**Java Chat Report**

Coursework

**Author:** COMP1549: Advanced Programming

Victor Dinev – 001115103 University of Greenwich

**Co-authors:** Old Royal Naval College

Zhenyuan Shen – 001197922 United Kingdom

Sapphire Nsinsi Nsiala – 001071889

Kurt K Parlak – 001028714

Tharshan Vijayakumaran – 001112250

***Abstract*** – **In this document we explain the process behind the given task to create a Java Chat System. The system works with a client and a server. They correspond through sockets, sending and reading strings. Each Client (user) chooses a unique username (ID) upon launch and gains access to the chat, where they can send group messages or private messages. The client also displays all online member and the coordinator, which is always the first to join the chat and changes whenever the coordinator disconnects. The server also keeps a record of the chat session in a Memory.txt file, found in the src folder. In this file you can find all recorded server sessions with a start date & time and then the server activities with timestamps.**

**I. Introduction**

In this report you will get familiar with basic and advanced programming principles and techniques used in building a software project. We will look not only at how to write a java chat program, but also how a project is organised and developed. We will also be exploring and demonstrating the main programming principles such as Design Patters, JUnit testing, Fault tolerance and more. By the end of this report, you should know the working process behind the development of software. In other words, you will be prepared to start your first job as a software developer.

**II. Design & Implementation**

First let’s look at the requirements for our application:

* Unique ID/username
* Send/Receive messages
* Public/Private messages
* Coordinator Role
* Display active members
* Timestamp messages
* Record and Store server activity

Those are the basic requirements our chat system should cover. In other words, brief description of what it should look and act like.

**Initial Design**

We decided to use the code from Lab 3 (DemoThree) as the base of our chat design. This code consists of ChatClient and ChatServer, which communicate by passing strings to one another, using socketing.

The Server is multithreaded allowing many users to join. It uses two sets, one for the users and one for the writers (PrintWriter used for communicating with the Clients) and a “Handler” class to handle login, messaging and disconnect. The Handler uses Scanner for input and PrintWriter for output. It works by using a Flag based system, applying a flag to every output. Flags are then interpreted by the Client. The flags are: “SUBMITNAME” asking for a name to be submitted, “NAMEACCEPTED” notifying that the name has been accepted and added to the set of names, “MESSAGE” meaning that the sting after is a message and should be displayed as such. It uses try/catch/finally to handle different scenarios. The login and messaging are part of “try” and disconnecting is handled under “finally” (executes if the “try” is over, meaning the client disconnected). The disconnect works by notifying everyone in the server about the disconnected user, removing them from the set of names and writers and closing the socket.

The client uses “java.swing” for the interface, which consists of message area (containing messages) and text field (used to write and send messages). It also uses a Scanner and PrintWriter for input/output (just like the server) and needs the server IP to be passed as an argument. It also has a getName() function, which shows a Input Dialog box, asking for username. In the run function uses a try/catch/finally statements, similar to the server. In the “try” part it tries to connect to the server and handles the input from the server (interprets the Strings received from the server and the Flags with which they start). In “finally” it disables the interface, executing that code in case of server disconnect.

Now let’s explore what else we must do:

* Coordinator Role
* Private Messages
* Display Active Members
* Timestamp Messages
* Record and Store server activity

**Timestamp**

Let’s start from an easier task and build up. Timestamp should be easy to do and won’t change as we complete the other tasks. In order to timestamp messages, we need to know what the local time is, and add it to the message. We can import and use java.time.LocalTime for that purpose. Now we can add to the messages a timestamp by using the following function: LocalTime.now(), however this displays seconds and nanoseconds as well, which is a bit overwhelming to look at. In order to make it simpler and display only hours and minutes, we can import and use java.time.format.DateTimeFormatter. We also decided to add the timestamp in brackets next to the name (of the sender of a message) and after the server messages (i.e., name has joined).

**Private Message**

Next, we will do the private messaging. For the private messages to work, we need to be able to send messages only to one user. Currently our code has a set of writers and whenever a message is sent, we loop through that set and send to everyone. To be able to know which writer is for which user, we will change the set of writers to a dictionary, with the name as key and the writer as value. But how would we know if a message is private or public? To send a private message the user will have to type “/[username] message” and this will send the message only to the given user. Now, that we have chosen the rule for sending private messages we will have to tell the server how to interpret and act with the new command. We will do this by using a simple if/else statement looking at the start of the message. Then we will use “try” to send the message and if that is not possible (i.e., a wrong username has been entered) we will add an exception stating that a wrong command was used.

