Middleware Transparency through Aspect-Oriented Programming using AspectJ and Jini

Devon Simmonds, Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, Colorado, 80523, USA

ABSTRACT
While distributed middleware technologies, such as CORBA, DCOM and Jini have enabled the development of complex software applications in varied domains, they have also presented challenges. One major challenge is middleware transparency — abstractions that capture those elements of the application specific to the middleware and allow seamless integration of the abstracted elements into an application. Aspect oriented programming (AOP) promises to make the possibility of achieving middleware transparency a reality. This paper describes our attempt to realize middleware transparency by applying AOP with AspectJ to a distributed Jini “stock broker service” application. We successfully encapsulated Jini code into a number of aspects that greatly simplified the application development process. In addition, we also implemented a number of visualization utility aspects having applicability in a variety of different applications and scenarios. We describe both the benefits and challenges of this approach.

Keywords: Aspect oriented programming, Jini, middleware, distributed software, transparency

1. INTRODUCTION
Development of distributed software has evolved to a large extent over the years and there exist a number of distributed middleware technologies, such as CORBA, DCOM, COM+, Java RMI and Jini. These technologies share some basic features and also offer other diverse features. Developers need to be aware of subtle differences even while using the same feature in an application. At a high level of abstraction, the basic design of the application may be similar, and, perhaps, independent of the middleware. Detailed designs and implementations, however, need to take into account specifics of the middleware technology.

In our work we address the problem of middleware transparency. Ideally one would like to use abstractions that capture elements in the application specific to the middleware, and use techniques that allow seamless integration of the abstracted elements into the application.

A new paradigm has evolved to address prevalent crosscutting concerns experienced by software developers in their profession. The paradigm is called Aspect Oriented Programming (AOP) or Aspect Oriented Software Development (AOSD). AOP is a way of designing and coding software using a component language (to design and code components), an aspect language (to design and code aspects) and a weaving mechanism to place the aspects in the appropriate places in the components they crosscut. AOP promises to make the possibility of achieving middleware transparency a reality. This paper describes our attempt to realize middleware transparency by applying AOP with AspectJ to a distributed Jini stock broker service application. We successfully encapsulated Jini code into a number of aspects that greatly simplified the application development process. In addition, we also implemented a number of visualization utility aspects having applicability in a variety of different applications and scenarios. We describe both the benefits and challenges of this approach.

The remainder of the paper is organized as follows. We give a brief background on AOP and AspectJ semantics in Section 2. We present our case study on using AspectJ to describe crosscutting concerns in a Jini application in Section 3. We present our conclusions and outline directions for future work in Section 4.

2. BACKGROUND
In this section we present the necessary background material for this paper. We describe the nature of crosscutting concerns and how AOP addresses the separation of concerns. We also provide a quick overview of AspectJ semantics.
2.1. CROSSCUTTING CONCERNS
Traditionally, software requirements are classified as functional and non-functional. Functional requirements deal with the application's functionality while non-functional requirements tend to address items affecting the final application but which cannot be expressed as something which the software will do. Non-functional requirements include performance measures, user interface peculiarities and others. The concept of a cross-cutting concern while not identical, is closely akin to that of non-functional requirements. Crosscutting concerns are elements of a software which cannot be expressed in any functional unit of the programming language’s abstractions.

In object-oriented programming parlance, crosscutting concerns are elements of an application which cannot be cleanly captured in a method or class and so has to be scattered across many classes and methods. Such concerns include:

1. Applying design patterns
2. Applying synchronization policies
3. Applying exception handling, error-checking or fault tolerance concerns
4. Sharing resources
5. Security issues
6. Performance measures

The success of AOP will depend in large measure on a clear understanding of crosscutting concerns and the all important question - do crosscutting concerns warrant a new programming paradigm? The consensus at the moment seems to be a reserved ‘yes’. That there are crosscutting concerns is accepted, but what needs to be established is that the object-oriented paradigm cannot properly address this phenomena. Issues that need to be explored and quantified include inheritance versus aspects composition, and problems and issues of crosscutting concerns in object-oriented systems.

