Aspect Oriented Distributed System using AspectJ

Inderjit Singh Dhanoa, Gurdas Singh, Mohanjeet Singh
BIS College of Engineering & Tech., Moga
RIMT, Mandi Gobindgarh

Abstract
This paper discusses the impact of Aspect oriented programming (AOP) on Java-based communication middleware (Java RMI) system and need to introduce aspects in Java RMI systems. There are certain properties like tracing, exception handling, distribution and profiling in Java RMI system which we cannot encapsulate properly using object oriented programming and lead to the problem of code tangling and code scattering. Therefore it is difficult to modularize them in separate functional modules. These properties are known as cross cutting concerns, which can be encapsulated into Aspect using Aspect Oriented Programming. We have used AspectJ: Java based language to create aspects in Eclipse supported framework. We have shown the impact of this new programming method through a distributed auction system using Java RMI in Eclipse Platform’s Aspect visualizer.

Key Words: Aspect Oriented programming, AspectJ, Crosscutting Concerns, Eclipse, Java RMI.

1. Introduction

Most Distributed Systems are designed using Java RMI [2] middleware Technology and Object Oriented Programming concepts which are sufficient to encapsulate the functional concerns of application as like setCurrentBid, setMinBid, getCurrentBid, setProduct etc in our distributed Auction System. Despite the many advantages of OOP, it has proven to be encumbered with many hindrances and shortcomings such as repetitive code, disorderly code, low productivity, low reusability, and difficulty to design and maintain the system. The Object Oriented Programming approach cannot efficiently solve the problem that arise from the crosscutting concerns such as the distribution, tracing, exception handling, logging etc, it forces the implementation of those design decisions to be scattered throughout the code, resulting in tangled code that is excessively difficult to develop and maintain. These flaws in Object Oriented Programming are removed with the use of Aspect Oriented Programming concept to encapsulate the non-functional or crosscutting concerns[5] like distribution in Java RMI systems. Our experience with this system have been proved that use of AspectJ language helps to modularize the crosscutting concerns and improved the productivity, reusability, maintainability and code independence in our Java RMI system. Our results are shown in Aspect Visualizer of Eclipse Platform.

2. Related Work

There are a number of publications reporting the possible applications of aspect oriented programming. Zhang et al [7] verified the advantages of Aspect Oriented Programming paradigm through a small management information system. M. Nishizawa, et al [8] presented extension to AspectJ for distributed computing; the language construct that we call remote pointcut enables developers to write a simple aspect to modularize crosscutting concerns distributed on multiple hosts. A.Stevenson et al [9] described an aspect oriented approach to construct smart proxies in Java RMI. It does not change existing RMI code, and functionality in the smart proxy can be added and removed at runtime. Soares et al [13] reported that they could use AspectJ for improving the modularity of their program written using Java RMI. Without AspectJ, the program must include the code following the programming conventions required by the Java RMI. AspectJ allows separation of that code from the rest into a distribution aspect. Avadhesh kumar et al [15] found that average change impact in aspect oriented system is less than the average change impact in object oriented system, that means aspect oriented systems are easily maintainable.

In this paper, I present experimental study of aspect oriented programming in Java RMI System and try to show the affect of ‘Aspects’ on application through Eclipse’s Aspect Visualizer.

3. Java RMI Architecture

Java RMI is based on the distinction between object interface and implementation. It relies on the fact that a client cannot distinguish between objects implementing a remote interface if their behavior is identical. The architecture of Java RMI consists of the three layers in Figure 1 [14]. The first layer provides a proxy object on the client and a skeleton object at the server. In current versions of Java, there is one skeleton object for the server. The proxy object is a local object on the client JVM that implements the same remote interface as the object implementation on the server. The proxy translates method invocations to remote method invocations to the server. Part of this translation uses the remote object reference for the remote object held in the remote reference layer. The Transport Layer handles client/server communication.

The proxy object may be statically-generated by the rmic stub compiler or may be a dynamic proxy generated at runtime by the JVM. The rmic compiler starts with a class
that implements a remote interface (one derived from java.rmi.Remote). From this, rmic generates a proxy class that implements the same remote interface. The name of this proxy class is the name of the implementation with “Stub” appended. For each method in the remote interface, rmic generates code that uses the remote object reference to invoke the same method on the object implementation at the server. At runtime, when the client imports the remote object using the RMI registry, it loads this proxy class using its name. If the proxy class is successfully loaded, a proxy object is created. If not, then the second method of proxy generation is used. The second method of generating a proxy object is using the dynamic proxy mechanism introduced in Java 1.3 [12]. Given a list of interfaces, the JVM can create a proxy implementing them at runtime. Method calls on the proxy are delegated to an invocation handler object provided by the developer. In Java RMI, if the JVM cannot load the rmic-generated proxy class, the client creates a dynamic proxy using the remote interface. A RemoteObjectInvocationHandler object is created as the invocation handler, which provides identical functionality as rmic-generated stubs. Stub and skeleton on server communicate with client’s stub through Remote reference layer. Rmiregistry on server side register objects which are remotely available for clients.

