 2 in this paper describes the pervasive navigation environment in more detail. Section 3 discusses the aspect-oriented implementation of the environment in question. We carry out the implementation in an incremental fashion: first building a standalone application with adaptation concerns modularised with AOP, then adding P2P distribution capabilities. This provides interesting insights into whether such closely related concerns can be implemented incrementally using AOP. Section 4 discusses two alternative OO implementations of the environment, one using a regular client-server distribution model and the other using a P2P application framework. We must emphasise that all three implementations, i.e., the AO implementation as well as the two OO implementations, have been carried out completely independently of each other. This has ensured that no biases for or against a particular development technique have crept into the comparison presented in section 5. We have chosen to compare the three implementations on the following criteria for both adaptation and distribution concerns:

- Modularity
- Reusability
- Maintainability
- Evolvability

In addition we also compare the complexity of the three implementations with respect to the above qualities. Section 6 discusses some related work while section 7 concludes the paper and identifies directions for future work.

2. THE PERVASIVE DISPLAY ENVIRONMENT

The pervasive environment we are developing involves a set of display devices (e.g., flat LCD panels, PDAs, etc.) to be deployed 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 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.

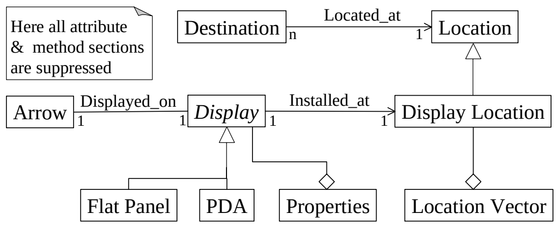


Fig. 1: UML diagram of the pervasive environment

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.
- *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).

Note that each display is a self-contained computer with its own processing, storage and wireless networking capabilities. It is also equipped with a GPS receiver and an electronic compass to determine its position (i.e., location and orientation) hence, facilitating position-aware adaptation (filtering and transformation) of content.

3. ASPECT-ORIENTED IMPLEMENTATION

In addition to exploring whether we can modularise adaptation and distribution in a reusable, maintainable and evolvable manner using AOP, we aimed to explore two other key issues pertaining to AOP when developing our pervasive environment. Firstly, we wanted to explore whether it is indeed possible to aspectise, and if so to what extent, two closely related crosscutting concerns, adaptation and distribution, in an incremental fashion – a number of approaches, e.g., [15, 32], have advocated such an incremental approach to aspect-oriented development. Secondly, we aimed to investigate whether it is really possible to develop a non-distributed implementation of the environment and later introduce the distribution aspect without significant changes – this is often cited as a promising application of AOP [7, 8, 16, 23, 24, 32].

In order to obtain some answers to the above questions, we first developed a non-distributed version of the environment. This version solely focused on modularising adaptation code using AOP and no distribution concerns were taken into account. Once we had a working implementation of the non-distributed version, we set about adding distribution capabilities to the environment, again using AOP, based on a fully decentralised P2P architecture.

3.1 Non-distributed Implementation

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 stylised before they are 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.

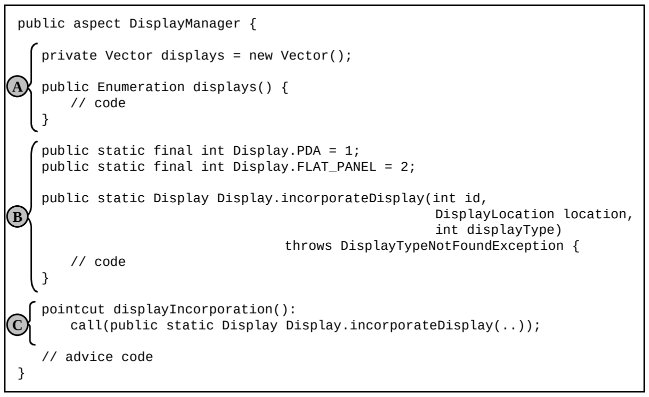
We have modularised each of these facets of the adaptation concern using AspectJ aspects.

3.1.1 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.



```
public aspect DisplayManager {

   private Vector displays = new Vector();

   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,
                                                    DisplayLocation location,
                                                    int displayType)
                                 throws DisplayTypeNotFoundException {
      // code
   }

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

   // advice code
}
```

Fig. 2: The Display Manager aspect

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. Had we not used this aspect, display management concerns would have been coupled with the core functionality of the `Display` class.

3.1.2 Content Manager 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 pointcuts within the aspect, hence decoupling content management from the type of content being managed. Any classes that manipulate content in the pervasive applications deployed in the environment are required to implement the `ContentManipulator` interface, which specifies a number of methods for content addition, removal and update. Note that this interface is not implemented via the aspect, i.e., using the declare parents feature of AspectJ, as each application has its own content manipulation requirements. Like the `Content` interface, the `ContentManipulator` interface also provides an application-independent point of reference to capture all content manipulation behaviour within the applications in the environment, including the navigation application. The `contentAddition` pointcut (cf. label (B) in Fig. 3) traps calls to `addContent` methods in all application classes manipulating content. An `after` advice for the pointcut then traverses all the displays registered with the `DisplayManager` and updates them with the new content. The `contentDeletion` and `contentUpdate` pointcuts (cf. label (C) in Fig. 3) and their associated advice perform similar functions upon content deletion and update. The `pushContentOnNewDisplay` pointcut (cf. label (D) 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):
B          call(public * ContentManipulator.addContent(Content))
           && args(c);

   pointcut contentDeletion(Content c):
           call(public * ContentManipulator.deleteContent(Content))
           && args(c);
C  pointcut contentUpdate(Content c):
           call(public * ContentManipulator.updateContent(Content))
           && args(c);

