Ian Percy

Chat Application

Design Document

Table of Contents

1 Introduction

Chatting between each other digitally has become common place today. Since everyone uses a form of a chat application, the task at hand will be extremely relevant. Creation of a simple chat application will allow a deeper understanding of how a medium-sized software application works. The goal create a simple platform for chatting with a group or with another individual through a Java interface. Our requirements are to have the ability to register, login, chat, view chat history, and disconnect from the server. Implementation for this project will allow encapsulation of tasks into modules/classes to achieve the goal of accomplishing the main goal: creating a chat application.

1.1 Purpose and Scope

The purpose of this document is to describe the design of the Chat Application software as described in the Chat Application requirements document. The document will describe the design considerations, system architecture, and system design. Scope is to create a local chat application that will allow multiple users ability to chat simultaneously and view their past chat history.

1.2 Target Audience

Intended audience include the instruction staff of CS300 (Fei Xie and Bin Lin), other students from CS300, and also myself as the designer.

1.3 Terms and Definitions

Client-server- describing when a central server provides services to multiple connected workstations

Socket- internal endpoint for sending and receiving data through a single node

TCP- (Transmission Control Protocol) rules governing the delivery of data over internet using IP

2 Design Considerations

For this chat application to work properly, we must address the requirements carefully. Most important is allow multiple clients to connect to a chat server and these connections must be based on a successful authentication request with username and password.

2.1 Constraints and Dependencies

Based on the requirements there are certain features that a required to be implemented. The design must be centered around a server-client communication platform, from here these connections will be shown through a GUI. Constraints will come in how we will manage creating the new client connections and managing their object state. Also, management of the user data.

2.2 Methodology

Chosen to implement the project with incremental development practices. Creation of this project can be done while fully implementing the object modules one by one and bringing them together until all requirements are met.

3 System Overview

The chat application will use the structure from Figure 1 below. Chat application executables for the program will create a connection with the server. Server will manage each of the new clients, user authentication details, as well as chat history. The design for this will be based around a client-server TCP connection maintained through socket connections. The server will wait for requests from clients and accept their connection requests. This will allow for the creation of a connection stream and thus open a channel between the client and server. Java has useful implementation tools for management of sockets and input/output streams such as the ServerSocket and Socket classes.

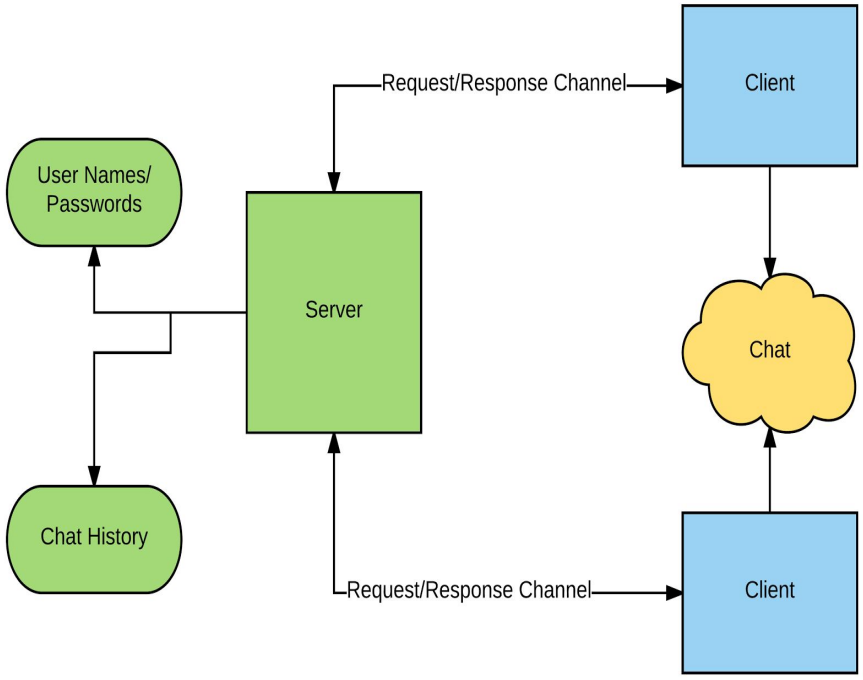

Figure 1: System Overview Diagram

4 System Architecture

The application is built around a server maintaining the user information and the client connections. Each of these components needs implementation in their separate classes. Important systems to note are a client, a server, storage of the chat history, storage of user information, server interface, client interface and a message object.

