Aspect-Oriented Multi-Client Chat Application

Duygu Ceylan, Gizem Gürcüoğlu, Sare G. Sevil
Department of Computer Engineering, Bilkent University
Ankara, Turkey 06800
{dceylan, gizem, sareg} @cs.bilkent.edu.tr

Abstract

As the demand for network applications increases and the necessity of developing more sophisticated systems arises, distributed computing such as Client-Server applications receives more attention. Due to the distributed nature of these applications, many concerns like synchronization, security, and performance issues become crucial and the ability to modularize these concerns becomes more valuable. This paper focuses on using Aspect-Oriented Software Development approaches for separating such concerns. An example case, namely a chat application, constitutes the basis of this study and several concerns specific to this application are illustrated. Considered and handled concerns are types of concerns that occur both in development and production phases of the application. Implementations using different environments are also discussed.

Keywords
aspect-oriented software architecture design, multi-client chat application, design patterns.

1. Introduction

From publishing and sharing of information through software, to the execution of complex processes within several computers, distributed systems are widely used among all kinds of applications in software development. These systems are mainly composed of multiple processes that generally work in parallel in some predefined structural design. Among these structural designs, client-server architecture models are some of the most commonly applied software architecture models where their application range varies from simple business applications to standardized internet protocols including HTTP, DNS and SMTP [3].

In client-server applications, processes are classified into client and server systems. These systems communicate over a network connection through which client systems make connections and request for operations while server systems wait for requests from clients and serve them accordingly.

Throughout the years, providers such as Microsoft, Yahoo and Google, and many open-source software developers have adapted client-server model to a popular user application: the multi-client chat. These chat applications generally appear in the form of *chatrooms* where synchronous, and sometimes asynchronous, conferencing takes place; or they can be in the form of instant messengers where users communicate by exchanging textual, visual or audio data over a connection. While in chatrooms all users are allowed to communicate with each other, in IMs only users who accept to communicate with each other are allowed to exchange messages.

In this paper we go through the design and implementation process of an instant messaging chat application program. We analyze the components of a standard chat application and the concerns that might be encountered during development and production stages of the application. In order to facilitate the handling of concerns we propose the application of aspect oriented approaches and discuss why these aspects are more convenient for usage.

The paper is organized as follows: in section II the initial object oriented design of the application is explained. Section III describes the problems that might be encountered with this design. In section IV a new design with aspects, applied in AspectJ, is presented. Section V discusses the implications of aspect usage. Section VI describes possible implementation differences with JBoss usage in implementing aspects. In section VII related work is discussed and section VIII concludes the paper.

2. Multi-Client Chat Application: Object-Oriented Design

2.1 Description of the Application

The multi-client chat application focused in this paper is a server-client chat application where multiple clients

communicate with each other by sending text messages over a connection provided by a single server. For this reason, it would be convenient to analyze this system in two main components: server and client.

Each client component found in the system represents a user who wants to communicate. A client is identified by his/her username. All users have their own friend lists which are simply lists of other clients (i.e. contacts) using the chat application. Each user is allowed to communicate only with his/her own contacts.

Each contact can be in one of two possible states, “Online” or “Offline”, depending whether they are currently connected to the server or not. When an offline contact connects to the server, his/her status changes from offline to online. Similarly, when an online contact disconnects from the server, his/her status becomes offline.

Users are allowed to have private conversations (private chats) with their online friends. During a private conversation, the contact being talked to might disconnect from the server without notifying the client. If the client continues to send messages when this happens, an automatic notification is send to the client informing him/her of the situation. Finally, a client can add a new contact as his/her friend and remove an existing friend.

The main function of the server component is to wait for connection requests from the clients and authenticate these requests by checking the usernames and passwords provided in requests. The server thus stores a list of valid username-password pairs which are used in the authentication process. In our current implementation, when a username which is not found in the list is provided by a connection request, the server accepts this request as a new client and adds the username-password pair to the existing list. In addition to authentication information, the server also keeps data related to the friend lists of the clients. For each client, a list of friends is kept as well as a list of other clients that has this client as friend. Upon confirming a connection request from a client, the server sends usernames of the friends of the client and their connection status. Finally, the server processes the requests from a client such as adding or removing a friend by updating lists accordingly and it provides necessary communication base for private conversations with friends. Use case diagrams shown in Figures 1 and 2 summarize these components and their features.

