Design Document

Jason Yang, Jenny Lin, Sayeed Tasnim

Server

Package: server.rooms

Class: ChatRoom:

- Attributes:
  - String name: The name of the particular chatroom
  - RoomList room: A RoomList object that contains the list of rooms to which the particular ChatRoom belongs
  - ChatUserList connectedClients: A ChatUserList object that contains a list of the users that are currently connected to the particular chatroom
  - LinkedBlockingQueue<String> messageBuffer: A buffer that contains the list of incoming messages from users. A consumer consumes from the queue and sends to all the connected clients.
  - Thread self: The particular thread for ChatRoom that consumes from the LinkedBlockingQueue
- Methods:
  - ChatRoom(String name, RoomList rooms, ConnectionHandler connection): The constructor adds itself to the list of chat rooms of the user in a synchronized method if the room name does not already exist. Otherwise, it throws an exception.
  - run(): Method that controls the main loop for the chatroom that loops as its alive (when there is a positive number of people in the chatroom), consumes from the message buffer, and relays the message to all the users and ConnectionHandlers connected to the room.
  - addUser(ConnectionHandler connection): Adds a user to a room
  - removeUser(ConnectionHandler connection): Removes a user from room
  - cleanup(): The room removes itself from the list of rooms of the server
  - updateQueue(String info): Adds a command or message to the buffer
  - isAlive(): Returns a boolean whether the room is alive (a positive number of people in the chatroom)
  - getList(): Returns the ChatUserList of the chatroom

Class: RoomList

- Attributes:
  - ServerUserList users: A ServerUserList of all the users connected to the server
  - Map<String, ChatRoom> rooms: A Map with String keys of the orom names to the ChatRoom objects for the chatroom of the same name
- Methods:
  - RoomList(ServerUserList users): Constructor
  - add(ChatRoom room): Adds a ChatRoom object to the map and informs

all the users that a new room has been added to the server
        - remove(ChatRoom room): Removes the specified ChatRoom from the server and informs all users connected to the server about the change
        - contains(String name): Returns a boolean whether a room name is in the list of rooms of the map
        - getRoomFromName(String roomName): Returns the ChatRoom with the specified room name
        - getRooms(): Returns a String containing the sequence of all the rooms on the server
        - updateUser(ConnectionHandler user): Updates a specific user of all the rooms in the rooms list.
        - getMap(): Returns the map of the String chat room names to the ChatRoom objects

Package server.lists

Class: UserList (abstract)

- Attributes:
    - Map<String, ConnectionHandler> users: A map of usernames to the associated ConnectionHandler
- Methods:
    - add(ConnectionHandler connection): Adds a ConnectionHandler to the list of users and informs all users of the new change of users connected to the server
    - remove(ConnectionHandler connection): Removes a ConnectionHandler from the list of users and informs all users of the new change of users connected to the server
    - contains(String userName): Returns a boolean whether the given userName is in the UserList and has an associated ConnectionHandler
    - getList(): Returns a string representation of everyone in the list
    - size(): Returns the size of the list of users
    - informAll(String message): Informs everyone on the list with the particular message
    - getMap(): Returns the Map of the usernames to the ConnectionHandlers

Class: ServerUserList (extends UserList)

- Methods:
    - getList(): Returns a String command with the header serverUserList and the list of users on the server

Class ChatUserList (extends UserList)

- Attributes:
    - String name: The name of the particular chatroom of the user list
- Methods:
    - ChatUserList(String name): Constructor that takes in the name of the chatroom
    - getList(): Returns a String command with the header chatUserList