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

   //advice code
}
```

Fig. 3: The Content Manager aspect

3.1.3 Display Adaptation Aspect

While the `DisplayManager` and `ContentManager` aspects are application independent and handle adaptation facets that span across applications in our 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():
A      execution(public void Destination.setLocation(..))
       || execution(public Destination.new(..));

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

   //advice code
}
```

Fig. 4: The Display Adaptation aspect

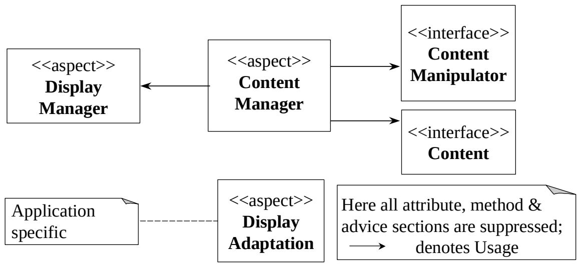
The `displayMoved` pointcut (cf. label (B) in Fig. 4) identifies that a display has been moved by capturing the change in its location vector[1]. An associated `after` advice then proceeds to adapt the content of the moved display and any neighbouring displays accordingly.

3.1.4 Discussion

The three aspects in section 3.1.1-3 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 the design of the adaptation concern. Firstly, the use of `Content` and `ContentManipulator` as application independent points of reference makes it possible to decouple the `ContentManager` from application-specific content and content manipulation operations. This is similar to the use of a Persistent Root Class in [27] to decouple the persistence concern from application-specific data. Also, similar to [27], 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 in the case of the adaptation aspect either. The three aspects and the `Content` and `ContentManipulator` interfaces together modularise adaptation (cf. Fig. 5). While different classes and AspectJ aspects modularise specific facets of the adaptation concern, it is the *framework* binding them together that, in fact, aspectises this particular crosscutting concern.

[1] 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.



<<aspect>> Display Manager
<<aspect>> Content Manager
<<interface>> Content Manipulator
<<interface>> Content
Application specific
<<aspect>> Display Adaptation
Here all attribute, method & advice sections are suppressed;
denotes Usage

Fig. 5: Framework modularising *Adaptation*

3.2 Adding P2P Distribution

There are some very interesting observations that come to light once we start on the next increment, i.e., the introduction of the distribution aspect. Some features that "naturally" reside in the three aspects in the non-distributed version in section 3.1 do not necessarily belong there once we start considering distribution. They either are not needed any more or would have been placed in a separate aspect had we considered distribution when designing our adaptation concern.

The first example of this can be seen in the form of the `DisplayManager` aspect itself. The code in labels (A) and (B) in Fig. 2 is used to incorporate new displays into the system and keep track of all existing displays in a centralised fashion. However, in our increment we have chosen to use a purely decentralised P2P distribution mechanism. Consequently, each display forms a peer node within the P2P network. There is no central control server and as a result a centralised tracking of displays in the environment does not make sense – as a new display is added it advertises its existence to displays (peers) close to it (note that it can also advertise to all peers in the environment and not just the nearby ones) resulting in them becoming aware of its existence. This means that a centralised collection of all displays in the environment does not make sense any more. As a result, code in labels (A) and (B) in Fig. 2 is not needed any more. This also brings to front a problem with the `displayIncorporation` pointcut (cf. label (C) in Fig. 2). The pointcut is currently defined on the basis of the `incorporateDisplay` method which is introduced into the `Display` class by the `DisplayManager`. Once the introduced method is removed, the pointcut is no longer valid. This highlights the problems that can arise if pointcuts and advice are specified with reference to methods or attributes introduced using inter-type declarations. Had the pointcut been defined directly on the basis of instantiation of sub-classes of `Display` as:

```
pointcut displayIncorporation( ):
                                    call(Display+.new(..))
```

it would have not become invalid once the inter-type declaration was removed. Furthermore, had the `displayIncorporation` pointcut been specified in a separate aspect, we could simply remove the (remainder of the) `DisplayManager` aspect as a whole as it is not needed in our P2P distribution implementation. Note, however, that had we chosen a regular client-server model then the (remainder of the) `DisplayManager` aspect could have been reused as it is; it would have formed an effective element of a central server managing displays in such a case.

A similar example can be observed in the `ContentManager` aspect. In a non-distributed implementation, pushing content on new displays, i.e., the `pushContentOnNewDisplay` pointcut, seems to naturally reside in the `ContentManager` aspect. However, when we start to introduce distribution, it is clear that this is something that must be handled by the P2P communication features of the environment. Even in a regular client-server model this would be the task of the distribution concern and not the content management concern. The pointcut simply serves as a temporary mechanism to simulate communication in the non-distributed implementation. It would be best placed in a separate aspect so that it can simply be excluded when the distribution concern is implemented.

The above examples highlight that incremental development of an aspect-oriented system may lead to us removing or keeping elements of the concern implemented in the previous increment depending on the design choices we make about the aspect being implemented in the current increment. This also implies that an increment should not be oblivious to the fact that there are further increments to follow it as well as the nature of the aspects to be implemented in those increments. If we had carefully considered the next increment, i.e., the distribution aspect when designing adaptation, we would have been aware of the design considerations highlighted above and catered for these when modularising adaptation.