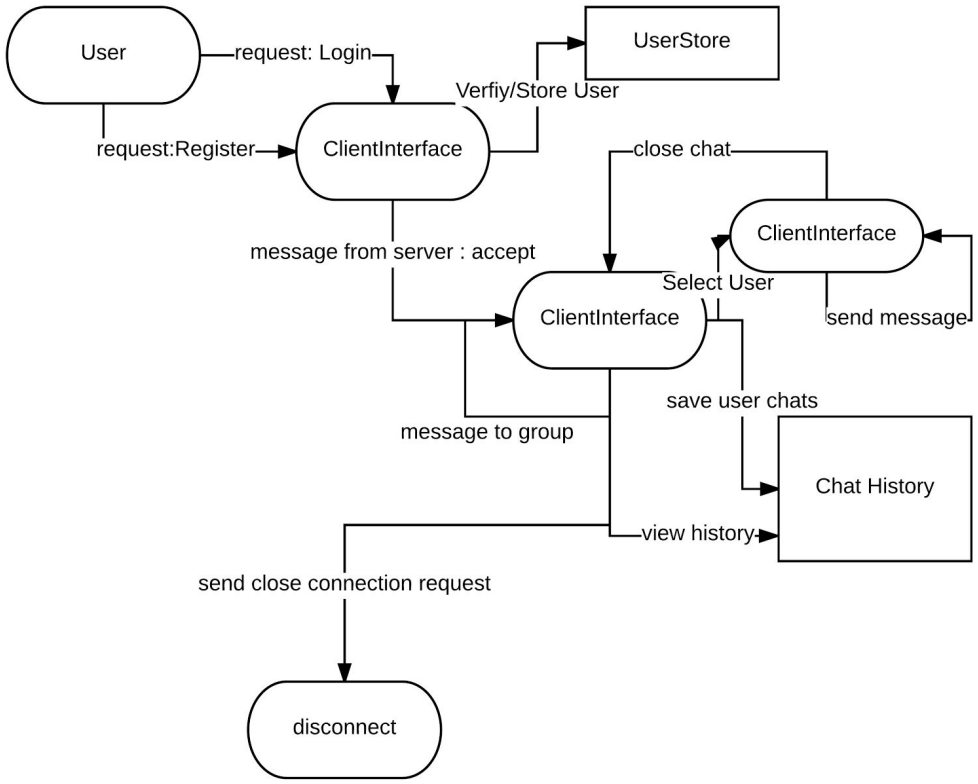

Figure 2: Data Flow Diagram

The figure above gives a general idea of the how requests and actions will create changes in the application. Users will login (register or login with credentials). Then they will be able to interact with the chat history or send messages to the group or individuals.

4.1 Server Class

The server class object for this project will have the task of connection and data management. Once a server has started running, it will be waiting for connections from clients and responding to their requests. A flow of this process is as follows:

