Case Study: A Distributed Concurrent System with AspectJ

Rajeev R. Raje
Department of Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202
rraje@cs.iupui.edu

Ming Zhong
Department of Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202

Tongyu Wang
Department of Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202

ABSTRACT
This paper presents a simple three-tier client/server application using two prototypes. The first prototype makes the use of a traditional OOP language, Java. The second prototype uses the Aspect-Oriented Programming (AOP) language AspectJ. The advantages and disadvantages of developing a concurrent system using the AOP paradigm are evaluated, as well as the potential of the AOP paradigm.

Keywords
Aspect-oriented Programming, Concurrent Computations, Client-Server AspectJ

1. INTRODUCTION
Object-oriented programming has been a great success. The fundamental principles behind it, inheritance, encapsulation, and polymorphism, have allowed us to cleanly encapsulate related application data and logic, making code maintenance easier. However, it has been realized that objects alone do not provide mechanisms for dealing with systematic concerns such as synchronization, multi-object protocols, resource sharing, distribution, memory management, replication, error handling, and so on. These concerns tend to cross-cut the system's classes and modules. This means that the implementation of these kinds of concerns is usually scattered throughout the code, making the system more difficult to maintain.

In order to resolve this problem, it has been suggested that the traditional programming paradigms can be extended with the notion of aspects. The resulting paradigms are called Aspect-oriented Programming. In this paper, we will examine this paradigm in the context of a small, real-world, distributed application. Section 2 discusses what AOP is and how it achieves its goals. In section 3, we will take a look at synchronization and how it can benefit from AOP. Section 4 presents the application in detail. Section 5 will discuss the results. In section 6 we will make some concluding remarks.

2. ASPECT-ORIENTED PROGRAMMING
AOP was first proposed by Gregor Kiczales et al at Xerox PARC in 1997. In order to explore ideas, AspectJ, the first AOP language, was implemented. The current state of AOP research is similar to that of OOP 20 years ago[1]. Researchers are still working to form the basic concepts of AOP, and to find new applications of the concepts.

AOP introduces the concepts of an Aspect and Weaving to provide a better way to manage the handling of cross-cutting concerns. An aspect is a new unit of software modularity that encapsulates state (roughly, variables), behavior (roughly, methods), and behavior enhancements in other units (roughly, weaves). Aspects tend to be units that cross-cut several components in the design, according to some natural division of labor, and they interact with the components according to well-defined interfaces. Weaving is the systematic process of combining aspects and objects. It can be done with an interpreter, a compiler or a pre-processor. The overall process can be seen in Figure 1.

Aspects can be used in both design and implementation phases in a software development life cycle. During the design phase, the concept of aspects facilitates thinking about cross-cutting concerns as well-defined entities. During implementation phase, Aspect-Oriented Programming languages make it possible to program directly in terms of design aspects, just as object-oriented languages have made it possible to program directly in terms of design objects. Aspects cross-cut the modularity of classes so that an aspect can affect the implementation of several classes (or several methods within a single class) in a clean, organized way. A special tool called weaver automatically merges the aspects with the regular classes on which they intend to have effects.

2.1 AspectJ
AspectJ provides the functionality described in the previous section by extending the Java programming language with the notion of Aspects and crosscuts. A detailed description of AspectJ is presented in many papers at the AspectJ

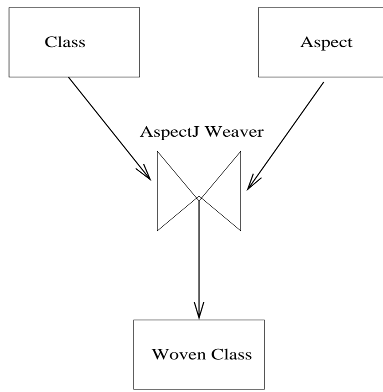

Figure 1: Aspect weaver weaves aspect and normal class together

site [2, 3, 4]. Descriptions of other aspect-based projects is found at [5, 6]. For the sake of brevity, here we provide a brief introduction to AspectJ. This is based on [3, 4].

AspectJ is a superset of Java, created by adding general purpose aspect-oriented extensions to Java. An aspect consists of a name, ordinary variables, methods, *crosscuts* and *crosscut actions*, which may be *advice* or *introduction*. Hence, aspects classes[1] are like Java classes, except that they may contain the crosscut constructs.