Having made the above changes to the aspects modularising our adaptation concern, we can move on to introducing P2P distribution capabilities into the environment using AOP. We have chosen to build the P2P capabilities using Sun Microsystems' JXTA [17] which provides a set of open protocols for realisation of a decentralised P2P network. The choice of a decentralised P2P architecture is driven by the fact that we wish our display nodes to be independent and autonomous. Content can be added to an application running on any peer and it should not only get propagated to other peers but the peers (or at least those close to each other) can communicate to figure out the best way to display the content. An example of this is a scenario where a user walks up to a small, PDA-sized, display and requests information that is too rich to be displayed on the device. In such a scenario, the small display can enquire whether any of the peers close to it is a large display capable of presenting the information to the user, request it to do so and, provided the request is accepted by the large display, direct the user to it using the navigation application. Note that this might require further communication and interaction with nearby peers if the larger display is not immediately close to the small display. In this case, the intermediate displays will have to display the correct direction of the arrow for the user to reach the large display.

3.2.1 P2P Communication Aspect

Before discussing the `P2PCommunication` aspect, it is important to highlight two helper classes, `JXTA_Setup` and `PipeListenerThread`, that play a key role in the modularisation of the P2P distribution code. The `JXTA_Setup` class encapsulates functionality to initialise JXTA and create a peer group. It also encapsulates features for peer discovery as a new peer joins the peer group. The `PipeListenerThread` is a simple listener that waits for an incoming message on the input pipe for the peer (JXTA uses the notion of input and output pipes for connection among peers).

The `P2PCommunication` aspect (cf. Fig. 6) traps the instantiation of a display, i.e., addition of a new display into the environment (cf. label (A) in Fig. 6). An `after` advice operating on the `peerCreation` pointcut then instantiates the `JXTA_Setup` class, initialises JXTA and carries out peer discovery. This results in the new peer being added to the peer group and publishing its advertisement (a means to inform other members of the peer group about its existence). The `inputPipeCreation` pointcut (cf. label (B) in Fig. 6) works in tandem with this and traps the publishing of such an advertisement during JXTA initialisation. An `after` advice then instantiates the `PipeListenerThread` to associate an input pipe listener with the peer.

The `messageArrived` pointcut (cf. label (C) in Fig. 6) simply waits until a new message is received by the `PipeListenerThread` instance. Once a new message arrives, it passes it onto the relevant application for processing.

The `P2PCommunication` aspect also has three pointcuts identical to those in the `ContentManager` aspect. There relate to content addition, deletion and update (cf. label (D) in Fig. 6; note the use of application independent reference points: `ContentManipulator` and `Content`). Since all displays in the environment are autonomous, content can be added at any display and must be conveyed to other peers in the display network. The `after` advices associated with each of the three pointcuts capture this communication facet; note that the `ContentManager` aspect handles the stylistic issues pertaining to the content and not communication (the only pointcut simulating communication, i.e., the `pushContentOnNewDisplay` pointcut in Fig. 3 has been removed in this iteration).

```
 public aspect P2PCommunication {

A   pointcut peerCreation(): execution(Display+.new(..));

    pointcut inputPipeCreation(): cflow(
B              execution(public void JXTA_Setup.initialise())) &&
               call(Advertisement AdvertisementFactory.
                                              newAdvertisement(..));

    pointcut messageArrived(Message message):
C              call(void PipeListenerThread.setMessage(Message)) &&
               args(message);

    pointcut contentAddition(Content c):
            call(public * ContentManipulator.addContent(Content))
            && args(c);
    pointcut contentDeletion(Content c):
D           call(public * ContentManipulator.deleteContent(Content))
            && args(c);
    pointcut contentUpdate(Content c):
            call(public * ContentManipulator.updateContent(Content))
            && args(c);
    //advice code and helper methods
 }
```

Fig. 6: The P2P Communication aspect

3.2.2 Aspect Precedence

The `P2PCommunication` and `ContentManager` aspects operate with reference to the same set of pointcuts for content addition, deletion and update. Similarly, the `P2PCommunication` and the `DisplayManager` aspects (with now only the new `displayIncorporation` pointcut and associated advice) both operate with reference to the instantiation of new displays. Therefore, we need to define clear precedence rules between these two aspects.

If we look at our display environment, we can observe that any stylistic manipulation of the content by the `ContentManager` must be carried out before it is passed onto other peers by the `P2PCommunication` aspect. Similarly, it is important to establish connections among peers (via the `P2PCommunication` aspect) before discovering the properties of the environment and adapting a new display to these properties (in the `DisplayManager` aspect). Since these precedence rules are simple and static (i.e., the precedence doesn't change depending on the dynamic context as in [27]), we can specify them easily with the `declare precedence` declaration in AspectJ. We have chosen to define these precedences in a separate aspect as we consider interaction rules to be crosscutting the aspects whose interactions they govern. Consequently, the `AspectPrecedence` aspect (cf. Fig. 7) is an aspect of `ContentManager`, `DisplayManager` and `P2PCommunication` aspects.

```
    public aspect AspectPrecedence {

           declare precedence: ContentManager, P2PCommunication;
           declare precedence: P2PCommunication, DisplayManager;
    }
```