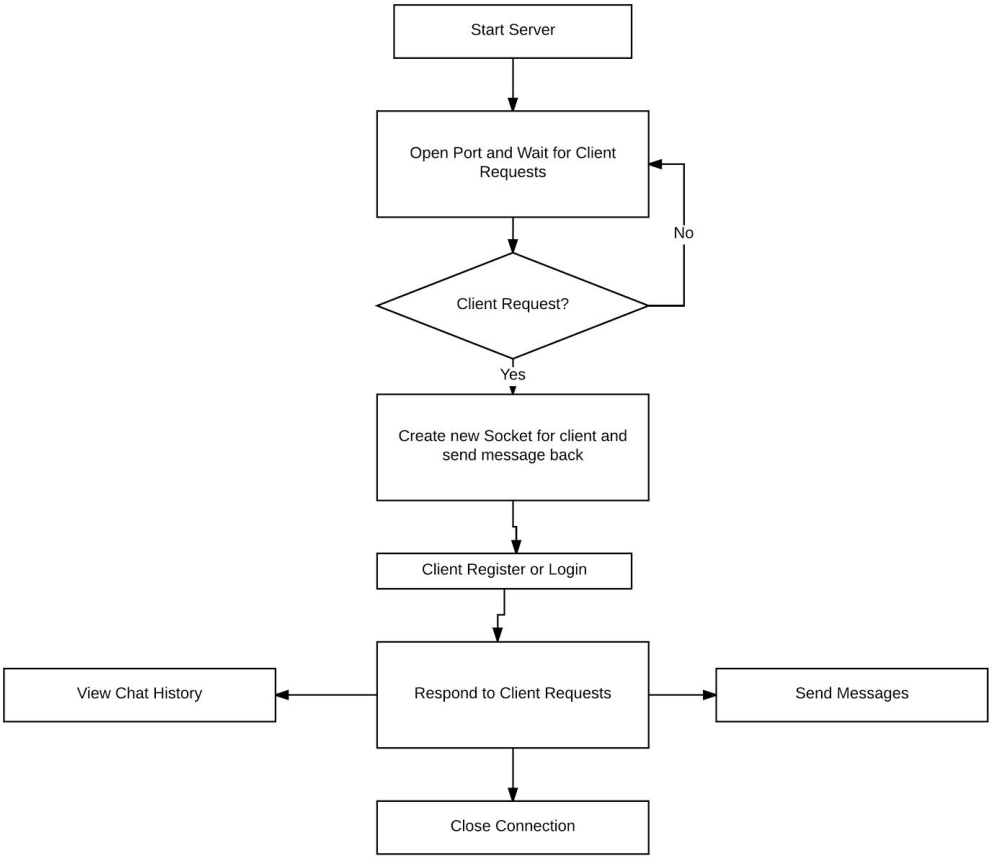

Figure 3: Server Basic Flow

Servers will continue running until a message is received to exit and close the port from accepting any more clients. Figure 3 shows the general logic flow for the server: boot, wait for client requests, connect with client, respond to requests from the client and close client. Any interactions (messages) will be loaded into the chat history portion of the program mentioned later.

4.2 Client Class

Clients will be the main actors for this system with many clients working with a server. A connection will be opened by the server and once the client requests to connect, then the user will be required to login or register with the server.

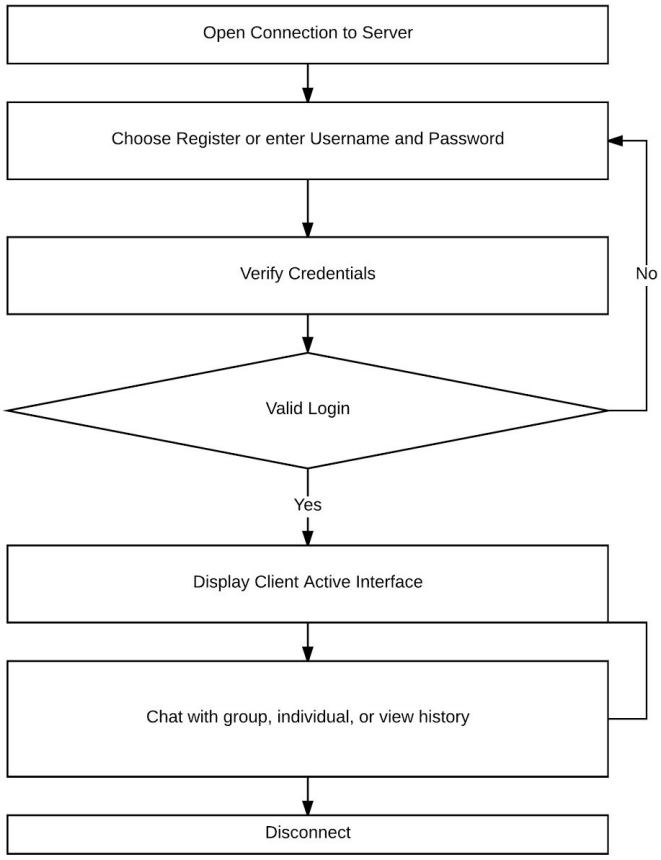

Figure 4: Client Basic Flow

Figure 4 gives a general overview of how the client control flow will execute. Once clients have authenticated with the server, the interface will be changed and display the active chat to the user. From here the user will have several options to send requests to the server: start a chat with an individual user, view chat history, logout, or send message to the general group chat.