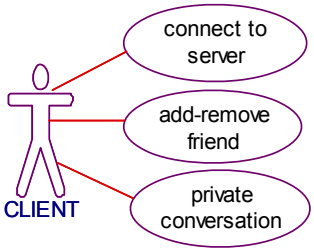

Figure 1. Use case diagram: Client

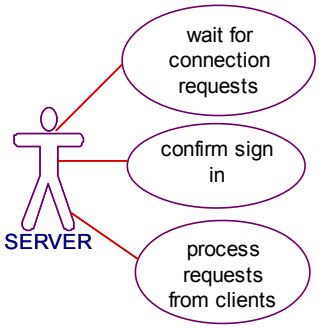

Figure 2. Use case diagram: Server

2.2 Object-Oriented Design

When a closer look is taken at the design of the multi-client chat application system, we can see that it is possible to represent the system with two main classes, corresponding to the two main components of the system, and some other helper classes.

In our design we have chosen to implement the client component as the *Client* class that has associations with user interface classes *MainWindow* and *PrivateWindow*. The *MainWindow* can be defined as the class responsible for the main user interface window of a client. This class is used in displaying the list of contacts, providing necessary interface for adding and removing contacts and for signing in and out of the chat application. Figure 3 shows an example of a main window for an online client.

The *PrivateWindow* is the class responsible for the user interface for a private conversation between two clients. An example of a private window is shown in Figure 4.

Figure 3. Main Window for signed in Client

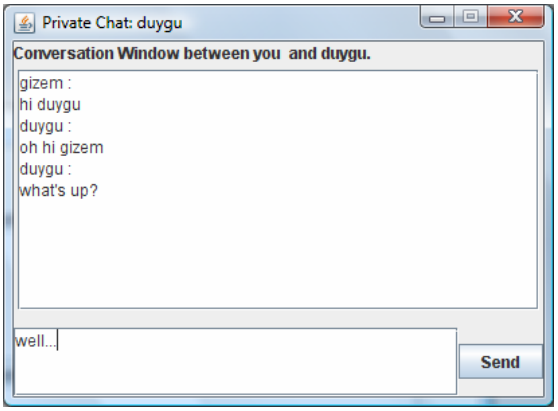

Figure 4. Private Chat Window of Client Gizem with Client Duygu

In addition to the user interface classes, *Client* also has an association with the class *ContactList* which contains a list of the friends of a given *Client*. Contacts in the *ContactList* are represented as *Contact* objects that consists of a username and connection status. *Client* objects also include a *MessageFormatter* object which is responsible for formatting the messages sent by the *Client* in the correct way. These relationships are summarized in the class diagram shown in Figure 5.

For the server component, the main class is the *Server* class. As a socket connection is received, the *Server* creates a new thread, namely a *ServerThread* object, to send and receive messages from the corresponding socket. *ServerThread* class is also responsible for parsing the messages coming from clients. In order to be able to hold information related to clients such as username, password and friend list, *Server* class creates *ServerClient* objects which are a simpler version of *Client* objects containing only relevant information and excluding everything else. Finally, the status of *ServerClient* objects is implemented through a *ConnectionStatus* interface. Currently, there are two concrete implementations of this interface which are *OnlineConnectionState* and *OfflineConnectionState* respectively. Operations for sending messages to the clients are defined in the *ConnectionStatus* interface and implemented differently according to the status of the client. The relations of the server component are summarized in the class diagram shown in Figure 6.

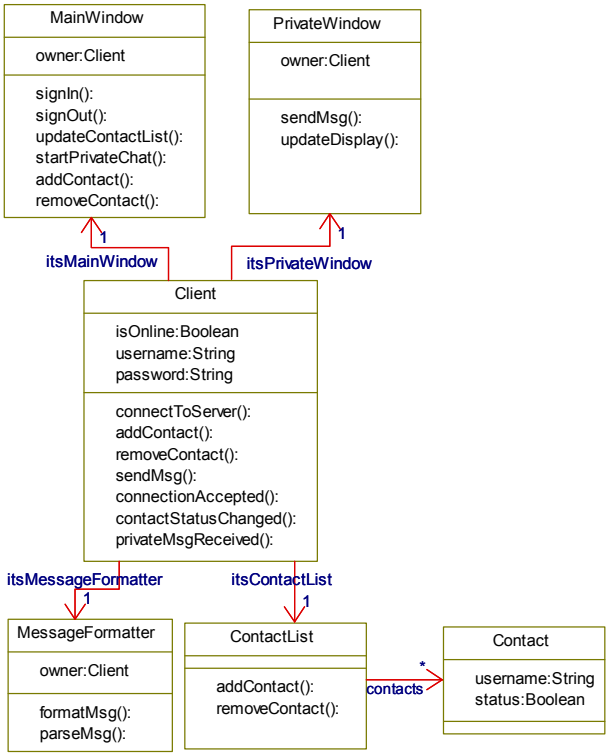

Figure 5. Class Diagram of the Client component.