2.2. COMPONENT AND ASPECT LANGUAGES
In one of the earliest and seminal papers on aspect oriented programming,\(^5\) a clear distinction is made between components and aspects. A property to be implemented in software is a component if it can be cleanly encapsulated in a generalized procedure (i.e. object method, procedure, API). A property to be implemented in software is an aspect if it cannot be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API). The term cleanly is used to mean well-localized, and easily accessed and composed as necessary.

Languages are needed to implement components and aspects. There are several open issues in the design of these two languages. In order to reduce complexity, minimize conflicts and enhance usability the following principles should guide the development of these languages.

1. Although both languages must have different abstractions and programming mechanisms, they must also share an abstraction commonality. This commonality will be the basis on which weaving is done.
2. Each language should address different concerns for the abstraction commonality. This will help minimize conflicts.
3. Each language must address its specific objective: the aspect language addresses the specification of aspects, while the component language addresses the specification of functional requirements.

2.3. ASPECT WEAVING
The necessity of weaving in AOP arises from the following facts:

1. Aspects are abstraction mechanisms that must be captured by some generalizable procedure. For example, in the AspectJ language, aspects are class-like abstractions.
2. Aspects crosscut applications and some mechanism must be found to distribute the aspect code to the appropriate places in the components.

An aspect weaver takes a component program and an aspect program and produces as output a program with the aspect code appropriately distributed. Aspect weaving requires join points. A join point is an element of the component language semantics where weaving is coordinated or done. A basic aspect-weaving algorithm consist of two parts:
1. Create a join point representation of the component program.
2. Execute or compile the aspects program in accordance with this representation.

Join points would typically be expected to include:

1. Method calls
2. Handling of exceptions
3. Assignments and access of program variables and objects

2.4. ASPECTS AND DISTRIBUTED MIDDLEWARE

There is a clear and natural appeal of AOP to distributed applications. This natural appeal stems from the fact that flexibility, ease of use, and usability (among other attributes) of distributed applications engender the evolution of many middleware technologies. Developers are required to not only be familiar with the target programming languages and application domains, but also with the syntax and semantics of a variety of middleware. Code for middleware, such as CORBA or Jini, are natural crosscutting concerns and have to be applied to many elements of both client applications and service applications. AOP allows programmers to capture middleware code in aspects in such a way as to eliminate the need for programmers to have to know middleware syntax and semantics in order to compose distributed applications. The general objective of our work is to explore the extent to which this is possible for Sun’s Jini framework. Middleware transparency is a research area of interest to several stake-holders in the distributed applications community. Research in this area is also important because it seeks to apply an exciting new paradigm to an important, increasingly difficult, and increasingly pervasive problem.

2.5. ASPECT ORIENTED PROGRAMMING

Kiczales et al. sought to clarify the meaning of AOP and gave a brief outline of its major tenets. Most of the current literature focuses on the challenges of crosscutting concerns or the separation of concerns. Some deal with the benefits of using AOP and the implications of AOP on software design. AOP is still in its infancy and it is not clear the extent to which it actually works. Most persons are now familiarizing themselves with the basic tenets, challenges and potential of AOP. It is probably for this reason that there is only a sprinkling of research literature available on the application of AOP (AspectJ) to distributed software development using middleware such as CORBA. These papers indicate that the researchers have successfully encapsulated some parts of CORBA services in aspects. The basic objective is to make CORBA programming transparent to programmers. Hunleth, Cytron and Gill goes further and suggests the creating of an AspectIDL for CORBA to complement the IDL’s now available for languages such as Java, C++, etc. They also introduce a new form of crosscutting concern - those that crosscut programming languages.

A variety of AOP tools and applications are evolving at the time of writing this paper. These include AspectC, AspectC++, AspectArchitect, MixJuice (Java extension), a-kernel (OS application), FACET (realtime framework), and AspectJ (Java extension). We used AspectJ as the AOP programming language for our work.