4.3 Message Class

The message class will be the objects which are being communicated through the sockets from one client to the next. Since the application is communicating through Java, the design I have chosen is to actually send the object between clients, which will behave as a stream of bytes sent over the connection. This class will contain an int for the type and a String for the physical message.

4.4 History Class

A requirement for the project is to maintain user chat history, which will be managed by this class. The history class will maintain writing to a file and reading from the file when requested by the server.

4.5 UserStore Class

User information must be maintained for users who have connected to the system. the UserStore class will contain the information related to each of the users, mostly a key:value pairing for login verification by the server. This application will verify any information requests from the server or add new users if a registration has occurred.

4.6 ServerInterface Class

The server interface will be simple allowing the designer to change the port, close the port and start more easily than through a command console. As the project has been updated, there will be changes to this design document to reflect any views developed for the server interface.

4.7 ClientInterface Class

The client interface will have multiple views which need to be managed by the class. One for preauthentication where the user can choose to register or login. A second for after successful connection where the group chat and active users are displayed along with the

choices for chat history and logout. Third will be the different view when an individual chat has been started with another active client. The interfaces will need to send and receive the messages properly from server to client and vice versa. Again, as the project is updated there will be changes to this design document showing any different views developed for the client interface.

5 Detailed System Design

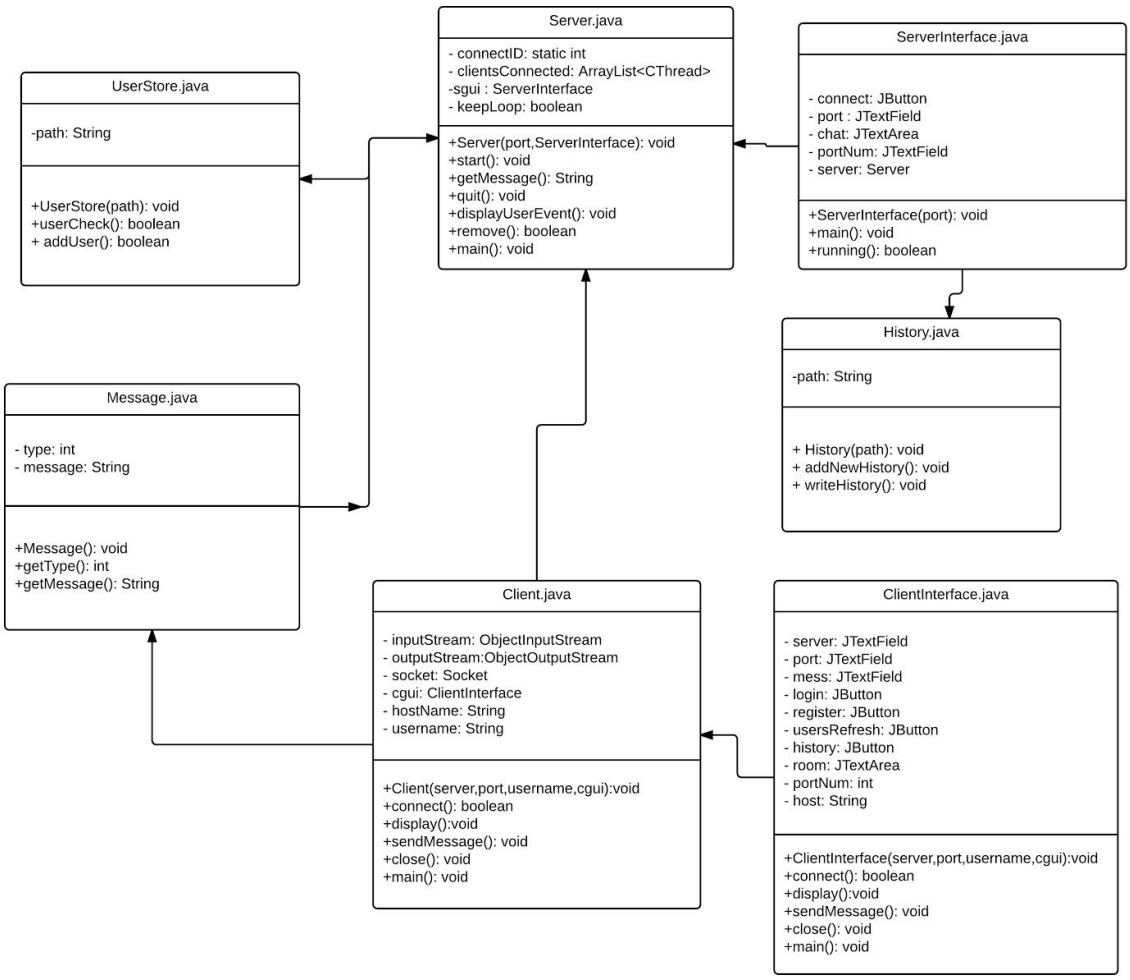

Figure 5: Chat Application Class Diagram

This figure gives the representation of the class interactions in the system. Server objects will interact with the server interface, history and userstore. Clients will interact with the client interface. Message objects can be sent between the Server and Client connections through the socket connections which will be created.