2.3 Implementation of Design Patterns

In order to enhance the object-oriented design described in the previous subsection and ease some operations of the application, two design patterns have been implemented. The first design pattern that has been considered is the Observer pattern. In the system it is important to notify clients when another client that appears within their contact lists changes his/her status (online/offline). For this purpose, subscription-notification structure is used. In this structure, each client has a list of other clients that contain this client as friend. We call this list as the observer list. When client A adds client B as a friend, A is automatically attached to the observer list of B. Similarly, when client A removes client B from its friend list, it is automatically detached from the observer list.

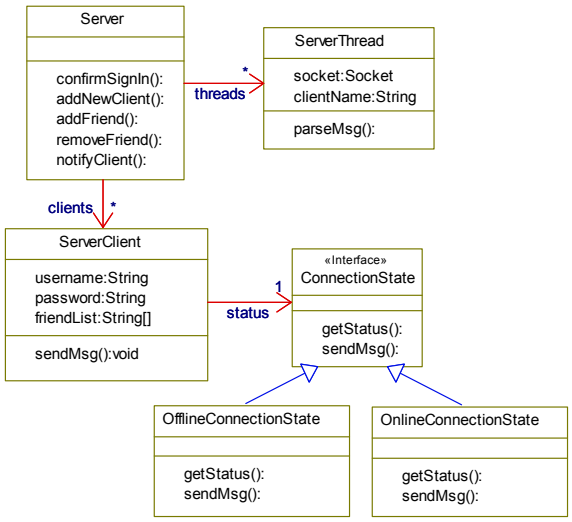

Figure 6. Class Diagram of the Server component.

Thus, when B changes status, a notification message is sent to the clients contained in the observer list of client B such as client A. Upon receiving a notification, other clients update their contact list views. This way, the need for checking whether each client contains B as a friend or not is eliminated. The structure of this pattern can be shown as in Figure 7.

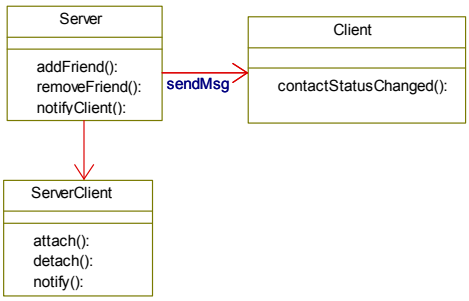

Figure 7. Class Diagram: Observer Pattern implementation.

The other design pattern included in the architecture is the State pattern. In our system clients may be in one of many states and the server needs to check the status of a client when a need of sending a message to that client arises. Currently our system supports two different states for its clients: online or offline. But in the future, other Client states may be added to the system such as Busy and/or Away where in each state, the client may perform different actions for sending messages. For example for the Away state, the client can send automatic replies to incoming private chat messages. But defining new states may result in the need of heavy code maintenance. In order to ease these operations, State pattern is used to implement the connection state of the client. A *ConnectionState* interface is defined which currently contains methods like *getStatus()* and *sendMsg()*. The concrete implementation of this interface is defined in *OnlineConnectionState* and *OfflineConnectionState* classes. For example, in the *OfflineConnectionState* class, *sendMsg()* function simply returns without sending any messages. This structure also eases the operation of adding new methods related to the connection state of the client in the future. This structure is shown in Figure 8.

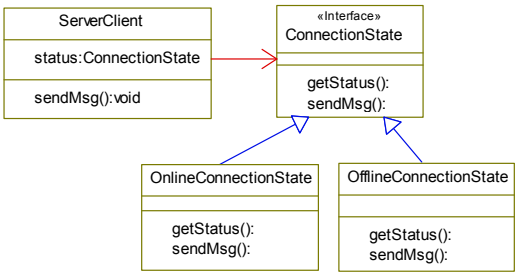

Figure 8 - Class Diagram: State Pattern implementation.