2.1.1 Crosscuts

AspectJ's crosscuts capture collections of events in the execution of the program. These events can be methods invocations, constructor invocations and signaling/handling of exceptions [3]. Crosscuts do not describe options; they simply describe events [4]. A simple crosscut could be:

```
crosscut reduce(): Account & (void debit(int));
```

This crosscut describes the reception of debit(int) message by any object of type Account.

A crosscuts consists of a left-hand side, that defines the crosscut name and the context; and a right-hand side, that defines the events in that crosscut. The descriptors of the event in the right side of a crosscut are called as *designators*. Many different types of designators exists, for details refer to [4].

2.1.2 Crosscut Actions

There are two types of crosscut actions: a) advice, and b) introduction.

Advice:

[1]The earlier versions of AspectJ used the keyword *aspect* to denote classes of aspects. From version 0.5beta1 onwards the *class* keyword is restored [3]. As the system presented in this paper was developed before this change took place, the code at the end of this paper uses the keyword *aspect* to denote aspect classes.

Advice declarations define places of aspect implementation that execute at well-defined points. These points can be indicated either by named crosscut or by an anonymous crosscuts. A named advice on a crosscuts may look like:

```
crosscut reduce(Account a1, int value): a1 & (void
debit(value));

advice (Account a1, int value): reduce(a1, value){
before {System.out.println("Ready to debit
to the account");}}
```

While the advice on an anonymous crosscut will be:

```
advice (Account a1, int value): a1 & (void debit(value)) {
before {System.out.println("Ready to debit
to the account");}}
```

The `before` advice (as shown above) will run just before the the execution of the actions associated with the events in the crosscut, i.e., before the *debit(int)* function is executed.

There are five types of advices: *before, after, catch, finally* and *around*. The semantics of them are as suggested by their names. More details are found in [4].

Introduction:

Introduction declarations introduce whole new elements in the given classes. For example:

```
introduction Account {int getBalance() {return
balance;}}
```

This introduces a method named *getBalance* in a class named *Account*. This method has no arguments and returns an integer.

In addition to methods, an `introduction` declaration can introduce a constructor and a field.

More detailed description of the new concepts and other issues, such as a creation of aspects instances, etc., are found in [4].

3. DISTRIBUTED CONCURRENT PROGRAMMING

Concurrency control is always a complicated issue for any Distributed Computing System. Java Remote Method Invocation (RMI) allows programmers to develop distributed programs with native Java syntax. Since remote objects may be accessed concurrently, programmers should take care to implement their classes and methods so that they are thread-safe, in the same way that Java run-time libraries are thread-safe.

The Java language itself provides some basic primitives for synchronization. The keyword *synchronized* can be used

both as a statement and as a method modifier. It implements monitors, which are associated with objects. If methods are declared to be synchronized, they can not run concurrently by more than one thread, enabling them to maintain consistent state in the shared variables. When a synchronized method is entered, it acquires the monitor associated with the object. The monitor precludes any other synchronized methods in that object from running. When a synchronized method returns by any means, its monitor is released. Other synchronized methods within the same object are then free to run. [7, 8].

Only simple synchronization schemes can be achieved using Java's native synchronized methods. More complex problems, like the classic reader/writer problem, require checking different conditions and either waiting or proceeding accordingly. Java provides language level waiting and notifying constructs: *wait(), notify()* and *notifyAll()*. While these constructs give programmers more control over what synchronization schemes to choose, they also make the design and implementation complicated. Since the synchronization is usually not a functionality requirement of the application, but rather a mandate for correctness of the application, it usually affects a significant number of modules in the system. Hence, typically the synchronization code would be scattered and repeated across all affected modules. The result is a tangled code that is difficult to program and maintain. In AOPs terminology, it is a cross-cutting concern of the system. It is ideal to focus on the functionality of the application and let the synchronization be taken care of in a separate module. This is where Aspect-Oriented Programming comes in handy. The design and implementation of the synchronization can be done as an aspect. This aspect is explicit in the design phase of the application development and can be implemented separately and later plugged-in to the main modules of the application, which do not have any synchronization consideration.