2.6. ASPECTJ SEMANTICS

The AspectJ language provides a compiler which produces Java code or Java bytecode. AspectJ is the aspect language and Java is the corresponding component language. AspectJ assumes all the programming abstractions available in Java. In addition, it adds abstractions needed to express crosscutting concerns. The major abstractions presented by AspectJ are aspects, join points and pointcuts, advice, and introduction.

An aspect in AspectJ is similar to a class in Java. It is a major abstraction in the language and is meant for use as a vehicle to express crosscutting concerns. Aspects are composed of pointcuts, advice, and introduction.

2.6.1. POINTCUTS

AspectJ uses pointcuts to specify join points. There are a number of pointcut designators that are used to construct pointcuts. These include:

1. execution(method signature) - indicates when a given method body executes,
   e.g.: execution(void Sort(int[])).
2. call(method signature) - indicates when a given method is called, e.g. call(Sort(list)).
3. this(someType) - indicates when the object currently executing is of type `someType', e.g: this(HelloWorldService).
4. target(someType) - indicates when the target object is of type someType, e.g.: target(HelloWorldService), or target(p).
5. within(someClass) - indicates when the executing code belongs to class someclass.

We used the following pointcuts in our implementation of a stock broker service:

1. Pointcut to indicate when the GUI for the service class is invoked.

   pointcut GUI(StockBrokerInterface obj):
   target (obj) & call (* showGUI());

2. Two pointcuts to create a service monitor for each client and server.

   pointcut ClientBuy(int id, String stock, int shares):
   target (StockClient) & args(id, stock, shares) &
   call (* buy(int, String, int));

   pointcut BrokerBuy(int id, String stock, int shares):
   target (Broker) & args(id, stock, shares) &
   call (* updateClientResults(int, String, int));

The this designator can be used to resolve context conflict similarly to the scope resolution operator in C++ or the this operator of Java. This will be clearly seen when we give examples of introductions below.

2.6.2. ADVICE

AspectJ uses advice as a means of specifying code to be executed at join points. As stated above join points are specified using pointcuts. AspectJ provides three form of advice: 1) before advice, 2) after advice, and 3) around advice.

Before advice is executed when the specified join point given by the stated pointcut is reached and before any computations proceed, i.e. before the method starts executing. `After advice' is executed after the method of the join point is executed but before control is returned to the calling method or program. `Around advice' is executed when the specified join point given by the stated pointcut is reached and determines if the code under the join point is executed. Some pointcuts and advice for our application can now be shown together.

Two pointcuts and their corresponding advice are shown below:

   pointcut ClientBuy(int id, String stock, int shares):
   target (StockClient) & args(id, stock, shares) &
   call (* buy(int, String, int));

   pointcut BrokerBuy(int id, String stock, int shares):
   target (Broker) & args(id, stock, shares) &
   call (* updateClientResults(int, String, int));

   before(int id, String stock, int shares): ClientBuy(id, stock, shares) {
   try {
      this.updateScreen("Client "+id+" is buying "+ stock+ " shares");
   } catch (Exception ex){}
   }
after(int id, String stock, int shares): BrokerBuy(id, stock, shares)
{
    try {
        this.updateScreen(""+shares+" shares sold to Client "+id);
    } catch (Exception ex){}
}

2.6.3. INTRODUCTION
An introduction is AspectJ’s mechanism of modifying classes and their hierarchy by introducing or inserting new code into a target class of an application, or by changing the inheritance or interface hierarchy of a class or collection of classes. Compared to an advice which is generally dynamic, introductions are static and are executed at compile time. A typical problem for which aspect introduction is well suited is the classic problem in object-oriented languages (such as Java) which do not allow multiple inheritance. In order to add new functionality to a class that is already in an inheritance hierarchy, since multiple inheritance is not allowed, the only alternative until now was to create a new interface and allow the class in question to implement it. AspectJ’s solution is simple: create an aspect that introduces into the target class the methods and data items needed to implement the desired functionality. Introduction can be used to:

1. Add data variables to a class:
   
   ```java
   private int StockClient.numberOfStocks = 0;
   ```
   
   This adds a new integer variable to class StockClient. This assumes of course that a variable with an identical name does not already exist. A special naming convention is needed to prevent conflicts of this type.