Fig. 7: The Aspect Precedence aspect

3.2.3 Discussion

Similar to the modularisation of adaptation, we can observe that the notion of a single, large aspect modularising distribution is not true. The `P2PCommunication` aspect together with the `JXTA_Setup` and `PipeListenerThread` classes and the `Content` and `ContentManipulator` interfaces provides a framework (cf. Fig. 8) which allows us to modularise distribution effectively. The framework is application independent and can seamlessly apply to any new application entering our pervasive display environment. At the same time, it could be reused in other similar content manipulating P2P environments. A modularised distribution approach also makes it possible to change the distribution approach without requiring any changes to the applications operating within the environment. This application independence is facilitated by the two interfaces which provide application-independent reference points for the aspect to operate on. It is also interesting to note that these interfaces are shared by both aspect frameworks, i.e., adaptation and distribution.

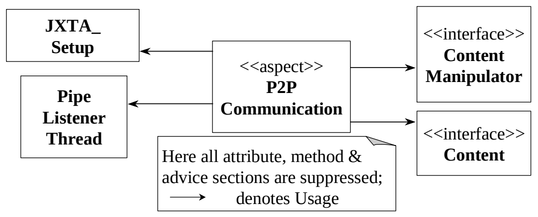

JXTA_ Setup
Pipe Listener Thread
<<aspect>> P2P Communication
<<interface>> Content Manipulator
<<interface>> Content
Here all attribute, method & advice sections are suppressed; ⟶ denotes Usage

Fig. 8: Framework modularising *Distribution*

Our experience has also highlighted the fact that incremental introduction of aspects during development is not as straight forward as it might seem at the first glance. One needs to be aware of aspects to be added in future increments otherwise it is likely that design decisions will need to be revisited when new aspects are added. This can be addressed by communication among different members of the development team working on different increments. Alternatively, an architect can keep abreast of these *"development aspects"* (note that this is a lifecycle concern that cuts across the development increments) and ensure that a holistic picture of the design is maintained at each increment.

4. PURE OO IMPLEMENTATIONS

We now discuss two independently developed OO implementations of the same pervasive environment. The first, GAUDI, uses a regular client-server distribution model and XML-based content transformation and adaptation. The second employs a decentralised P2P distribution model (similar to the AO implementation) but, instead of using AOP, employs a P2P application framework offering high-level services for P2P application development.

4.1 The GAUDI System

GAUDI, Grid of Autonomous Displays, is an OO realisation of the pervasive display environment. It consists of a central content server and an arbitrary number of autonomous displays units. The content server is responsible for storing content and pushing updates out to the displays. Content is generated by a number of applications running on the same physical host as the server. Since the displays are equipped with a GPS receiver and an electronic compass, the content server is ignorant of the position and capabilities of each display; each display receives the same generic (i.e., position-unspecific) content and decides on its own how best to adapt the content.

As shown in Fig. 9, the `DisplayManager` class in the content server keeps track of all connected displays by managing a collection of IP addresses and providing methods for traversing it. Displays contact the content server to explicitly connect and disconnect from it. The content server manages content for several applications, each of which is an independent process running on the same physical host as the server. The content is represented as an XML file or, more specifically, as an instance of the `GenericContent` class. Generic content consists of a collection of multimedia objects and adaptation rules. In the navigation application, the `ContentCreator` computes its content from two pieces of information: a campus map and the destination to which users should be guided. The output is an image object depicting an arrow and rules of how to rotate this object depending on a display's position. Note, however, that the adaptation is performed by the display and not by the content server or the application.

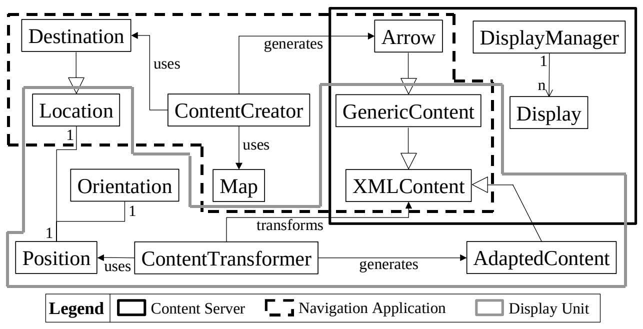


Fig. 9: The GAUDI system and its various elements

The main class in the display unit is the `ContentTransformer`. It takes as input positional information delivered from the sensor subsystem and uses it to adapt a generic content file in a position-aware manner. This is done by evaluating the adaptation rules contained in the `GenericContent` file. The result is an instance of the `AdaptedContent` class.

The GAUDI implementation does not focus on modularising the distribution behaviour. However, it handles the three specific facets of adaptation introduced in Section 3.1 as follows:

- *Display management:* Flexible display management is achieved by strictly limiting the knowledge the server (and application) needs to have about individual displays. Thus new displays can be incorporated by simply registering with the server, regardless of their specific characteristics.
- *Content management:* Flexible content management is achieved by introducing a common content representation format for all applications.
- *Content adaptation:* By strictly separating the roles and responsibilities between the application (content creation), server (content management) and display (content adaptation) it is possible to dynamically adapt the content in a position-aware manner. New content can be accommodated by pushing it out to all connected displays.