4. HSN CUSTOMER SERVICE SYSTEM PROTOTYPE

In this paper, we will study and evaluate the AspectJ and Java languages within the context of a Home Shopping Network (HSN) application. There are many commercial products which use a similar architecture, however, here we present a generic architecture for a HSN.

The HSN Customer Service System is the main component that serves all service related customer requests, including account establishment, account maintenance, order inquiries, order changes, returns, refunds, and so on. HSN involves multiple platforms, multiple data sources and is in a highly distributed environment. It is a very good example of 3-tier client/server architecture that involves a legacy system. The first tier is the client side front-end. The front end application runs on desktop PCs in the HSNs huge call center. The middle tier consists of the servers that are driven by the clients and also custom-made server objects that serve as a bridge to the legacy system. In the third tier are relational databases that support the servers and the mainframe that houses almost 95% of the HSNs business logic. This architecture provides improved performance over a mainframe alone and also allows great flexibility in legacy system migration and future extensions.

Given the sheer magnitude of this system, we chose to implement only the Product Information subsystem as a testbed for evaluating the AspectJ programming language. This Product Information System is required to provide customer service agents the product information, from the mainframe, of the item currently being broadcasted on the air as well as the items that customers request.

4.1 Design Details

The Product Information System was designed using 3-tier distributed client/server architecture and was implemented in both Java/RMI and AspectJ.

The first tier is thin client layer. It provides a user with a graphical interface to retrieve the information of an on-air item and retrieve detailed information on a certain item from the mainframe database.

The middle tier consists of the server objects that accept client requests, access the mainframe to get the information needed and return the results to the clients. The client and server objects communicate through Java RMI.

The third tier is a backend-main frame, which stores the majority of the business logic and item information. In this system, we will use an Oracle database to simulate the main frame database. The middle tier server objects get persistent data from a JDBC (Java Database Connection) compliant Oracle databases.

This architecture provides better performance, flexibility, extensibility and fault tolerance than the equivalent two-tier architecture. The thin client carries minimum business logic and runs on cheap, less powerful PC. This approach prevents the ever-changing system requirement from having too much of an effect on the client. The server objects run on powerful servers to provide maximum performance and scalability. They also make the data source transparent to the client. Therefore, reliability and availability of the data can be achieved through redundancy of data source. Any change to the data source, e.g., legacy migration, will not have much impact on the client layer. The architecture diagram is showed Figure2.

There are two different server objects needed in this system. One is the *ItemOnAir* service object. It provides the remote interface *readBuffer* for the client to view the item that currently being broadcasted on air. The server object contains a shared buffer to store this item information. The server object will constantly update this data field with the current item showing on the air from the mainframe. Concurrently, multiple clients may read the data from this data field. An access control must be ensured so that when the server object is updating the data field, all clients must wait.

The second server object is the *ItemInfo* service object. This server object can provide the remote interface for client to supply the Universal Product Code (UPC) of a product and to return the detailed information about the product identified by the UPC. The server object will go to the back-end mainframe to retrieve the detailed information and return the result back to the client. Since the database connection is very expensive, the server object maintains only one