followed by the name of the chatroom and the list of users in the chatroom

Package: server

Class: ConnectionHandler

- Attributes:
  - String username: The username associated with the ConnectionHandler
  - Socket socket: The socket the ConnectionHandler uses for io
  - RoomList rooms: The list of rooms of the server
  - ServerUserList users: The list of users in the server
  - HashMap<String, ChatRoom> connectedRooms: A map from the String room name key of a chat room to which the user is connected to the ChatRoom object
  - BufferedReader in: The input stream from user to the server
  - PrintWriter out: The output stream from the server to the user
  - LinkedBlockingQueue<String> outputBuffer: The output queue that is to be sent to the user from which the ConnectionHandler will consume
  - Thread outputConsumer: The consumer thread for the output queue
  - boolean alive: The state of the ConnectionHandler, true if connected, false otherwise
- Methods:
  - ConnectionHandler(Socket socket, RoomList rooms, ServerUserList users): The constructor for the ConnectionHandler. Initializes the variables.
  - ConnectionHandler(string username): Constructs a ConnectionHandler with the given username and sets all the other variables to null.
  - run(): Begins the 'handshake' with the outputConsumer described later. It obtains a valid username from the client and begins an outputConsumer thread. It reads the input stream, parses it, sends it to the queue filter, and loops. If the user is disconnected, it also closes the stream and cleanly removes the user.
  - parseInput(String input): Returns a String after parsing the input from the user. Uses a regex to determine if it's valid. Then it performs the respective action based on the command. If the server does not have to reply back with anything immediate, it returns the empty string.
  - informConnectedRooms(): Returns a String of the rooms the user is connected to preceded by the header "ConnectedRooms"
  - parseOutput(String input): A filter that sends the String input to user if the String is nonempty
  - removeAllConnections(): Removes the user and ConnectionHandler from all the rooms it is connected to and from the list of users
  - updateQueue(String info): Adds the String info into the outputBuffer, can be used externally by other methods

Class: Server

- Attributes:
  - ServerSocket serverSocket: Server socket to handle io
  - RoomList rooms: List of all rooms contained within the server
  - ServerUserList users: List of all users connected to the server
- Methods:
  - Server(int port): Constructs a server with the given port number
  - serve(): Runs the server while listening to connections and creates ConnectionHandlers for those connections
  - isInteger(String s): Returns true if s can be parsed into an integer, false otherwise

Client

Package: client.gui

Class: MainWindow

This class represents the JFrame that holds all of the tab components. It has a TabBar that contains a permanent maintab, where the list of chatrooms and usernames reside, and any number of chat tabs/history tabs that can be closed. It also contains a menu with two options, Chat History and Logout. Choosing Chat History causes a history tab to be created, and choosing Logout causes the application to close. The MainWindow is also the home of all the models and contains appropriate methods to modify those methods on the EDT by using invokeLater.

Class: MainTab

This Main tab of our GUIChat. It contains a list of all chatrooms, a list of all users and a button for creating a new chat. Clicking on the New ChatRoom button will trigger a popup prompting the user for a chatroom name. Submitting an empty string as a name will cause the popup to continue triggering until the user submits a non-empty chat name or they click on the cancel/exit button of the popup. An invalid/taken chatroom name will cause another popup to trigger with the appropriate error message. Double clicking on a chatroom in the chatroom list will cause the user to join the chatroom and open up a new tab containing that chat. Trying to join a room that the user has already joined will cause an error popup to occur. The list of users will update to match the server, but cannot be interacted with. It can only be added to a MainWindow and not any other JFrames. It takes in the MainWindow it is a part of as an argument.

Class: ChatTab

A class representing the gui for a single chatroom. Messages are entered into a text field and submitted to the server either by hitting the send button or by pressing ENTER on the keyboard. The conversation is displayed in a text pane on the left, and all users in the chatroom are displayed on the right. Earlier chat history is displayed in the window upon the user rejoining the room, where the history is defined as conversation held while the user is present in the room, and rejoining is defined as entering a chatroom

with the exact same name as a previously entered chatroom.

Class: HistoryTab

This class represents the history of a user's chat. It contains a list with all the chatrooms the user has joined prior to the opening of a the history tab. In order to view the history of a chat joined after the tab has been opened, a new History Tab must be made. Selecting a chatroom on the list causes the text pane to display the history of that chatroom up until the user closed the chatroom. If the user is still in the chatroom, the history will also update itself to remain consistent with the open chatroom.

Class: LoginWindow

This class represents the login window for the chat. It opens upon application start. While the login window is open, it is impossible to interact with the chat at all until a valid username, ip address and port number is given. This includes trying to use the application's title bar. Closing the login window closes the application. This performs concurrently with the server's "handshake" until it is confirmed.

Package: client

Class: Message

- Attributes:
  - String username: The username of the user sending the message
  - String message: The message the username sent
- Methods:
  - Message(String u, String m): Constructor for a message
  - getUsername(): Returns the username of the Message
  - getMessage(): Returns the message of the Message

Class: ChatRoomClient

- Attributes:
  - String chatRoomName: The name of the chatroom
  - ArrayList<Message> messageHistory: The message history the user has of the chat.
  - DefaultStyledDocument displayedMessages: The document that has all the messages contained in a nice styled document to be displayed in a tet area.
  - DefaultListModel userModel: The default list model to hold the list list of users connected to the chatroom
  - String myUsername: A string representing the client's username
- Methods:
  - ChatRoomClient(String nameOfChatRoom, String username)
  - addMessage(Message message): Adds the message to the message history and updates the styled document with the message.
  - updateUsers(ArrayList<String> newUsers): Takes in an array list of the entire list of users and updates the list of users.

