Comparative Analysis of Java and AspectJ on the Basis of Various Metrics

Inderjit Singh Dhanoa
IT Department
BIS College of Engineering & Technology
Moga(Punjab),India
inderp10@yahoo.co.in

Er.Dalwinder Singh Salaria
CSE/IT Department
Lovely Professional University
Phagwara(Punjab),India
dalwinder.singh@lpu.co.in

Dr.H.S Johal
CSE/IT Department
Lovely Professional University
Phagwara(Punjab),India
johalhs2008@gmail.com

Abstract -- This paper compares aspect oriented approach using AspectJ with object oriented approach using Java programming in distributed environment and discusses the need to introduce aspects in Java RMI systems. These two approaches compared empirically using RMI auction System in Eclipse’s framework in terms of various metrics. We developed RMI Auction System with AspectJ and Java languages in Eclipse’s framework independently. 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. In this, AspectJ language is used to encapsulate distributed cross cutting concern of RMI auction system in Eclipse supported framework. We have shown the comparison of AspectJ and Java through a RMI auction system in Eclipse Platform’s Aspect visualizer and Metrics 1.3.6 plug-in.

Keywords-- Crosscutting Concerns; Eclipse; Java RMI; UMRT

I. INTRODUCTION

Most Distributed Systems are designed using Java RMI (Remote Method Invocation) [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 Object Oriented Programming, 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 [3]. 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 [4] like distribution in Java RMI systems. Our experience with this system has been proved that use of AspectJ language helps to modularize the crosscutting concerns and improved the UMRT (Understandability, Maintainability, Reusability and Testability) attributes of RMI auction system. Our results are shown after execution of RMI auction system using Java and AspectJ code in Metrics 1.3.6 plug-in and Aspect Visualizer of Eclipse Platform.

II. RELATED WORK

To our knowledge, there is no previous metrics based empirical work that compares AspectJ and Java implementation in distributed environment using Eclipse’s plug-in. There are number of publications reporting the possible applications of aspect oriented programming but no one discussed the experimental results to enhance UMRT attributes of distributed system. Fazal-c-Amin et al [5] reviewed the research in Aspect Oriented field and highlight the application domains along with results, opportunities and challenges. The results revealed that the major benefits of using aspect orientation as the product line technology are enhanced modularity (21%) and better variability management (32%). P. Greenwood et al [6] Compares the design stability Aspect Oriented implementation against an Object Oriented implementation (using Java) of Health-Watcher, a system designed to monitor public-health-related complaints and notifications in Recife, Brazil. The study applied numerous common maintenance scenarios to both the Aspect Oriented and Object Oriented versions of this benchmark application. The analysis revealed that concerns modularized using Aspect Oriented techniques showed superior design stability, and modifications tended to be confined to the target modules. 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 [10] 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 [11] found that average change impact in aspect oriented system is less than the average change impacts in object oriented system, that means aspect oriented systems are easily maintainable. Katharina Mehner [12] gave a short overview on the necessary steps for validating metrics that are to be used in an evaluation process of Aspect oriented code.
In this paper, we present experimental study to compare AspectJ and Java through RMI auction system and try to show the affect of ‘Aspects’ on system’s attributes through Eclipse’s Aspect Visualizer and Metrics 1.3.6 plug-in.

III. 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 Fig. 1 [13]. 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 [14]. 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.

IV. 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’. Fig. 2 displays the weaving process of application code with aspect.

A. 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 study which is described in following sections. AspectJ is built on top of the programming language Java [15,16]. 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 language 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[17]. 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 RMI system in AspectJ language in following sections.

**B. 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
{
    public String name;
    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
public 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** aspect on server side of RMI system designed to encapsulate the distribution crosscutting concern. Later we consider the error handling aspect code. Distribution aspect code is weaved with java byte code dynamically without changing the code of base application on server side. Advice code of distribution aspect applied in execution flow join points when a match occurred with specified signature of distribution aspect’s pointcut.