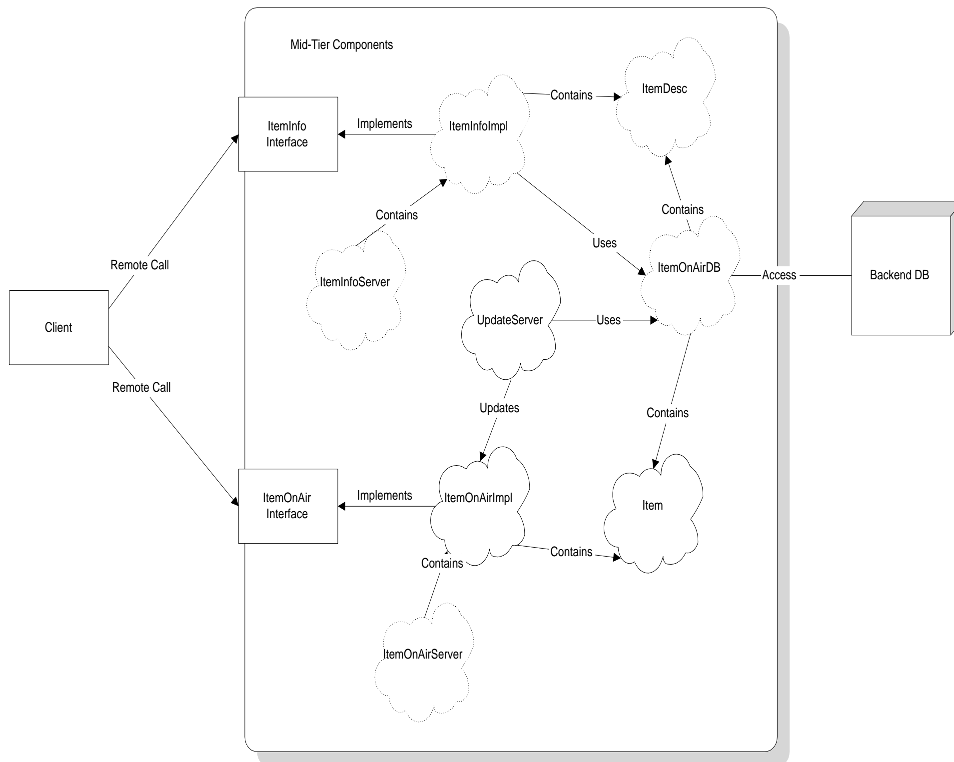

Figure 2: Design diagram of the HSN Customer Service System – Product Information Subsystem.

database connection throughout its life cycle. Querying the database for every request is unnecessary since the product information rarely changes and clients may request the same product information repeatedly over time. Because of this, we want to cache the item information. When the server object receives a request to get the item information, it first searches the cache. If the item is found in the cache, the server will return the item information directly to the client. Otherwise, the server will make a request to the database to get the item information from backend mainframe. Then it will put this information into the cache so that the next time the same item is requested, there will be no need to query the mainframe again. The cache will not be able to hold an unlimited amount of data. Therefore, the writer has to erase the oldest data first, and then store the new data in the cache when the cache becomes full.

According to the Java RMI specification, multiple threads could be accessing a server object concurrently. The buffer is shared by all of these threads. Hence, in the design and implementation of both server objects, the access control should be used to ensure such that no thread is reading the cache while another thread is updating the cache, and no more than one thread can update the cache at the same time.

4.2 Implementation details

4.2.1 Functionality Implementation

The client and server communicate with each other using Java/RMI. Two interfaces: ItemOnAir and ItemInfo are defined. On the server side, the ItemOnAirImpl class implements the ItemOnAir interface and ItemInfoImpl class implements the ItemInfo interface. The client uses both interfaces to make requests to the two server objects. Since RMI can handle multiple client requests at a time, we only need one instance for each of the server object class. The RMI system handles the runtime thread creation and request dispatching.

4.2.2 Database Implementation

The oracle database system simulates a mainframe database that stores general product information: UPC, product short description, retail value, sell price, original show date, date last purchased, quantity available, weight, and a long description. We created the entity relationship in Third Normal Form. Two tables have been created: sItemInfo and eItemInfo. The sItemInfo only stores the basic information of items, and eItemInfo stores the extend information of items. The ItemInfoView joins the two tables to get the full description of the product items.

4.2.3 Synchronization Scheme

The shared buffer problem can not be handled properly by a simple synchronized method. The reader and write should be mutually exclusive. Writers are self-exclusive while readers are not. This means that multiple readers should be allowed to access the buffer simultaneously. The synchronized method prevents this. In fact, methods that are declared synchronized are all exclusive to each other. Therefore, we decided to use language level Waiting and Notifying constructs: wait(), notify() and notifyAll() provided by Java to have more control over synchronization scheme. We

use synchronized blocks instead of synchronized methods for concurrency control. Our implementation is similar to the crowd monitor scheme.

In ItemOnAir service object, the buffer is a sharing variable. Since there should only one writer updating the buffer, we use three conditional variables: *activeReader*, *activeWriter*, and *WriterIsWaiting* to ensure the concurrency control and to avoid starvation and deadlock. The scheme works in the following way:

When the writer wants to update the buffer, it will check the *activeReader* count to see whether there is a reader reading the buffer. If there is, it will set *WriteIsWaiting* to be true, which will ensure that no more new readers enter to the critical section, and then sleeps. This synchronization scheme is to avoid writer starvation when there are significantly more non-self exclusive readers then writers.