3. Aspect Oriented Programming

An important milestone for software development is to be able to define the object-oriented architecture of the system being developed. After this milestone is accomplished, concerns that seem to be scattered over several modules are identified as crosscutting concerns. Modularizing these concerns not only improves the functionality of the system but also eases the maintenance of the software. For this respect, after completing the object-oriented design of our chat application, we defined the possible crosscutting concerns that may arise during the implementation phase. We adopted the aspect-oriented approach to modularize these crosscutting concerns and developed corresponding aspects. These aspects can be categorized in two main groups which are production and development aspects respectively. Production aspects are those that add functionality to our system where as development aspects tend to ease the development phase of the system. The aspects have been implemented in AspectJ framework and code segments showing implementation details are included. Each of these aspects will be now considered in detail.

3.1 Production Aspects

Production aspects add functionality to a software application by facilitating the implementation of some features resulting in crosscutting concerns. In our system, there were some such features which we decided to implement with aspects. These features can be named as providing notifications and handling exceptions related to message sending. We have defined production aspects

for both of these features which will be described in further detail.

3.1.1 Notification Aspect:

One of the most important features of a chat application is to play several notification sounds to provide a friendlier user interface. Alerts can be used when a client signs in or receives a private message for example. However, the places where alerts are added can be increased. This means that code becomes scattered when an object oriented approach is used. Moreover each time a new notification sound is defined, both the Client code where the new notification will be added, and the class responsible for playing the sound should be changed. In order to overcome these obstacles, we have defined a notification aspect that collects all operations related to alert sounds in one place.

Currently, the notification aspect creates alerts when a client signs in and receives a private message. Therefore two pointcuts and related advices have been defined as shown in Code Fragment 1.

```
pointcut signedIn() : execution(void
      Client.connectionAccepted(..));

void around() : signedIn()
{
   new AudioWavPlayer(signedInAlertFile)
        .start();
    proceed();
}
```

Code Fragment 1: *signedIn* pointcut captures the execution of the *connectionAccepted* method of the *Client* class and the *around* advice plays an alert.

```
pointcut receivedMsg() : call(void
      Client.privateMessageReceived(..));

void around() : receivedMsg()
{
  New AudioWavPlayer(receivedMsgAlertFile)
        .start();
  proceed();
}
```

Code Fragment 2: *receivedMsg* pointcut captures the calls to the *privateMessageReceived* method of the *Client* class and the *around* advice plays an alert.

An additional *AudioWavPlayer* class has been defined to a play a wav file in a separate thread. In the notification aspect, when either of *signedIn* and *receivedMsg* pointcuts is reached, the play method of the *AudioWavPlayer* is called with the corresponding wav file name.

3.1.2 Message Send Failure Handling:

As chatting among clients is the main process of a multi-client chat application, it needs to be done properly. In our systems, private chat windows have been defined to control the chatting operation between two users. When users send messages to each other, necessary methods of the Client, ServerClient and Server classes are called and messages are sent from one user to another in pre-defined formats over a proper network connection. But what if one of the users looses this connection and is abruptly out of the chat? Obviously, due to the usage of the observer pattern, all contact lists containing the client that lost connection are updated but it is necessary for the clients that were chatting with the client during the status change to be explicitly notified.

Although this notification operation can be implemented using standard OO techniques, an implementation at the object level will increase the number of dependencies among classes and thus will result in scattered concerns. Also, at maintenance phases it would be possible to change this notification structure by reacting each time a message could not be sent due to networking problems. So this problem can be seen as a crosscutting concern and thus we have used aspects to enhance our application.

To do this, we defined a point cut that catches all calls made to the *sendPrivateMsg* method of the server class. When this pointcut is captured, the corresponding advice sends a warning message to the client whose message could not been sent. This scenario is implemented in Code Fragment 3.

```
  pointcut msgSendFailedNotification(Server s,
String name, String receiver, String msg) :
     call(* Server.sendPrivateMsg(..)) &&
     target(s) &&
     args(uname, receiver, msg);

  after(Server s, String uname, String
receiver, String msg) throwing():
  msgSendFailedNotification(s, username,
                          receiver, msg)
  {
    System.out.println("exception aspect");
    DataOutputStream dout = s.getDout(uname);
    try
    {
    dout.writeUTF(header + receiver + separator
                + notification);
    }
    catch(IOException ex)
    {}
  }
```

Code Fragment 3: *msgSendFailedNotification* pointcut captures the *sendPrivateMsg* method call in the *Server* class. The advice sends a warning message to the client when the operation returns throws an exception.