2. Add methods to a class:
   
   ```java
   public StockClient.new(String message);
   ```
   
   This introduces a new constructor into class StockClient, while a new method is introduced by:

   ```java
   public StockClient.RegisterWithLookup();
   ```

3. Modify the inheritance or interface hierarchy of a class:
   
   ```java
   declare parents: StockClient implements newServiceInterface;
   declare parents: StockClient extends Frame;
   ```
   
   They would cause class StockClient to implement the stated interface and extend the Frame class respectively.

3. CASE STUDY
The appeal of middlewares such as CORBA and Jini is that they provide a variety of services and features intended to make program development consistent and transparent to developers. Middleware code crosscut application classes and are scattered across multiple methods. In addition client classes tend to have a specific set of middleware code while service classes have their own set. It seem only natural to apply AOP to the development of software by encapsulating in aspects code specific to clients and services. This is the approach taken for our work. Our approach provides the following benefits:

1. It solves the multiple inheritance problem in languages such as Java. Even where multiple inheritance is not a problem, it provides greater flexibility, by minimizing (it can, in fact, prevent) the occurrence of this problem in the future. This allows a developer to use inheritance in the most judicious way.
2. It promotes reusability, allowing developers to compose distributed applications with minimal knowledge of a specific middleware, in this case Jini.

Based on our work and the survey of relevant literature, we give an outline of the three general categories of ways that AOP can benefit distributed programming using Jini.

1. creating a general purpose set of aspects for use by clients and servers,
2. creating aspects specific to the needs of clients, and
3. creating aspects specific to the needs of servers.

3.1. A SET OF GENERAL PURPOSE ASPECTS

These aspects are needed to deal with a variety of issues not specific to the middleware but which are useful for the application domain. An example provided by Viega and Voas and implemented in our application is the following:

```java
public aspect ExceptionPrinter {
    pointcut allMethods(): execution(* *(..));
    static after() throwing (Exception e): allMethods() {
        System.out.println("Uncaught exception: " + e);
    }
}
```

Since many exceptions of all kinds are typically thrown in a distributed application which can suffer from communication and other failures, and interruptions of a variety of forms, an aspect to trap all uncaught exception is very useful. There are a variety of other general-purpose aspects which may be useful for applications. In our work, one aspect was a service monitor. A service monitor is an aspect that monitors interaction between clients and servers. Each client and each server may chose its personal monitor. The general purpose monitor used in this research is as follows:

```java
public aspect ServiceMonitorAspect.java implements Runnable {

    TextArea result, ownList;
    Frame myFrame;

    public ServiceMonitorAspect.java() {
        createGUI("SERVICES MONITOR");
    }

    pointcut GUI(StockBrokerInterface obj):
        target (obj) && call (* showGUI());

    after(StockBrokerInterface obj): GUI(obj) {
        try {
            this.updateScreen("Server initialized");
        } catch (Exception ex){}
    }

    pointcut ClientBuy(int id, String stock, int shares):
        target (StockClient) && args(id, stock, shares) &&
        call (* buy(int, String, int));

    pointcut BrokerBuy(int id, String stock, int shares):
```
target (Broker) & args(id, stock, shares) &
call (* updateClientResults(int, String, int));

before(int id, String stock, int shares): ClientBuy(id, stock, shares) {
  try {
    this.updateScreen("Client "+id+" is buying "+ stock+" shares");
  } catch (Exception ex){}
}

after(int id, String stock, int shares): BrokerBuy(id, stock, shares) {
  try {
    this.updateScreen(" "+shares+" "+ stock+" shares sold to Client "+id);
  } catch (Exception ex){}
}

public void run() {
  while (true) {
    Thread.sleep(someTime);
  }
}

public void createGUI(String str) {
  myFrame = new Frame(str);
  myFrame.setLayout(new FlowLayout());