When the writer leaves critical section, it will set *WriterIsWaiting* to false, decrease the *activeWriter* number and notify all, so that all the readers can be waken up and enter the critical section.

When a reader wants to read the buffer, it also needs to check both *activeWriter* and *WriterIsWaiting* to see whether there is writer updating the buffer or a writer is waiting. The *writerIsWaiting* flag is used to avoid writer starvation. We gave writer higher priority than reader. When the writer leaves the critical section, it will decrease the *activeWriter* and set *writerIsWating* to be false. If the waiting writer is woken up and check the *activeReader* and found that some reader has already enter the critical section, the writer has to wait again. However, since the *writerIsWaiting* now is true, the new readers can not enter the critical section, which will ensure there is no writer starvation.

When the reader leaves the critical region, it decrements the *activeReader* count.

In ItemInfo service object, the cache table that stores some products information is sharing data. This table is implemented as a vector. Writer can add an element to the vector or remove an element to the vector if the vector is filled to the maximum number of items. The synchronization scheme is similar to the one used on the ItemOnAir service object.

5. RESULTS AND DISCUSSIONS

5.1 Comparison of Conventional Java/RMI and AspectJ Implementations

The conventional Java/RMI gives programmer more flexibility on the synchronization schemes that can be implemented. However, the cross-cutting nature of synchronization, with synchronization code scattered throughout the program, shows in the code segments in the Item OnAirImpl class. Figure 3 displays the code for the RMI implementation of ItemOnAirImpl. Synchronization code is shown in Italic.

The AspectJ code is much cleaner looking and more importantly, extracts the synchronization concern into a different module: Aspect Coordination. This approach enables programmers to implement functionality separately from synchronization. The benefits are enormous. First, the code looks cleaner and much shorter in the AspectJ implementation. Second, and more important, now we can implement a simple synchronization scheme first and change it later on without affecting the functionality module. Separation-of-concern makes the program more immune to future changes one of the fundamental goal of OOP. Figure 4 shows the code segments in the ItemOnAirImpl class and Aspect Coordination. Synchronization code is shown in Italic.

6. CONCLUSIONS AND FUTURE EXTENSIONS

Aspect-Oriented Programming introduces the concepts of Aspect and Weaving to provide a better handling of crosscutting concerns. It is a promising new programming paradigm that can really excel where OOP comes short.

AspectJ provides a compiler, `ajc`, which is like any other compiler, except it has the additional step of coordinating the crosscutting modularities of aspects, i.e., the “weaving” step. Thus, it weaves classes and aspects together. This weaving happens solely at compile time. A normal Java compiler (current version of ajc uses JDK javac compiler) has to be used to further compile the woven classes into class files. The aspects will be merged (disappear) into the woven classes. This can have several drawbacks:

- It is difficult to debug aspects since they do not exist at runtime.
- Object can not choose to have certain aspects at runtime (dynamic binding) since aspects are not separate entities at runtime.

The ajc compiler outputs two additional sets of files – one indicating the result of weaving and the other being the data files that describe the results of the waver to the IDE tools. However, it still does not have a good IDE support to make the programming convenient for the developer.

We investigated Aspect Oriented Programming as a new programming paradigm that addresses cross-cutting concerns in system design and implementations. We also implement a subsystem prototype for Home Shopping Networks Customer Service System using both the conventional Java programming language and AspectJ. We compared and contrasted the AOP with OOP in general and also discussed AspectJs extension to Java.

In our experience, AspectJ did provide better separate-of-concern in the design phase of the project and also made the implementation much cleaner than using the conventional Java programming language. We did, however, find AspectJ inadequate for solving real world problems because of lacking language constructs, and a simplified compiler implementation (weaver).

For instance, in order to make use of AspectJ for our synchronization, we had to modify our initial decomposition of the functions. In the original code for ItemInfoImpl we had a

single function getItemDesc() which located the item within the array and read it. Using the synchronization methods we used, in order to implement this within AspectJ, we had to decompose this function into two separate functions, one to do the locating, and another to read the array. This was because in the original we had to have synchronization concerns in the middle of the function, which AspectJ will not allow currently. This is a result of AspectJs ability to weave code only into specific places, such as the beginning/end of a function, or in an exception handler. In the future, AspectJ will have to adopt a much finer grained notion of where cross-cuts can occur.