4.2 Using a P2P Application Framework

A second decentralised P2P implementation of the display environment was built using Lancaster's P2P Application Framework [34] (cf. Fig. 10). The framework is effectively an abstract layer geared specifically towards P2P application development. It reduces the burden on developers to understand the underlying P2P technology by providing a set of generic, protocol independent, application-oriented services. Such services include peer communication, discovery/searching, awareness, file sharing and network monitoring. By using the P2P Application Framework, users can rapidly build an application once, and use it over many different P2P protocols and network topologies.

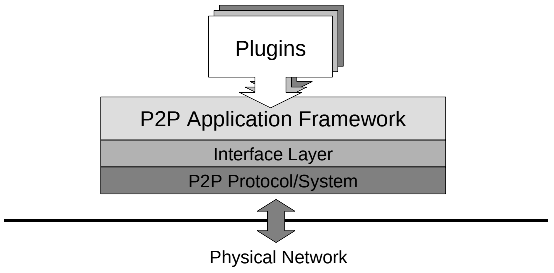


Fig. 10: Using the P2P application framework

Applications using the framework are developed in the form of Java-based plug-ins to the framework (cf. Fig. 10). Therefore, in case of our pervasive navigation application, the `NavigationPlugin` acts as not just the content manipulator (it implements the `ContentManipulator` interface) but also captures all the functionality for distributing information to other peer displays and adaptation behaviour to respond to changes in destination or moving of the display. The actual functionality for discovering and communicating with other peers is provided by the framework and accessed via its API. This separation means that the plug-in is not tied to a specific underlying P2P technology though the adaptation, distribution and content manipulation behaviour is closely coupled within the monolithic plug-in. Furthermore, any other applications on the same node in the display environment have to provide their individual implementation of the high-level distribution functionality as well as response to changes in case of content addition, deletion and update in a fashion similar to the `NavigationPlugin`.

5. COMPARING THE AO AND OO IMPLEMENTATIONS

We now compare the three implementations of our pervasive display environment with regards to modularity, reusability, maintainability and evolvability as well as the complexity of each implementation in realising the above properties.

Modularity: We can observe that the two aspect-based frameworks for adaptation and distribution in the pervasive environment help us modularise these concerns effectively. The frameworks use application-independent points of reference to decouple themselves from the details of individual applications within the environment. At the same time, the aspectisation of adaptation caters for application-specific facets of this particular concern. The use of aspects makes it easier for us to keep the application-specific element of adaptation separate from the application-independent elements. This is a direct consequence for choosing the right level of granularity for the aspects in our design and avoiding the temptation to modularise a concern using one large aspect module. The AO implementation initially has some development overhead due to the changes introduced to the past increment (in this case adaptation) when a new increment (distribution) is introduced. However, the guidelines we have inferred from this experience can help minimise such revisions during incremental development.

The XML-based content management and transformation approach in GAUDI makes it possible to modularise the various facets of adaptation. However, distribution is not effectively modularised and any changes to the distribution behaviour can have systemic, environment-wide impact. The `NavigationPlugin` implementation (based on the P2P Application Framework) is largely monolithic in that core application concerns are intertwined with adaptation and high-level distribution behaviour (e.g., application/domain dependent algorithms for distributing content evenly across peers). However, the framework does provide effective modularisation of low-level P2P protocols and services.

Reusability: The AO implementation of adaptation lends itself to a high degree of domain-specific reuse, e.g., pervasive environments of a similar sort manipulating and sharing information. The P2P distribution aspect framework, on the other hand, is much more generic and can be reused in any content manipulating application. Furthermore, new applications and content can be seamlessly deployed within the environment as long as they implement the `ContentManipulator` and `Content` interfaces respectively.

In a similar fashion, adaptation behaviour in GAUDI is also highly reusable in a domain-specific manner. The content generation and transformation approach is generic. However, the transformer might need to be extended to deal with other types of content from new applications. This is in contrast with the AO implementation where the adaptation aspect framework does not need to be modified as new applications are deployed. GAUDI does not modularise distribution so this concern cannot be reused. The `NavigationPlugin` implementation has a low degree of reuse with reference to adaptation and distribution but the underlying P2P framework provides a large-scale reuse mechanism facilitating development of other P2P applications, in markedly different environments and using different protocols.

Maintainability: The revisiting of the previous increment in our AO approach provides us with some insights into its maintainability. The changes to adaptation behaviour are limited to the two application-independent aspects, `ContentManager` and `DisplayManager`. The application-specific adaptation behaviour is isolated from these and hence remains unchanged. Since the P2P distribution code is completely separated from other elements of the environment, any changes or updates to it are localised to the distribution aspect framework.

In GAUDI, though the adaptation code is modularised through the XML-based content management and transformation approach, any changes to it are likely to carry a significant overhead as there is a significant code bloat arising from the inclusion of the XML processing code. The distribution behaviour is not modularised so changes to it will affect multiple elements of the pervasive environment, e.g., the server, the navigation application and the display unit. In case of the `NavigationPlugin`, changes to either adaptation and distribution code are expensive as they are not effectively localised. However, changes, such as, moving to a different P2P protocol or service are very inexpensive as the framework provides facilities for a seamless exchange.

Evolvability: Similar to maintainability, evolvability is facilitated by the AO implementation by keeping the adaptation and distribution behaviour modularised in the two aspect-based frameworks. Any updates or changes to application-independent or application-specific adaptation behaviour are localised to that particular aspect framework. Similarly, one can evolve the P2P distribution behaviour or move to a different mechanism without affecting the code implementing the rest of the elements of the pervasive environment.