4. Aspect Oriented Programming
Aspect Oriented Programming (AOP) is a program development methodology proposed by Gregor Kiczales in "Aspect-Oriented Programming"[1], published in 1997. In AOP, the requirements (requests) of the program are termed 'concerns'. Concerns are divided into core concerns and crosscutting concerns. An example that is used most frequently to explain core and cross-cutting concerns is the Distributed Auction system. In a system, core concerns are the main functions of the Auction System, which are to set the product for auction, set minimum bid, set current bid etc. However, other features required by a distributed system, such as logging, distribution, profiling and tracing are cross-cutting concerns. Although object oriented programming is currently the most widely used methodology for dealing with core concerns, it comes up short in processing crosscutting concerns. This becomes more so for complex applications.

AOP is a new methodology that enables separation of crosscutting concerns and their implementation through a new module termed the 'aspect'. Figure 2 displays the weaving process of application code with aspect.

4.1 AspectJ
AspectJ, originally from Xerox PARC, but now part of the Eclipse initiative supported by IBM, is currently the most widely adopted programming language supporting AOP and was also used for our case studies which are described in following sections. AspectJ is built on top of the programming language Java [3,4]. It provides mechanisms to modularize crosscutting concerns as explained above. In AspectJ programs, Java classes are used to implement the core characteristics, and aspects (understandable as pseudo classes) are used to implement crosscutting concerns in a modular fashion. In an AspectJ application, everything revolves around join points. These are points in the control flow graph of a compiled program, where crosscutting concerns are woven in. According to AspectJ’s terminology there are two types of crosscutting:

**Static crosscutting** describes crosscutting that influences the interfaces of the involved types and does not modify the execution behavior of the system. AspectJ provides the following two mechanisms to achieve this kind of influence:

- **Introduction** introduces changes to the classes, aspects and interfaces of the system.
- **Compile-time Declaration** adds compile time warnings and error messages for the case that certain occurrences of patterns are captured.

**Dynamic crosscutting** describes crosscutting that influence the execution behavior of an application. AspectJ provides the following two languages constructs to achieve this kind of influence:

- **Pointcut** is a constructor that selects join points and collects the context at those points based on different conditions. This construct is an aggregation of execution join points and object join points.
- **Advice** declares a method which will be executed before, after or around join points in execution flow of application.
picked up by a pointcut whenever a match is occurred with signature of defined join points. With these additional constructs, there are two execution object pointcut designators: this() and target() as defined in [6]. The Java developer can add new functionality in the system without changing any code in the core modules (classes). AspectJ retains all the benefits of Java and is therefore platform-independent. As far as compatibility is concerned it is important to note that

- Every syntactically correct Java program is also a syntactically correct AspectJ program, and
- Every successfully compiled AspectJ program can be executed on a standard Java Virtual Machine.

After these preliminary explanations we are now prepared to consider the Java RMI application in AspectJ language in following sections.

4.2 Distribution Aspect

The server-side distribution aspect is responsible for making the auc instance of Auctioneer class to remotely available. It also ensures that the methods of the auc have serializable parameter and return types, since this is required by RMI. AuctionI pointcut is defined to capture the join points of constructor execution and object type execution using this() designator. Context of specified join point is collected as parameters using args() pointcut in aspect. Distribution aspect code is given below:

```java
public aspect DistributionAspect {
    declare parents:Auctioneer implements AuctionInterface;
    pointcut AuctionI(Auctioneer auc,String pr,String mb):execution(Auctioneer.new(..) throws RemoteException) && this(auc) && args(pr,mb);
    after(Auctioneer auc,String pr,String mb) throws RemoteException:AuctionI(auc,pr,mb){
        try {
            UnicastRemoteObject.exportObject(auc);
            name = "//localhost:1099"/"+pr;
            Naming.rebind(name,auc);
        } catch (java.net.MalformedURLException me) {
            System.out.println(me.toString());
        }
    }
}
```

The server side aspect has to define a remote interface that has all Auctioneer methods signatures adding a specific RMI API exception (java.rmi.RemoteException).

```java
interface AuctionInterface extends java.rmi.Remote {
    public void setCurrentBid(String bid) throws RemoteException;
    public String getCurrentBid() throws RemoteException;
}
```

We used the AspectJ's introduction mechanism that can modify the static structure of program to implement AuctionInterface interface, as in the following piece of code:

```java
declare parents:Auctioneer implements AuctionInterface;
```

distribution aspect on server side designed to encapsulate the crosscutting concerns. Later we consider the tracing, profiling and error handling aspects. Distribution aspect code is weaved with Java byte code dynamically without changing the code of base application. Advice code of distribution aspect applied in execution flow join points when a match occurred with specified signature of distribution aspect pointcut.