5.1 Server Class

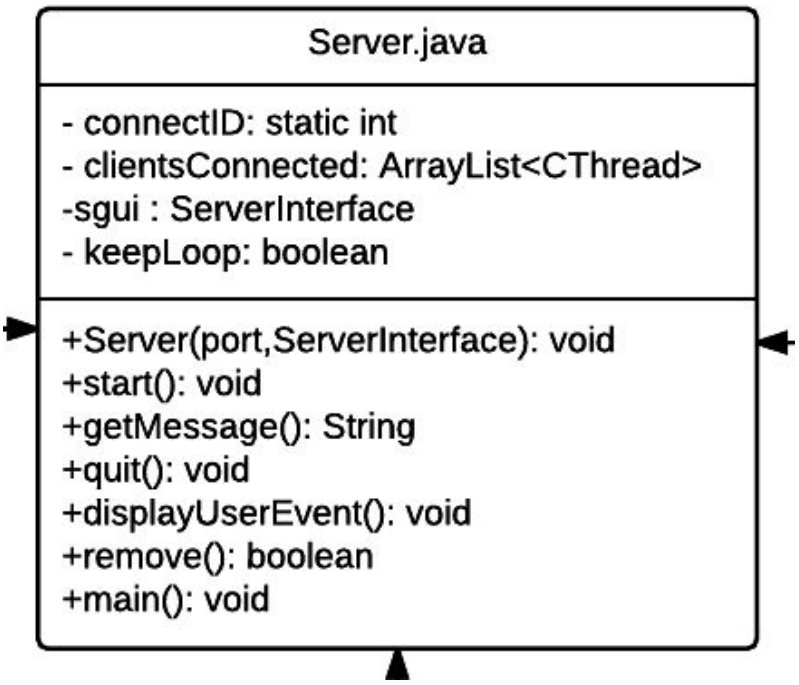

Figure 6: Server Class UML

Once the server has started running, which will be done with the call to start() from main. Here is the basic flow for the start() function:

loop(while true)

Create server socket/port

while(connection open)

accept any client requests

create new client thread

add to client thread ArrayList

close loop once socket closes

Threads will be created inside the server so that a multithreaded server application can be running and have accepted connections from multiple clients. Any of the responses seen from these thread streams will allow the server to broadcast the message to the clients

which are currently active. Threads will be added to the ArrayList and managed so that an active list of users is easily viewed. The connectID private member will be used for a unique for each thread.

For the server class, I have added the necessary function method of displayUserEvent() so that when client connections have been closed or opened, the interface can update and broadcast this message to the other clients.
Any interaction between the client threads and the server will have a particular action which the server will respond to.

getMessage() will implement a listening on the socket so that any of the interface actions can be met with an actual response from the server. Actions this will be checking for : register, login, user pressed the interface button for history, send message, or start new chat. Any new chat messages will also need to be sent to the history class. Any user information will be sent to the UserStore class for verification or registration.

remove() will allow the server to remove any client threads which no longer are active and have recieved a close connection from the client itself. This will be done in the following manner:
loop through array list checking each thread for close request

    match found
    close thread and remove from array list
    print error if thread does not close
    message other users of logout

Once a server has received a message to close its connection, the loop variables will then be set to false, all connections will be terminated and the server will close.