The adaptation behaviour in GAUDI is also quite evolvable albeit it is complex to do so due to the significant amount of XML processing code. The distribution behaviour is much harder to evolve as it is tangled with the various elements of the environment. In the P2P Application Framework, one can evolve the framework easily by adding support for more low-level protocols and services. However, application evolution is a more intensive and difficult task as crosscutting properties are not modularised effectively.

Complexity: All three implementations provide modularity mechanisms which are easy to understand and use. In case of the AO implementation, the two aspect frameworks are fairly straight-forward to use. Same is the case for the adaptation mechanism in GAUDI and the high-level services provided by P2P Application Framework. In terms of reuse, maintainability and evolvability, the two aspect frameworks in the AO implementation provide a simple yet effective set of abstractions that one can employ, change or evolve. In case of GAUDI, the XML processing code poses significant complexity when one is trying to adapt it for reuse or maintenance or evolving it in line with changes to requirements for the pervasive environment. The P2P Application Framework is very simple to reuse due to its high-level application interface. The layered architecture offers support for maintaining and evolving individual layers. However, there is significant coupling among the layers and changes to elements in lower layers can have an impact on those residing in the immediately adjacent higher layer.

Table 1 summarises our comparative analysis of the three implementations.

Table 1: Comparative overview of the three implementations

Implementation Property	*Concern*	**AO Impl.**	**GAUDI**	**P2P Appl. Framework**
Modularity	*Adap.*	Yes	Yes	No
	Dist.	Yes	No	Low-level protocols only
Reusability	*Adap.*	Domain & Appl. Specific	Domain & Appl. Specific	None
	Dist.	Any content manipulator	No	Low-level protocols only
Maintainability	*Adap.*	Good	Average	Poor
	Dist.	Good	Poor	Poor
Evolvability	*Adap.*	Good	Average	Poor
	Dist.	Good	Poor	Poor
Complexity		Low	Medium	Medium

6. RELATED WORK

A number of middleware platforms have focused on support for adaptive, mobile and ubiquitous computing applications. Román and Campbell [28] propose a middleware-based application framework for the purpose. The framework is specifically geared towards device rich, mobile environments. Popovici et al. [25] discuss the use of the aspect-oriented middleware platform, PROSE, to support dynamic adaptation of mobile robots to different production needs. The Distributed Aspect and Object Platform, DAOP, [24] reifies the architecture specification, provided using its own architecture description language, which can then be adapted at runtime in line with the adaptation requirements of the application. The platform has been used to construct adaptive environments for collaborative work. All these platforms focus on supporting adaptation with distribution support provided by the middleware platform itself. Our application study of AOP is, therefore, complementary to these approaches as it focuses on evaluating a general purpose AOP technique, in this case AspectJ, to develop adaptive, distributed pervasive environments. In this sense, our AO implementation of the pervasive environment can be seen as a kind of middleware providing distribution and adaptation support for applications being deployed within the environment.

Brooks et al. [5] discuss aspect-oriented design of an adaptive sensor network supporting military applications. Their adaptive environment is developed using a custom-built, petri-net based solution while our comparative study is based on using general purpose AO and OO techniques. Furthermore, the nature of their sensor network, and applications supported by it, results in complex aspect interactions which requires a resolution model more elaborate than that of AspectJ. In case of our pervasive environment, the aspect interactions are fairly simple and can be easily handled and resolved by AspectJ.

Soares et al. [32] have focused on development of persistence and distribution aspects as separate increments to a system. Our experience provides further insight into the mechanics of such an incremental approach. The AO implementation of our pervasive environment shows that though such an incremental approach is viable, there has to be significant communication across the increments to avoid overhead of revisiting aspects developed in earlier increments.

Some researchers, e.g., Murphy et al. [21] and Baniassad et al. [4] have undertaken empirical studies of developers using AOP techniques. Our application experience is orthogonal to such studies as we analytically compare different implementations of the same environment. Ethnographic studies of such comparative implementations would provide interesting insights into the way developers approach the modularisation of crosscutting concerns both with and without AOP techniques.

Adaptation is a recurring theme in pervasive computing. A major thrust of systems-level research in pervasive computing is aimed at building context-aware systems [30, 31] that exhibit adaptive behaviour, i.e., systems that can adapt at runtime to the user's tasks and needs, and to the availability of system resources such as network bandwidth. The main strategy to achieve this goal is to provide generic system and application platforms with built-in adaptation capabilities. Examples of such platforms are the Context Toolkit [10], Context Fabric [14], Aura [12, 33] and OneWorld [2, 13]. Another approach is based on the use of explicit software architecture models to monitor a system and guide dynamic change to it. Cheng et al [6] use externalised models to make reconfiguration decisions based on a global perspective of the running system and to gauge their effectiveness through continuous system monitoring. A variation of the same architecture-based approach is the work on ArchJava [1, 29], an extension to the Java programming language that unifies software architecture with implementation. ArchJava can be thought of as an aspect-oriented programming language with support for separating crosscutting structural concerns from behavioural concerns. Both of these architecture-based approaches aim at enabling a system to self-repair or self-heal in order to recover from an unwanted system state. However, all of the above-mentioned approaches focus on short-term adaptation concerns; so far not much attention has been paid to post-deployment issues of pervasive systems such as maintainability, evolvability and long-term adaptation.