4.3 Tracing Aspect

Tracing is done to know the logical flow of the server during execution. Tracing determines the various application components that were involved in a software task. Tracing is required to troubleshoot functional and performance problems. To trace the execution flow, usually print statements are inserted in all the components of software. Aspect Oriented Programming separates the lines of code that are required for tracing the execution flow, from the rest of application. It encapsulates the tracing concern in SimpleTracing aspect as given in following code snippet:

```java
public aspect SimpleTracing {
    // pointcut trace:call(* *(..)) matches all method /(calls
    pointcut trace(): call (* *(..)) && ! Within (SimpleTracing);
    before (): trace() {
        System.out.print("Entering"+thisJoinPoint.getSignature());
    }
}
```

ThisJoinPoint object of type JoinPoint contains the dynamic information of the advised join point. It gives access to the target object, the execution object, and the method arguments. We have used it to know the dynamic information of server execution. You can see the result of SimpleTracing in the command window figure 3.

![Figure 3: Result of Tracing & Profiling Aspect](image-url)
4.4 Profiling Aspect

Profiling in a Java RMI context cannot be achieved with standard profiling software incorporated into the usual operating systems for ordinary PCs. The distributed nature of the runtime application requires more refined approaches to performance profiling. In such scenarios it is necessary to deal with computation and communication separately, and sum up the components to calculate the total CPU time across the entire cluster. Profiling aspect as given in following snippet has been implemented in Auctioneer server and results are shown in Figure 3. Whereby execution time in milliseconds is the observable on which the profiling is solely focused.

public aspect profileAspect {
    private long d, endTime, startTime;
    pointcut measure(Auctioneer auc, String pr, String mb): execution(Auctioneer.new(..)) && this(auc) && args(pr, mb);
    before(Auctioneer auc, String pr, String mb): measure(auc, pr, mb)
    {
        d = 0;
        startTime = System.currentTimeMillis();
    }
    after (Auctioneer auc, String pr, String mb): measure(auc, pr, mb)
    {
        endTime = System.currentTimeMillis();
        d = endTime - startTime;
        System.out.println("Execution Time is:" + d);
    }
}

In this pointcut measure captured constructor execution join points as in distribution aspect, but advice used both before () and after () constructs of aspectJ language. before () advice body measured start time and after () advice measured stop time. Then total Execution time calculated in milliseconds.

4.5 Exception Handling Aspect

An exception is a behavior of the system indicating that the operation in process cannot be successfully completed, but from which other parts of the system can try to recover or chose to ignore. The code for exception handling often tangles the main code. In their study, Martin Lippert et al [10] found that Aspect Oriented Programming supports implementations that drastically reduce the portion of the code related to exception handling. They found that AspectJ provides better support for different configurations of exceptional behaviors, more tolerance for changes in the specifications of exceptional behaviors, better support for incremental development, better reuse and automatic enforcement of contracts in applications. Fernando C Filho et al [11] specifies the benefits and liabilities of Aspects in error handling.

public aspect Handler {
    pointcut exceptionHandler(Exception e):
    handler(Exception+) && args(e):
    after throwing (Exception e): exceptionHandler(e)
    {System.out.println("Exception " + e + " caught while Executing " + thisJoinPoint());
    }}

Above code snippet displays an aspect to handle exceptions. This aspect has an after throwing () advice that prints a message after an exception is thrown in the methods of the Auctioneer class.

4.6 Eclipse’s Aspect Visualiser

Aspect Visualiser is an extensible plugin that can be used to visualize anything that can be represented by bars and stripes. It began as the Aspect Visualiser, which was a part of the popular AspectJ Development Tools (AJDT) plug-in. It was originally created to visualize how aspects were affecting classes in a project. As in Figure 4 we have shown the member view of distribution, tracing, profiling and exception handling aspects with class Auctioneer in java RMI system. Here bars represent classes and aspects in server program and black colored stripes represent advised join points in the execution flow of program, which were matched with defined pointcuts in various aspects.
5. Conclusion

This paper presented a case study illustrating how aspect oriented software development (AOSD) is useful to resolve the tangled concerns specifically in Java RMI Systems. We demonstrate the use of AspectJ in distributed environment, which is Java based tool to encapsulate the crosscutting concerns like distribution, tracing, profiling and exception handling. We used Eclipse platform’s Aspect visualizer to analyze the impact of aspect advice on program execution flow. Aspects helped us to achieve high cohesion, low coupling that the software engineering require and to enhance the readability of the system, and made it easier to maintain. Constructs of AspectJ helped us to use the context of server in aspects at runtime and in future with this we can share server’s knowledge dynamically with clients.

6. Future Work

Aspect oriented programming has tremendous potential for building the distributed applications in the future. We are planning to investigate the finer details of aspect mining techniques with Eclipse’s plug-in FINT tool and security issues in distributed environment as aspects. We are also planning to implement aspect oriented smart proxy in Java RMI.

7. References