Once we have identified calls to our target method, we need to identify the times when this method throws an exception for not being able to send the message

properly through the network. This is done by using the **after throwing** aspect. Using the **after throwing** aspect we can perform necessary actions right after an exception has been thrown by the server. Of course necessary arguments and target objects need to be identified in the aspect in order to force the system to send an automated message to the client sending the message.

3.2 Development Aspects

Developments aspects have the characteristics of facilitating debugging, testing, defining enforcement policies, and performance tuning work. In our project, we had similar needs like profiling system requirements in terms of increase in the number of clients connected to the server and enforcing policies during the implementation phase. For each of these needs, we have defined two corresponding development aspects which are described below.

3.2.1 Profiling System Requirements:

In server-client applications, one of the most crucial issues is how the server behaves when the number of clients connected to the server is above a certain number. Although, we could test our system only with a limited number of clients, in the future it could be used in more sophisticated environments. In such a case, we thought it would be beneficial to have some statistical information such as when a client logs in to the system, how long s/he remains logged in, and when s/he logs out. This information can be used to determine the times of a day when the number of clients increase abruptly so that special care is taken if needed. Moreover, the type of information logged can be extended as new features are added to the system. For example, if a file transferring feature is added, the size of the files being transferred can be logged to determine how the server behaves if a large file is being transferred.

The requirements explained above however cannot be included among the functionalities of the system. They are rather used for development purposes. In addition, these requirements are scattered among the code. To begin with, in order to log information in a file, a file must be created and opened when the server starts and closed when the server stops. Secondly, code must be added to places where the information to be collected is found. For example, to collect information about when a client logs in and out of the system, code must be added to both log in and log out functions. Moreover, if the type of information collected is varied in the future, related code must be added to corresponding places as well. Finally, the option for turning collecting statistical information on and off should always exist. All these issues make the concern of profiling system requirements a good candidate to be implemented with aspects. When an aspect oriented implementation is chosen, the code will not be scattered but collected in the aspect body. Additionally, it will be easy to add and remove this concern to the system by only adding and removing the aspect itself.

As a result, a profiling aspect is added to the system. This aspect creates a log file each time the server is run. In order to create the file appropriately, the aspect defines pointcuts to catch when the server is started and stopped. Then, advices are defined corresponding to these pointcuts to open and close a file. The following code fragments show the described pointcuts and advices:

```
pointcut startServer() : execution(*
                         Server.listen(..));

before() : startServer()
{
  try
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
SimpleDateFormat("yyyyMMdd_hhmm");
    File f = new File("bin\\log" +
sdf.format(cal.getTime()) + ".txt");
    wr = new BufferedWriter(new
FileWriter(f));
    signInTimes = new
Hashtable<String,String>();
  }
  catch(IOException ex)
  {}
}
```

Code Fragment 4: *startServer* pointcut captures the execution of the *listen* method of the *Server* class and the *before* advice creates and opens a log file.

```
pointcut stopServer() : execution(*
                    Server.stopServer(..));

after(): stopServer()
{
  try
  {
    wr.close();
  }
  catch(IOException ex)
  {}
}
```

Code Fragment 5: *stopServer* pointcut captures the execution of the *stopServer* method of the *Server* class and the *after* advice closes the log file.

Currently, the system collects information about the log in and out times of clients. Therefore, two additional pointcut-advice pairs are defined which correspond to places where a client signs in and out. These operations are implemented by the Code Fragments 6 and 7.

```
  pointcut signIn(String username, Socket
socket) :
   args(username, socket) && execution(*
   Server.clientConnected(..));

  after(String username, Socket socket)
returning() : signIn(username, socket)
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
           SimpleDateFormat("hh:mm");
  signInTimes.put(username,sdf.format(
                        cal.getTime()));
  }
```

Code Fragment 6: *signIn* pointcut catches the *clientConnected* method execution in and the advice logs the current time as the sign in time of the client.

```
  pointcut signOut(String username, Socket
socket) :
   args(username, socket) &&
  execution(* Server.removeConnection(..));

  after(String username, Socket socket)
returning() : signOut(username, socket)
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
        SimpleDateFormat("hh:mm");
    try
    {
      wr.write(username + "\t\tsigned in at " +
        signInTimes.get(username) + ", signed
        out at " + sdf.format(cal.getTime()) +
        "\n");
    }
    catch(IOException ex)
    {}
  }
```

Code Fragment 7: *signOut* pointcut captures the execution of the *removeConnection* method and the *after* advice writes the previously logged time and the current sign out time to the log file.