  Button b = new Button("What's going on?");
  myFrame.add(b);

  result = new TextArea(20, 50);
  myFrame.add(result);

  myFrame.setSize(375, 375);
  myFrame.setEditable(false);
  myFrame.show();
}

public void updateScreen(String str) {
  result.append(str+"n");
}

// end of aspect

3.2. CLIENT-SPECIFIC ASPECTS
These are needed to make application development transparent to the middleware. To accomplish this objective most middleware specific code was collected into an aspect and weaved into clients using aspect introductions. A typical client application will make use of the following standard methods in Jini enhanced software:

1. A fixed constructor or a constructor with a fixed set of Jini-related statements.
2. A fixed set of functions with set internal code to handle:
   - capturing discovery events and services,
   - interacting with the lookup service,
   - searching for services
   - maintaining service leases
These standard methods were encapsulated into aspects. For the constructor, three options are available:

1. Introduce the constructor using an aspect. This option has the drawback that clients will not have access to the constructor code.

2. Use a separate method which the constructor will invoke. This is a better option, requiring the user to only include a method call in the constructor of any client.

3. Interrupt the execution of the constructor in the client program using the call or execution pointcut designators, and execute the required code using a before advice.

For our implementation we chose the second method. The method constructorProxy() in our application has the standard code for a client constructor. The complete code for the Jini client aspect is shown below.

```java
public aspect JiniClientAspect {

    public void StockClient.constructorProxy(Class[] types)
        throws RemoteException, IOException {
        this.template = new ServiceTemplate(null, types, null);
        // Set a security manager
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
        // Only search the public group
        this.disco = new LookupDiscovery(new String[] { "simmonds" });
        // Install a listener
        this.disco.addDiscoveryListener(new Listener());
        this.eventCatcher = new MyEventListener();
    }

    // Called when an event is received.
    public void StockClient.ClientNotify(RemoteEvent ev)
        throws RemoteException, UnknownEventException {
        try {
            System.out.println("Got an event from: "+ ev.getSource());
            if (ev instanceof ServiceEvent) {
                ServiceEvent sev = (ServiceEvent) ev;
                ServiceItem item = sev.getServiceItem();
                this.service = item.service;
                this.serviceFound = true;
                System.out.println("Got a matching service.");
            } else {
                System.out.println("Not a service event, ignoring");
            }
        } catch (Exception ex) {
        }
    }

    public void StockClient.ClientDiscovered(DiscoveryEvent ev) {
        System.out.println("Lookup service discovered.");
        try {
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0; i<newregs.length; i++)
```
public void StockClient.ClientDiscarded(DiscoveryEvent ev)
{}

// Once we've found a new lookup service, search for proxies
// that implement the service

public Object StockClient.lookForService(ServiceRegistrar lusvc)
throws RemoteException, InterruptedException {
    try {
        this.service = lusvc.lookup(template); 
    } catch (RemoteException ex) {
        System.err.println("Error doing lookup: " + ex.getMessage());
        return null;
    }

    if (this.service != null) {
        this.serviceFound=true;
        System.out.println("Got a matching service..##" + this.serviceFound);
    } else {
        System.err.println("No matching service... registering...");
        try {
            this.registerForEvents(lusvc);
        } catch (RemoteException ex) {
            System.err.println("Can't solicit events: " + ex.getMessage());
            // Discard it, so we can find it again
            this.disco.discard(lusvc);
        } finally
            return null;
    } // end of else

    return null;
} // end of method lookForService...

public void StockClient.registerForEvents(ServiceRegistrar lu)
throws RemoteException {
    EventRegistration evreg;
    evreg = lu.notify(template,
            ServiceRegistrar.TRANSITION_NOMATCH_MATCH,
            this.eventCatcher, null, this.LEASE_TIME);

    this.eventRegs.addElement(evreg);
    this.leaseThread.interrupt();
}
if (l.getExpiration() - (20 * 1000) < soonestExpiration) {
    soonestExpiration = l.getExpiration() - (20 * 1000);
}
long now = System.currentTimeMillis();
if (now >= soonestExpiration)
    return 0;
else
    return soonestExpiration - now;