5.2 Client Class

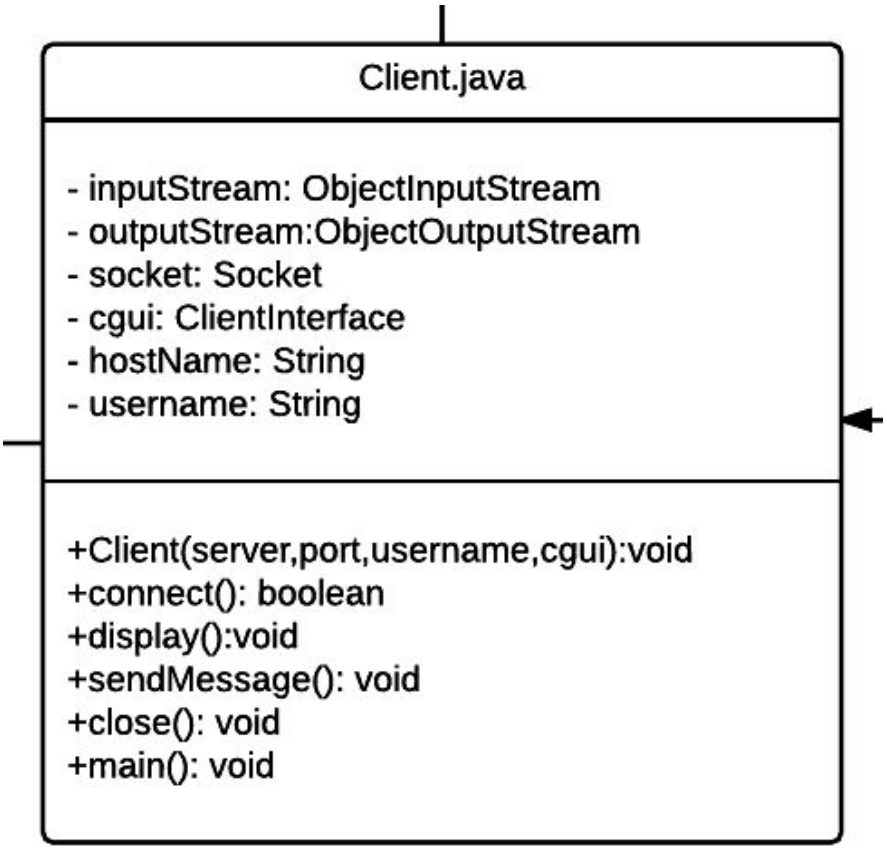

Figure 7: Client Class UML

Client class will maintain the state of the user connection. Once the constructor Client() has been created through main, then connect will be called. Connect will interact with the user choices of login or register. Connect will behave as follows:

try connection

accept socket connection to server

register or login with server

server sends failure or success to client

display() will interact with ClientInterface class to display any messages from the server

to the user. close() will take charge of sending a close connection to the server which will then close the thread on the socket and close the connection.

Messages will be sent based on the user's interaction with the client interface and any of the gui fields. Messages typed will be sent between the input and output streams of the interfaces. Any choices made will be reflected through the client.