**Coordinator**

So, we’ve done the timestamps and the private messaging, now let’s set up the coordinator role. We will do this by adding a coordinator variable which stores the name of the coordinator. This will be done in the client and the server, however we will do it for the client, when we do the display member list. This is because currently the client has no record of the members and we will add a way for the server to pass coordinator and members, later. After adding the coordinator variable, we need to give it value, or change its value, when the current coordinator leaves. The coordinator will be initially set to null and will be passed a value when the first user joins and when the current coordinator leaves. It will also need to be set to null when the last user leaves. We can do this with simple if/else statements in the appropriate places, which is straight forward, so we won’t be explaining it in detail here.

**Member List**

After the coordinator has been setup, we have to actually add it to the client as well. Now we will do this and also, setup the displaying of the active members, since they pretty much go together. For this we will add to the client a new JTextArea to display the members and coordinator. They will be displayed on the right-hand side of our main frame. Now we actually need to update the member list whenever someone joins or leaves. We will do this by a sending a string from the server with the appropriate Flag (“COORDINATOR”, “MEMBERS”). In the “COORDINATOR” message we will give the name of the coordinator and in the “MEMBERS” message we will send a list of all members.

It is important to add these to every code related to joining or leaving, so that every time there is a change the client will receive the new values. The client will then interpret the flags, just as it interprets every other Flag we use, meaning it will detect what the input starts with and act accordingly. It will store the coordinator in its own coordinator variable and then it will remove him from the members list. Removing the coordinator might be a bit tricky because, although we can use the replaceAll(coordinator, “”) function, it might remove part of someone’s name if it matches with the coordinator name (i.e., co, nameco – nameco will be only displayed as name). To go around this we will replaceAll(“ ” + coordinarot + “, ”, “ ”), this way we will only replace the coordinator, since another user cannot have “ ” in their name. Then we will clear all “ ”, and replace all “,” with “\n”. In order to not have any problems we have reworked the getName() function to actually check for “ ” and “,” and remove them. They are considered as forbidden characters.

**Server Memory**

The only requirement we have not covered yet is to record and store the server activity. Storing the server activity is fairly simple task. We will do this by adding a function called addToMemory() and a Memory.txt file, located in the src folder. The function will take a string as input and append it to the Memory.txt file. We will use a FileWriter for that purpose and after writing our simple function we can start implementing it in the code where needed. First, we want to store when the server was started, which would be in the main function. We will also store the date and time of the server start. Then we want to store every message that has been sent, as well as join/leave actions, which means calling our new function from the appropriate place in the code.

**Server Not Responding**

We will now create a refresh button for the client, so that it can work even if the server is offline. We will also make the client remember the last session it was in, meaning if you were already logged in, the client will automatically log you back in when refreshed. To do so, first we will declare a name variable in the client. Whenever we receive a NAMEACCEPTED we also receive the name with it. We add this name to the name variable. Then we would have to change the getName() function to check if name is empty. If it is empty, we display the login window, otherwise we return the name. But all that would be pointless if the client closes whenever the server disconnects. We will now create a pop-up window to show that the server is offline and give the option to retry connection. We will do this as an exception in the run method of the Chat Client. So if “try” fails (meaning the connection was interrupted), we will display a JOptionPane.showOptionDialog(…) with retry and cancel option. The option will be stored in an int where “1” would be cancel “-1” would be close (both exit the client) and 0 would be to retry connection. Worth noting is that the client session is stored locally and not in the server.

**III. Analysis & Critical Discussion**

We have looked at the design and implementation of our project, how it works and what we’ve done. Now we will analyse and discuss the workflow behind it. In other words, what was the process behind building this project.

We started by setting up a GitHub repository. This way we could all work simultaneously and easily keep track of the progress. We analysed the requirements and created issues for every Main task. Then we would create subtasks and declare whenever someone is working on a subtask. We also added labels to easily navigate through the issues. This helped a lot with the workflow of our team.

**Design Patterns**

As a programmer you often encounter problems that commonly occur, design patterns are the standard practice used to resolve this. Unlike algorithms which follow a fixed procedure in order to accomplish a specific task. Design patterns represent an idea and not a particular implementation these solutions were obtained via trial and error by a number of software developers over a substantial amount of time.

There are three main pattern groups of patterns:

* Structural patterns
* Creational patterns
* Behavioural patterns

In this case our program needs to be able to send requests to another program in order to access a service made available by the server. The chain of responsibility is a behavioural design pattern that chains multiple receiving in order to handle the sent request.