Future work will be focusing on proposing new language features to make the language more expressive. Current version of AspectJ only has two constructs: *advice* and *introduction*, that enable programmers to add new statements and new methods into an existing class. We found it necessary to have finer control over where the new statements are added. For example, a log may be entered to the log file whenever a variables value is changed (appeared on the left side of the assignment). A new language construct, adviseOnChange, will add a logging statement wherever necessary in the code. Another possible construct is adviseOnException, which would put a try-catch block around a statement whenever a statement throws a certain exception.

AspectJ, as a proof-of-concept prototype, works fine on solving simple, less demanding problems. AspectJ does provide better encapsulation and separation-of-concern over the conventional Java language alone. However, the language has a limited number of semantic constructs to enable programmers to express complex aspects encountered in real world applications.

Other work related to Aspect-Oriented Programming can be found at [5, 6].

7. ADDITIONAL AUTHORS

Additional authors: Joseph Williams, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, IN 46202

8. REFERENCES

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vldeira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP)*, 1997.

[2] http://www.aspectj.org.

[3] Anatomy of an Aspect in AspectJ. http://www.aspect.org/documentation/primer/basics/anatomy.html.

[4] The basics of the AspectJ Language. http://www.aspect.org/documentation/primer/basics/aspectjlang.html.

[5] Kersten, M., Murphy, G. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 1999 ACM Conference – OOPSLA*, pages 340–352. ACM, 1999.

[6] Kendall, E. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 1999 ACM Conference – OOPSLA*, pages 340–369. ACM, 1999.

[7] http://java.sun.com/docs/white/langenv/Threaded.doc2.html.

[8] Concurrent Programming in Java(TM). http://gee.cs.oswego.edu/dl/cpj/JavaSummary.html.

Java Code

```
public class ItemOnAirImpl extends UnicastRemoteObject
implements ItemOnAir
{
    private Item buff = new Item();
    // Variables to keep track of synchronization concerns,
    // Constructor
    public void writebuffer(Item str) throws RemoteException {
        beforeWrite();
        // Copy str into instance variable buf.
        afterWrite();
    }
    public Item readbuffer() throws RemoteException {
        Item newItem = new Item();
        synchronized (this) {
            while(activeWriter > 0 || writerIsWaiting)
                try { wait(); }
                catch (InterruptedException e) {
                System.out.println(e);
                }
            ++activeReader;
        }
        // Copy buff into newItem
        synchronized (this){
            --activeReader;
            notifyAll();
        }
    return newItem;
    }
    protected synchronized void beforeWrite() {
        while (activeReader> 0)
            try{
            writerIsWaiting = true;
            wait();
        }
        catch (InterruptedException e){}
        activeWriter++;
    }
    protected synchronized void afterWrite() {
        --activeWriter;
        writerIsWaiting = false;
        notifyAll();
    }
}
```

Figure 3: Conventional Java/RMI implementation of the ItemOnAir service object. Italicized code is related to synchronization.

AspectJ Code

```
public class ItemOnAirImpl extends UnicastRemoteObject
implements ItemOnAir
{
    private Item buff = new Item();
    // Variables to keep track of synchronization concerns
    // Constructor
    public void writebuffer(Item str) throws RemoteException {
        // Copy str into variable buff
    }
    public Item readbuffer() throws RemoteException {
        Item newItem = new Item();
        // Copy buff into newItem
        return newItem;
    }
}
aspect coordination {
    advice * writebuffer(..) & ItemOnAirImpl {
        before {
            while (activeReader> 0
                try {
                writerIsWaiting = true;
                wait();
                }
                catch (InterruptedException e){}
            activeWriter++;
        }
        after {
            --activeWriter;
            writerIsWaiting = false;
            notifyAll();
        }
    }
    advice * readbuffer(..) & ItemOnAirImpl {
        before {
            synchronized (this){
                while(activeWriter> 0 || writerIsWaiting)
                    try {
                    wait();
                    }
                    catch (InterruptedException e) { }
                ++activeReader;
            }
        }
        after {
            synchronized (this){
                --activeReader;
                notifyAll();
            }
        }
    } }
```

Figure 4: AspectJ implementation of the ItemOnAirImpl class and the Coordination aspect.