**C. 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 [18] 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 [19] specifies the benefits and liabilities of Aspects in error handling.

```java
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.
D. Eclipse’s Aspect Visualiser

Aspect Visualiser is an extensible plug-in 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 Fig. 3 we have shown the member view of distribution and exception handling aspects with class Auctioneer in AspectJ RMI system. Here bars represent classes and aspects in server program and yellowish colored stripes represent advised join points in the execution flow of program, which were matched with defined pointcuts in various aspects.

V. METRICS

The Chidamber and Kemerer Metrics (C&K) is a suite of metrics designed for evaluating object oriented designs and is detailed in [20]. Rosenberg et al [21] have proposed a set of guidelines as to how to interpret the metrics in the C&K suite. Table I below summarizes the objective for the values of the metrics.

<table>
<thead>
<tr>
<th>METRIC</th>
<th>OBJECTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted Methods per Class</td>
<td>Low</td>
</tr>
<tr>
<td>Coupling Between Objects</td>
<td>Low</td>
</tr>
<tr>
<td>Response For a Class</td>
<td>Low</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods</td>
<td>Low</td>
</tr>
<tr>
<td>Depth of Inheritance Tree</td>
<td>trade-off</td>
</tr>
<tr>
<td>Number of Children</td>
<td>trade-off</td>
</tr>
</tbody>
</table>

However, as indicated in the last two metrics, there is a trade-off with some of the metrics. A high DIT will increase maintainability complexity but also shows increased reuse. Similarly a high NOC will increase the testing effort but will also increase the extent of reuse efficiency. Thus developers must be aware of the relationship between the metrics as altering the size of one can impact areas such as testing, understandability, maintainability, development effort and reuse as shown by Zakaria and Dr. H. Hosny [22]. Eclipse’s Metrics 1.3.6 plug-in used to measure metric values of RMI system, developed in Java and AspectJ given in the following section.

A. Metrics 1.3.6 Plug-in

To start collecting metrics for a system in Metrics 1.3.6 plug-in, right click on the project and from the popup menu select "Metrics->Enable" (or alternatively, use the properties page). This will tell Eclipse to calculate metrics every time a compile happens. Now that you’ve enabled a project, the easiest way to calculate all its metrics is to do a full rebuild of that project. The metrics view will indicate the progress of the metrics calculations as they are being performed in the background. When it’s all done, the metrics view will look something like shown in Fig. 4 & Fig. 5.
VI. RESULTS

This section details the values gathered for each of the C&K metrics using Metrics 1.3.6 plug-in, when applied to the RMI Auction System using Distribution aspect in Java and AspectJ language.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Java</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>DIT</td>
<td>3.5</td>
<td>1.667</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CBO</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RFC</td>
<td>23</td>
<td>21</td>
</tr>
<tr>
<td>LCOM</td>
<td>0.375</td>
<td>0.25</td>
</tr>
</tbody>
</table>

This allows changing the communication code without impacting other system code. Same application server can use different middleware at the same time. AspectJ RMI system implementation also facilitates functional tests and understandability of the system. Functional tests are easier by testing the system with its local version; therefore, distribution code errors will not affect these tests. There might be some impact on performance of the distributed system because of AspectJ language constructs and context passing from Auctioneer class to aspect named DistributionAspect. At the cost of this, code quality and modularity of system improved which is beneficial for the AspectJ developer’s team. Eclipse plug-in performed a major role in the development of AspectJ Auction System, as it facilitated aspect development and binding process with Rmiregistry. The evaluation of this system was based on metrics that are used for evaluating OO designs. Previous studies of AOP have used OO design metrics to evaluate design of code and counting coupling connections between AO modules equivalent to coupling connections between OO modules. Classes and Aspects are often measured together as equivalent modules like in DIT. Aspect Oriented Programming affected various UMRT attributes of Java RMI system and this affect has been shown by various screen shots of metrics view and aspect Visualization property of Eclipse. Finally, it is evident from the analysis that the results given here are positive to modularize the Java RMI application and in favor of AspectJ as compare to Java.

VII. DISCUSSION

It is evident from this analysis that the results are in favor of AspectJ as it has been highlighted here that AOP is beneficial in terms of understandability, maintainability, reusability and testability (UMRT) attributes of application. It has been shown in the Table II and Fig. 6 that AspectJ RMI application has less value of metrics as compare to Java application except coupling between aspect and core-class, which leads to increase in UMRT attributes. Coupling value in Aspect application shows that coupling between class and aspect is more, which leads to low coupling between core-classes [22]. Low coupling means high UMRT attributes of AspectJ RMI application. All metric values met our objective as given in Table I of this paper and affected the understandability, maintainability, reusability and testability (UMRT) attributes of RMI application in AspectJ. In spite of this, AspectJ system is completely independent of the communication code and middleware to facilitate system maintenance, as communication code is not tangled with user interface code.

VIII. CONCLUSIONS

This paper presented a case study illustrating how aspect oriented software development (AOSD) is useful to resolve the tangled concern specifically in RMI System as compare to object oriented software development. We demonstrate the use of AspectJ in distributed environment as compare to Java and encapsulate the crosscutting concern distribution in RMI auction system. Comparative analysis based on metrics shows that UMRT attributes improved in AspectJ RMI system as compare its Java implementation. We used Eclipse platform’s Aspect visualizer and C&K metrics plug-in to analyze the affect of aspect advice on program execution flow. Aspects helped us to achieve high cohesion, low coupling that the software engineering requires 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.

IX. FUTURE WORK

Aspect oriented programming has tremendous potential for building the distributed applications in the future. Other crosscutting concerns like tracing, profiling, logging, security and exception handling can be modularized using AspectJ in Eclipse framework. 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.
REFERENCES


[22] Zakaria, A.A. and D.H. Hosny “Metrics for aspect-oriented software design”. In 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, July 2006.