The aspect works as follows. After the startServer pointcut is reached a new file whose name includes the current date and time is created. Each time signIn pointcut is reached, the log in time of the newly connected client is stored in a buffer. When the signOut pointcut is reached, the buffered log in time of the client and the log out time are written to the file. Finally, when the server is closed, the file is also closed. Figure 9 shows a view of a sample log file created.

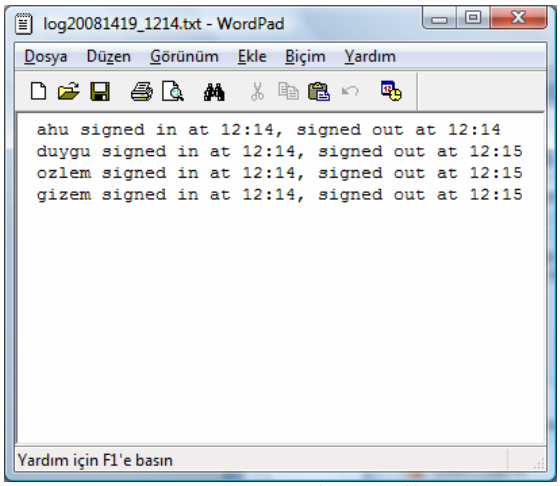

Figure 9. Sample Log File

3.2.2 Access Controlling for Client Class Aspect:

In our implementation of the multi-client chat application our Client class was a very extensive class that had some attributes and methods that by default were meant to be used by classes other than the Server class. But, as explained in the previous section, the Server class is required to store some of the information provided by the Client class.

In order to handle this access problem we had, we decided to add a separate class, *ServerClient*, that only stored Server related attributes and methods. But for a developer, two classes representing the same component might be confusing and it is possible for a developer to confuse these two classes in the development stage. Thus the access to the Client class from the Server class needs to be restricted.

As object oriented approaches have no concepts to handle this issue, we have used aspects. Due to the level of abstraction provided by aspects we are able to catch any illegal access to the Client class made by the Server at compile time and declare it as an error. In order to do this, we have used the *declare error* aspect as shown in Code Fragment 8.

```
declare error : call(Client.new(..))&&
  within(Server)
: "Client cannot be created in Server.";
```

Code Fragment 8: Calls of the *Client* constructor by a *Server* object are captured and considered as an error.

This way, whenever a Client object or any of its static methods is trying to be used by the Server class, a compile time error is produced.

4. Alternative Implementations: Jboss

In our current implementation, we have used AspectJ, one of the most popular frameworks of AOP, for applying aspect-oriented structures to our design. However we could also use some other popular framework like JBoss. However, due to the differences between the structures of AspectJ and JBoss, implementation using JBoss would be quite different.

First of all, the pointcut, and aspect declarations in the two frameworks differ from each other. JBoss supports pointcuts defined by XML or Java annotations where as AspectJ also supports language based pointcuts. Secondly, in order to define an aspect using JBoss, one has to encapsulate the aspect in its own Java class that implements the *Interceptor* interface of the JBoss API. All methods and constructors intercepted this way are turned into a generic *invoke* call. As the interceptors are attached to pointcuts, the methods intercepted are invoked at appropriate places. As an illustration, if we had used JBoss to define the pointcut and advice pair listed in Code Fragment 2, the pointcut definition and the corresponding interceptor would be as in Code Fragment 9.

Furthermore, compiling and deploying aspects in JBoss is more complicated than AspectJ. The compilation process includes compile and run steps separately and the deployment process involves adjustments related to XML files. However, AspectJ's compiler *ajc* does not require a second pass [4] and it has an easy to use plugin with Eclipse IDE tool that we have used [5].

```
public class AlertingInterceptor implements
Interceptor
  {
   public String getName() { return
       AlertingInterceptor; }
   public InvocationResponse invoke
       (Invocation invocation)
       throws Throwable
     {
       new AudioWavPlayer
       (receivedMessageAlertFile).start();
       return invocation.invokeNext();
     }
}

  <bind pointcut="public void Client->
   privateMessageReceived(String name,
   String msg)">
  <interceptor class="AlertingInterceptor"/>
</bind>
```

Code Fragment 9: Pointcut definition and interceptor attachment in JBoss

When we considered the above differences, AspectJ appeared as a more suitable solution for our case. It provided the necessary components to define both dynamic and static crosscutting through an interface we were more familiar with. Due to its popularity, we could collect more technical support for AspectJ than any other framework. JBoss had advantages over AspectJ when working with the JBoss Application Server but we did not need this feature anyway.