5.3 Message

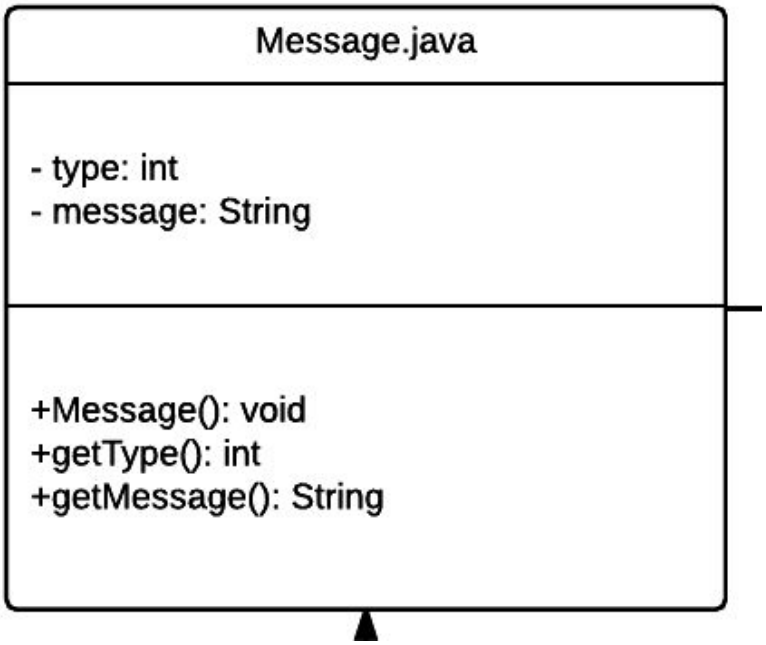

Figure 8: Message Class UML

As stated before, the message class will be the object sent in between servers and clients. The type can be changed depending on the type of message to be sent. String will contain the actual text message to be sent. Having a message object allows encapsulation of the message details to be enforced as well as possibly serialized if security of the messages was necessary. The class will primarily be used for the getter functions of the private variables.

5.4 History Class

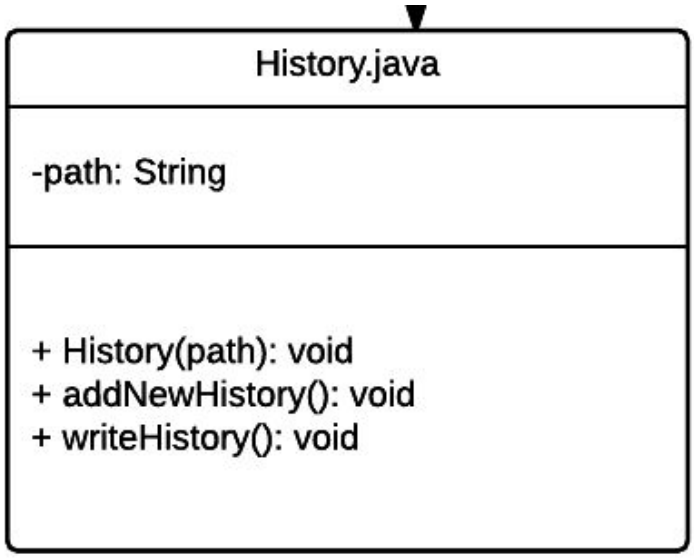

Figure 9: History Class UML

User's chat history will be maintained by the History class. For this, all history will be stored in a local file with the server executable. Currently, the information will be stored using DocumentBuilder class unless an alternative is found. This class will allow the server to interact with the history and to add or view any of the history for a particular user (if a match is found). Document builder will allow the information to be stored in an XML and parsed by the any DocumentBuilder functions.

5.5 UserStore Class

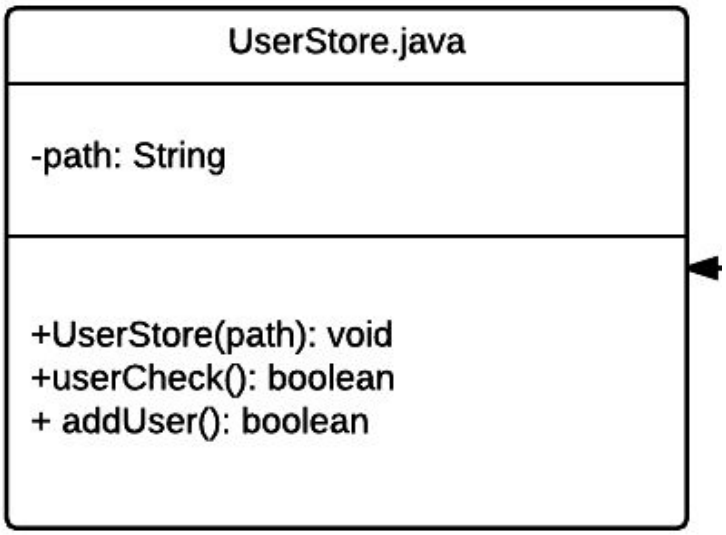

Figure 10: UserStore Class UML

User information will be maintained by this class. Currently, there will need to be a couple other local variables for the functions what will correspond to the username and passwords. Passwords will be hashed to help create a numerical representation of their strings. Servers will send request to this class for userCheck() which will return boolean value for any information passed to the function. addUser() will effectively add a new file into the user database.

Again, as with history this will be stored using a file path and the DocumentBuilder class. If an alternative for storing is found besides DocumentBuilder this is will be changed.