A number of approaches have examined how P2P technology can be modularised and abstracted. Due to the large number of P2P protocols that have been developed most of this has focused on low-level technological abstractions. Dabek et al. [9] abstract common APIs for structured P2P networks. PROST [26] builds on this and seeks to reduce the duplication of P2P routing functionality by extracting it into a separate layer that can then be reused by different applications. The Open Overlays project [22] takes this further by building abstractions for any type of P2P network technology. In all these cases (as with the P2P Application Framework in section 4.2) only the low-level functionality has been abstracted, meaning that the applications built on top would still possess a largely monolithic structure (for example, if having to implement functionality such as content manipulation and display adaptation, as discussed in this paper).

7. CONCLUSION

This paper has described our experience of using AOP, specifically AspectJ, to implement an adaptive peer-to-peer pervasive display environment. We have also undertaken two OO implementations of the same environment, developed completely independently of each other and the AO implementation. The three implementations give us a strong basis to compare the modularisation of two key crosscutting concerns in pervasive computing environments, adaptation and distribution. We have derived our comparison criteria from some of the key motivations behind AOP, i.e., the development of more modular, reusable, evolvable and maintainable representations of crosscutting concerns. At the same time, we have compared the three implementations for complexity of realising the above quality attributes with regards to adaptation and distribution. Our comparison clearly demonstrates that an AO approach facilitates modularisation of adaptation and distribution code in our pervasive environment in a manner which is more reusable, evolvable and maintainable compared to the two OO implementations. While the two OO approaches modularise some facets of each of the two concerns – an XML based content generation and transformation approach to modularise adaptation in GAUDI and an application framework modularising low-level protocols for P2P distribution in case of the P2P Application Framework based implementation – the AO approach does so in a manner that is less complex, avoids unwanted code bloat and is more intuitive to reuse, maintain and evolve.

Our experience also provides interesting insights into development of aspect-oriented applications. We can observe from the realisation of both the adaptation and distribution concerns that the notion of one single, large aspect module (in this case a single AspectJ aspect) encapsulating a crosscutting concern does not make sense. One needs to modularise different facets of a crosscutting concerns using abstractions most suited for the purpose, i.e., aspects, classes or interfaces, and the resulting *framework* that binds these facets together is, in fact, the *aspect* modularising the crosscutting concern. There is another argument for such an approach clearly visible from our implementation. Had we not separated application-independent and application-specific facets of adaptation using different AspectJ aspects, the changes to adaptation code required when the distribution aspect was introduced would have been much more difficult to achieve. Our fine-grained modularisation facilitated analysis of our design decisions and easy and effective implementation of any refactorings to the existing, aspectised adaptation code.

Our application experience also helps us better understand whether closely related aspects, such as adaptation and distribution, can indeed be developed in complete isolation in different system increments. We can see that, though this is an attractive proposition, in reality the semantics of such increments are too intertwined to allow strict isolation. In case of our pervasive environment, the design of our adaptation aspect framework would have been better informed had we taken into account the semantics of the distribution concern and the specific distribution architecture to be employed in the following increment. It is, therefore, clear that such *life cycle aspects*, that pertain to development guidelines and hence cut across development stages or increments, require significant attention from the AOSD community. Application studies similar to ours are a key to formulate a better understanding of such life cycle aspects.

We can also observe some interesting development styles for aspect-oriented applications. We have used interfaces as application-independent points of reference to decouple the aspect frameworks (modularising the crosscutting concerns) from the other concerns in the system. Similar, application-independent points of reference were employed by [27] to modularise persistence using AOP. We can see that such an approach works well for aspect-base decoupling especially to improve reusability, maintainability and evolvability of aspects implemented with approaches like AspectJ which, otherwise, require pointcuts to be specified with direct references to the signature of elements in the base. The use of application-independent points of reference offers a level of indirection to avoid such direct references hence significantly reducing, and in our case eliminating, the impact of changes to the signature of the base on the aspects and vice versa.

From a pervasive computing perspective, our application provides an opportunity to evaluate the suitability of an emerging development technique. One of the key points to note in our AO implementation is the focus on the more longer term qualities such as reusability, evolvability and maintainability. Most existing research in pervasive computing focuses on meeting the short-term adaptation needs of the applications and such long-term qualities are often ignored in system design. Our application brings forth AOP as a viable option to develop pervasive environments that are responsive to needs imposed by such long-term quality attributes without compromising the focus on short-term adaptability needs of applications.

Our future work will focus on studies of developers working on similar, independently developed, multiple implementations of systems, involving a variety of systems from a wide range of domains. This will not only provide further opportunities for comparative studies of the implementations but also make it possible for us to study how developers approach the modularisation of a crosscutting concern and how challenging the task becomes if AOP tools and techniques are not being employed. Such studies are a key to understanding the full potential of AO techniques.

8. ACKNOWLEDGEMENTS

The work presented in this paper is part of the ECampus initiative funded by Lancaster University. The authors wish to thank Chris Kray at Lancaster University for his contributions to the GAUDI implementation and Danny Hughes, also at Lancaster University, for helpful discussions.

9. REFERENCES

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Nokin, "Architecture-Centric Programming for Adaptive Systems", Proceedings of the Workshop on Self-Healing Systems. (WOSS'02), 2002.