In this program we have created n/a handlers

* Names
* Messages
* Leaving

ExecutorService pool = Executors.*newFixedThreadPool*(100);

It is a thread pool implementation, a recognised design pattern that reuses a fixed number of existing threads instead of creating new ones for each task, allowing us more control over the program's resource consumption. When all threads are occupied, the executor will queue new tasks until a thread is available. Any thread exists until the pool is shut down, thus removing any queues. The Mediator design pattern implemented allows for communication between parts within an application. By limiting direct communication between objects and forcing them to collaborate only through a mediator object:

private static class Handler implements Runnable {

which typically handles all communications between different classes.

**Component-Based Development**

As programmers we often use component-based development. This is a set of tools and techniques used to achieve reusability of the code also allowing us to go through the entire program process using predefined components. In our Java Char Application, we have a few components that are easy to find in the code and edit for further improvement or just general understanding of the code. First let’s look at the components of the Server. That would be the thread pool found in the main function, allowing multiple users to join, the Handler which handles connection and communication between the server and the clients and the addToMemory() which handles storing server operations to the Memory file. For the Client components we have the interface which uses java.swing to display the interface to the user, the getName function as part of the interface that handles declaration of username, the run function which handles connection/ disconnection and interpreting the input from the server.

**JUnit Testing**

JUnit is a unit-testing framework in the Java ecosystem. It is programmer testing, called white-box testing, JUnit is a set of frameworks, inheriting the testcase class. In this java chatroom project, it can be created by two test files (ChatServerTest.java & ChatClientTest.java). For ChatServerTest.java, it has one class from the original code. To test it, we can fetch the text in console to check if the “main()” method is correct (by using “assertEquals()” in JUnit). For the “run()” part, it need to be divided into several tests to get the final results. For example, when someone is first to the chat room will be the coordinator or change the coordinator when the old one is leaving and notify the message to everyone in the channel. In these two examples we will use “assertEquals() and assertTrue()” to check the answer. As for the ChatClientTest.java, we do the JUnit test to check if the client passes the IP address to the server. By importing java.net.InetAddress and java.lang.String we can fetch the IP and check for consistency by using “assertTrue()”. But for the rest of the code, due to the limitations of JUnit Test, which cannot do a test for a private method. So, it only tests the main part.

**Fault Tolerance**

Fault Tolerance is important part of every program. It ensures that the program still works even if something unusual happens. We have already explained how we handle certain abnormal scenarios, so here we will go more in depth on how we tested our fault tolerance. First, we decided to check the connection and disconnection. We ran a couple of tests turning of the client and server in various ways. Also, what was very important was to see how the client works without the server and what happens if the connection is suddenly lost. We found and fixed some minor issues, but apart from that everything worked flawlessly. Then we decided to actually test the username and messaging systems. We tried to enter as input different characters and non-plain text. Everything worked well, although when we input code from the IDE we get an exception error, however the chat continues to work. We also had some problems with the username inputs and the private messaging, which would cause different anomalies in the working order of the program but after adding a few rules and exceptions, we managed to get everything in working order.

**IV. Conclusion**

We have developed a chat system in Java, containing a server and client. The system allows multiple users to join and send private and group messages, each with unique username. We have also looked at the main programming principles and concepts when it comes to software development and left some space for improvement. In the future various commands can be implemented to the chat as well as making the coordinator role significant (i.e., enabling the coordinator to kick members, create server password and wherever your creativity leads you). Adding new features should be very simple as the structure of the code is easy to understand and the main parts have been included, so you only would need to find where the changes should be written.

**References:**

refactoring.guru. (n.d.). *Chain of Responsibility*. [online] Available at: <https://refactoring.guru/design-patterns/chain-of-responsibility>.

sourcemaking.com. (n.d.). *Design Patterns and Refactoring*. [online] Available at: <https://sourcemaking.com/design_patterns/chain_of_responsibility>.

baeldung (2016). *A Guide to the Java ExecutorService | Baeldung*. [online] www.baeldung.com. Available at: <https://www.baeldung.com/java-executor-service-tutorial>

java-design-patterns.com. (n.d.). *Thread Pool*. [online] Available at: <https://java-design-patterns.com/patterns/thread-pool/>

www.tutorialspoint.com. (n.d.). *Design Patterns - Mediator Pattern*. [online] Available at: <https://www.tutorialspoint.com/design_pattern/mediator_pattern.htm>.

refactoring.guru. (n.d.). *Mediator*. [online] Available at: <https://refactoring.guru/design-patterns/mediator>