5.6 ServerInterface

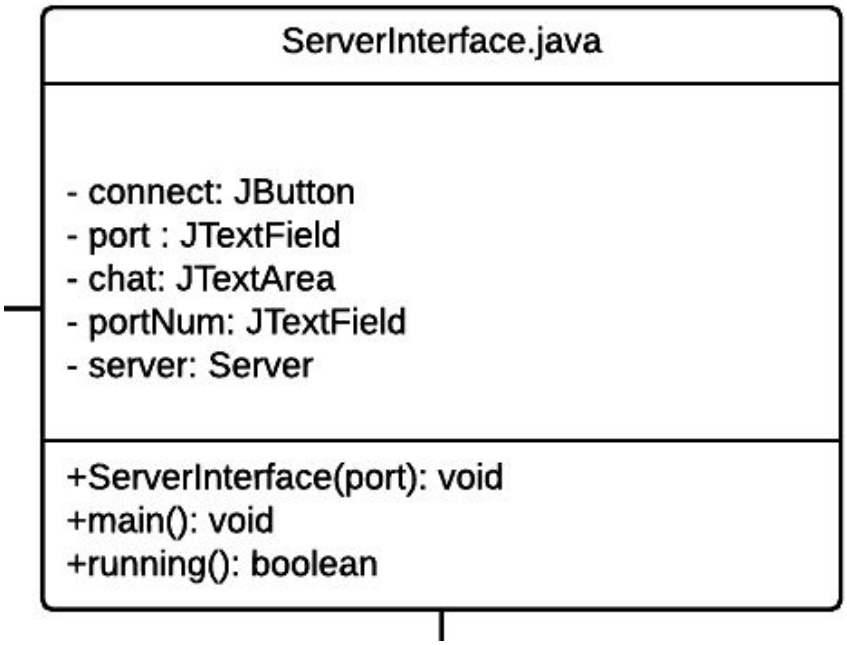

Figure 11: ServerInterface Class UML

The ServerInterface class will be interacted with at the creation of the server, but will not need to manipulated during execution until time to close the connection. Here the frame will give the option for entering a port and to connect. There will also be an area for the chat, but this will only be interacted with by the user and messages will be sent on behalf of the server automatically. Running() will be called when needing to check to see if the server is still active and if not to close any connections.

5.7 ClientInterface

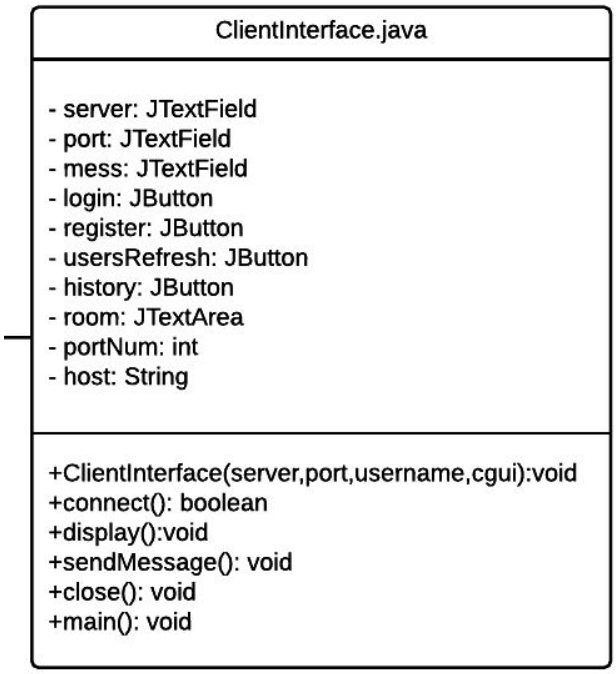

Figure 12: ClientInterface Class UML

ClientInterface class is the more important interface to be implemented for the project. The client will have several changes in views depending on current state of the application. First, port and server fields will be able to be changed to open a connection with the server. Once a connection has been created, the user will need to enter a username/password or to choose register. After validation, then the user will be presented with the chat and active users.

JButtons for history and refreshing the users will be in the view so that the user will have many active choices. The majority of the view will be dedicated to the text area for the room chat (group chat). The list of users in the view can also be interacted with to allow the user to change the view and start an individual chat with a user if the chat connection is accepted.