[2] L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, G. Look, S. B. Sigurdsson, J. Su, and G. Borriello, "Systems Support for Ubiquitous Computing: A Case Study of Two Implementations of Labscape", Proceedings of International Conference on Pervasive Computing, 2002.

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

[4] E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher, "Managing Crosscutting Concerns during Software Evolution Tasks: An Inquisitive Study", 1st International Conference on Aspect-Oriented Software Development (AOSD), 2002, ACM, pp. 120-126.

[5] R. R. Brooks, M. Zhu, J. Lamb, and S. S. Iyengar, "Aspect-Oriented Deign of Sensor Networks", *Journal of Parallel and Distributed Computing*, Vol. 64, No. 7, pp. 853-865, 2004.

[6] S. Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu, "Software Architecture-based Adaptation for Pervasive Systems", International Conference on Architecture of Computing Systems Trends in Network and Pervasive Computing, 2002, Springer-Verlag, Lecture Notes in Computer Science, 2299.

[7] A. Colyer, G. S. Blair, and A. Rashid, "Managing Complexity in Middleware", Workshop on Aspects, Components and Patterns for Infrastructure Software (held in conjunction with AOSD 2003), 2003.

[8] A. Colyer and A. Clement, "Large-Scale AOSD for Middleware", 3rd International Conference on Aspect-Oriented Software Development (AOSD), 2004, ACM, pp. 56-65.

[9] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays", Proceedings of IPTPS, 2003, Springer-Verlag, Lecture Notes in Computer Science, 2735, pp. 33-44.

[10] A. K. Dey, D. Salber, and G. D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications", *Human-Computer Interaction*, Vol. 16, 2001.

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

[12] D. Garlan, et al. "Project Aura: Toward Distraction-Free Pervasive Computing", *IEEE Pervasive Computing*, Vol. 1, No. 2, pp. 22-31, 2002.

[13] R. Grimm, "One.world: Experiences with a Pervasive Computing Architecture", *IEEE Pervasive Computing*, Vol. 3, No. 3, 2004.

[14] J. I. Hong and J. A. Landay, "An Infrastructure Approach to Context-Aware Computing", *Human-Computer Interaction*, Vol. 16, 2001.

[15] I. Jacobson, "Use Cases and Aspects-Working Seamlessly Together", *Journal of Object Technology*, Vol. 2, No. 4, pp. 7-28, 2003.

[16] "JBoss Aspect Oriented Programming Webpage", http://www.jboss.org/products/aop, 2004.

[17] "JXTA v2.0 Protocols Specification", http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html, 2004.

[18] M. A. Kersten and G. C. Murphy, "Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming", OOPSLA, 1999, ACM, SIGPLAN Notices, 34(10), pp. 340-352.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP), 1997, Springer-Verlag, Lecture Notes in Computer Science, 1241, pp. 220-242.

[20] J. Kienzle and R. Guerraoui, "AOP: Does It Make Sense? The Case of Concurrency and Failures", European Conference on Object-Oriented Programming (ECOOP), 2002, Springer-Verlag, Lecture Notes in Computer Science, 2374, pp. 37-61.

[21] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad, "Evaluating Emerging Software Development Technologies: Lessons Learned from Evaluating Aspect-oriented Programming", *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 438-455, 1999.

[22] "Open Overlays Project: Component-Based Communications Support for the GRID." http://www.comp.lancs.ac.uk/computing/research/mpg/projects/openoverlays/, 2004.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 1-25.

[24] M. Pinto, L. Fuentes, and J. M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development", International Conference on Generative Programming and Component Engineering (GPCE), 2003, Springer-Verlag, Lecture Notes in Computer Science, 2830, pp. 118-137.

[25] A. Popovici, A. Frei, and G. Alonso, "A Proactive Middleware Platform for Mobile Computing", ACM/IFIP/USENIX International Middleware Conference, 2003, Springer-Verlag, Lecture Notes in Computer Science, 2672, pp. 455-473.

[26] M. Portmann, S. Ardon, P. Senac, and A. Seneviratne, "PROST: A Programmable Structured Peer-to-Peer Overlay Network", Proceedings of IEEE P2P, 2004, pp. 280-281.

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

[28] M. Román and R. H. Campbell, "A Middleware-Based Application Framework for Active Space Applications", ACM/IFIP/USENIX International Middleware Conference, 2003, Springer-Verlag, Lecture Notes in Computer Science, 2672, pp. 433-454.

[29] V. Sazawal and J. Aldrich, "Architecture-Centric Programming for Context-Aware Configuration", OOPSLA Workshop on Engineering Context-Aware Object-Oriented Systems and Environments (ECOOSE), 2002.

[30] B. N. Schilit, "A Context-Aware System Architecture for Mobile Distributed Computing", PhD Thesis, Columbia University, 1995.

[31] A. Smailagic, D. P. Siewiorek, and J. Anhalt, et al. "Towards Context Aware Computing: Experiences and Lessons Learned", *IEEE Journal on Intelligent Systems*, Vol. 16, No. 3, pp. 38-46, 2001.

[32] S. Soares, E. Laureano, and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ", ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2002, ACM Press, pp. 174-190.

[33] J. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments", Proceedings of 3rd IEEE/IFIP Conference on Software Architecture, 2002.

[34] J. Walkerdine, L. Melville, and I. Sommerville, "A Framework for P2P Application Development", Computing Department, Lancaster University, Technical Report No. COMP-004-2004, 2004.