5. Related Work

Because Aspect Oriented Programming (AOP) is an immerging technology, it is being started to be used in several different areas of software development. Obviously one of these areas is development of distributed applications. Developing distributed applications involves many different concerns like synchronization, security, fault tolerance etc. Several approaches have been presented for using AOP to deploy these concerns. One such approach is illustrated in [1] where advantages and disadvantages of using AOP in a distributed environment are discussed through a case study. The case study focuses on a framework for investigating and exchanging of algorithms in distributed systems called ALGON. The study concludes that using AOP has many advantages when dealing with complex systems but it should be used with caution. AOP provides good solutions especially for monitoring performance and fault tolerance.

Another approach for using AOP in distributed systems, especially in client-server applications, is illustrated in [2]. This work focuses on security concerns in network-enabled server-client architecture. Security problems may arise when a malicious user has complete access to system resources over a networked computer base. The paper presents a solution by starting with an existing chat application and developing aspects on top of that application with PROSE, a dynamic AOP platform.

Our approach is similar to that of [2], in the manner that we too are developing aspects on top of an existing chat application. However, we tried to develop aspects corresponding to different concerns instead of focusing on only one issue.

6. Conclusion

In this paper, we tried to investigate our work of building a multi-client chat application with AOP approach. The experience we have gained proves the fact that aspect-oriented programming enhances the object oriented design by modularizing crosscutting concerns. Although object-oriented approach helps to modularize system functionalities into different objects, there still remain some concepts that result in scattering. This was also the case in our situation. After designing our system with an object-oriented approach and improving this

architecture with design patterns, there were still some concerns that were scattered in several different places of the code and could be scattered more as the system functionalities increased. Handling exceptions related to sending messages and playing notification sounds were among such example concerns. AOP helped us to implement these concepts as separate aspects without changing the original code.

Besides modularizing crosscutting concerns, AOP also helped us to improve the development phase of our system. As an illustration, through the use of development aspects, we could easily define some enforcement policies to check the correct usage of classes. Furthermore, performance profiling is an important concern for distributed applications and AOP allows this concern to be implemented easily via defining logging and tracing aspects. For example, we implemented a simple aspect for collecting information about the times when clients connected to and disconnected from the server in order to determine when the server becomes highly populated. Similar information can easily be logged to improve the performance of the system.

On the whole, the importance of aspect-oriented approach for client-server applications became solid in our experiences. Even though we have focused only on a simple case of client-server applications, there were several concerns that had a high possibility of resulting in scattered code. These concerns could be expanded if more functionality was aimed. For example, the type of information logged to analyze the performance of the system could be varied as features like file transferring, video conference, and multi-user conversations were added. In addition, the alerts provided could be increased by notifying users when they add or remove friends, send a message to an offline friend, or have been added as a friend. Moreover, if the application were analyzed in a wider perspective, more concerns would be considered as crosscutting. As an illustration, several concerns like security and error handling become crucial for client-server applications, like any other distributed application. To be more specific, both of these concerns affect both the server and the client components of the system. Implementation of these concerns from an aspect-oriented perspective could be interesting. These mentioned points can be considered as feature work and implementation of these points can lead to more interesting points about using AOP in distributed applications.

Acknowledgements

We would like to thank Assist. Prof. Dr. Bedir Tekinerdogan for his comments and suggestions. This work has been carried out in the Aspect-Oriented Software Development class given by Mr. Tekinerdogan.

References

[1] S. Subotic, J. Bishop, and S. Gruner, "Aspect-Oriented Programming for a Distributed Framework", in the *Proceedings of SACJ*, 2006.

[2] P. Falcarin, R. Scandariato, and M. Baldi, "Remote Trust with Aspect-Oriented Programming", in the *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, 2006.

[3] http://www.damnhandy.com/jboss-aop-vs-aspectj-5-pt-2/

[4] http://www.eclipse.org/aspectj/

[5] Ramnivas Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., Greenwich, CT, 2003