// Do the lease renewal work.
public synchronized void StockClient.renewLeases() {
    long now = System.currentTimeMillis();
    Vector deadLeases = new Vector();
    for (int i=0 , size=this.eventRegs.size() ; i<size ; i++) {
        Lease l = ((EventRegistration) this.eventRegs.elementAt(i)).getLease();
        if (now <= l.getExpiration() &&
            now >= l.getExpiration() - (20 * 1000)) {
            try {
                System.out.println("Renewing lease.");
                l.renew(this.LEASE_TIME);
            } catch (Exception ex) {
                System.err.println("Couldn’t renew lease: " +
                                   ex.getMessage());
                deadLeases.addElement(eventRegs.elementAt(i));
            }
        }
    }
    // clean up after any leases that died
    for (int i=0, size=deadLeases.size() ; i<size ; i++) {
        this.eventRegs.removeElement(deadLeases.elementAt(i));
    }
} //end of aspect

In our application, the size of the original client code was significantly reduced since it now contained only application specific functional code.

3.3. SERVER-SPECIFIC ASPECTS
These are similar in scope, content and purpose to those of clients. They are not being shown here for lack of space.

3.4. DISCUSSION
As with any new technology, there are challenges in the learning and use of AOP with AspectJ and applying its abstractions to programming using Jini. Among the challenges faced were the following:

1. The material available on AspectJ from the official web site\textsuperscript{9,10} is preliminary and does not provide easy answers to many of the questions an average programmer would ask of an aspect oriented programming language. This results in much experimentation, which can consume a significant portion of the allocated project time.
2. The AspectJ compiler is still evolving. One drawback is that even single errors resulted in multiple error messages.

3. The rationale for the use of inheritance versus aspect introductions is not clear at this point. In our research project, since our client classes were not extending any class, inheritance was a viable option for which we possess years of collective experience, and therefore semantic clarity. A major benefit of using aspect introduction is that it makes inheritance a viable option in the future, for languages which allow only single inheritance. However, for languages which allow multiple inheritance, inheritance may be a better option until our understanding of AOP becomes more mature.

4. AspectJ does not allow for the introduction of inner classes. Therefore, applications that require this kind of feature are left with no alternative but programmer calisthenics. The problem is that such maneuverings violate the very principles we are trying to implement. For example, our objective was to make middleware code programming transparent to programmers. In order to achieve this we attempted to encapsulate into an aspect (e.g. the JiniClientAspect for client code) code which all Jini client applications would use. Our problem was that because classes cannot be introduced, the client code required knowledge of some of the methods that were going to be introduced. For example, in the following inner class methods ClientDiscovered() and ClientDiscarded() are both introductions by the JiniClientAspect aspect.

```java
// An inner class to implement DiscoveryListener
class Listener implements DiscoveryListener {
    public void discovered(DiscoveryEvent ev) {
        ClientDiscovered(ev);
    }

    public void discarded(DiscoveryEvent ev) {
        ClientDiscarded(ev);
    }
}
```

This is not to suggest that applications should always be transparent to introduced methods or data. The point is that this information should be available only on demand and not because there are no alternatives.

5. The converse challenge is that sometimes an aspect needs to have information about the details of an application. For example, in our case the client aspect needed to have information about the name of variables to be used by a client. One simple solution is to have the aspect introduce all data items into the client class. Of course, now the client must know about these variables in order to use them. Our general consensus is that transparency is a spectrum or continuum of internal visibility. Total transparency means that no information is shared, while no transparency means that all information is shared. In the final analysis, total transparency may be an unrealistic utopian dream.

4. CONCLUSIONS AND FUTURE WORK

Aspect oriented programming is an exciting software development paradigm with many potential benefits but also many challenges. Our paper has sought to explore a single idea in AOP: how can AOP benefit software development using the Jini middleware. We can achieve middleware transparency, reduce the middleware learning curve for developers, reduce software development time and increase reusability for both middleware and general purpose aspects. The initial results are favorable. However, much more work needs to be done before a consensus can be reached. Future research must not only answer questions as to the benefits of AOP, but also address the issue of the cost of these benefits. In particular, the challenges and problems which AOP will introduce into the software development life cycle needs to be addressed with some urgency. We need to evaluate the utility of inheritance as against aspects and aspect weaving.
REFERENCES