        - getDoc(): Returns the DefaultStyledDocument to display the messages
        - getChatRoomName(): Returns the name of the chatroom
        - getUserListModel(): Returns the user list model

Class: Client

- Attributes:
    - String username: The username of the client
    - Socket socket: The socket the client uses for io
    - PrintWriter out: The ouput stream of the client socket to the server
    - BufferedReader in: The input stream of the client from the server
- Methods:
    - Client(String username, StringIPAddress, int port): Constructor for the client. The constructor begins the input and output stream and communicates with the LoginWindow JDialog. The LoginWindow JDialog does not close until the construction of client is complete. The Client attempts to communicate by requesting for one username. If the username is valid, the Client construction completes and returns a Client object to the LoginWindow. If the username is invalid, then the server returns an error to the Client at which point the constructor throws an IOException. The LoginWindow catches the IOException and displays the error message to the GUI.
    - readBuffer(): Returns a String obtained from the input stream
    - send(String output): Sends the String to the output stream to the server
    - getUsername(): Returns the String of the username of the client
    - start(MainWindow main): The start method begins to loop and read from the input stream and parses it.
    - parseInput(String input, MainWindow main): This method parses the input from the server and performs the respective action. Each of the respective actions are performed in event dispatch threads to avoid concurrency issues in Swing.

Class: CompleteChat

- Attributes:
    - LoginWindow login: The LoginWindow associated with the chat user
    - MainWindow main: The MainWindow associated with the chat user
    - Client c: The client the chat user is currently using
    - Thread consumer: A consumer thread that starts the Client
- Methods:
    - start(): Starts the entire chat in an event dispatch thread and opens the LoginWindow and the JDialog has a modality feature that stays waits until a valid client is returned once the user has successfully connected to a server.

Client-Server Protocol

From client to server:

- connect [username]
  - Sent when a client attempts to connect with a username
- disconnect [username]
  - Sent when a client with the specified username attempts to disconnect
- make [chatroom name]
  - Sent when a client attempts to make the specified chatroom
- join [chatroom name]
  - Sent when a client attempts to join the specified chatroom
- exit [chatroom name]
  - Sent when a client attempts to exist the specified chatroom
- message [chatroom name] [message]
  - Sent when a user attempts to send the particular message to the specified chatroom.

From server to client:

- connectedServer
  - Sent when the server successfully connects the user
- invalidUsername
  - Sent when the user submits a username that is already taken
- errorMessagePrompt
  - Sent when the user cannot successfully connect to the server
- disconnectedServer
  - Sent when the server is about to disconnect the user
- connectedRoom [chatroom name] [username]
  - Sent when the user connects to the specified room
- disconnectedRoom [chatroom name] [username]
  - Sent when the the user disconnects from the specified room
- invalidRoom [chatroom name] [message]
  - Sent if the room that the user wants to create or join cannot be created or joined with a message as to why
- serverUserList [username list]
  - Sent when the server updates the list of users on the entire server
- chatUserList [chatroom name] [username list]
  - Sent when the chatroom updates the list of users in the chat
- serverRoomList [chatroom list]
  - Sent when the server updates the list of chatrooms on the entire server
- message [chatroom name] [username] [message]
  Sent when the chatroom sends a message to the users

Client to Server Grammar

COMMAND :== (CONNECT | DISCONNECT | MAKECHAT | JOINCHAT | EXITCHAT | MESSAGE) NEWLINE
CONNECT :== "connect " USERNAME
DISCONNECT :== "disconnect “ USERNAME
MAKECHAT :== "make " CHATROOMNAME
JOINCHAT :== "join " CHATROOMNAME
EXITCHAT :== "exit " CHATROOMNAME
MESSAGE :== "message " CHATROOMNAME " " MESSAGETEXT
USERNAME :== [\p{Graph}]+
CHATROOMNAME :== [\p{Graph}]+
MESSAGETEXT :== [\p{Print}]+
NEWLINE = "\r?\n"

Server to Client Grammar

COMMAND:== (CONNECTSERVER | INVALIDUSERNAME | ERRORMESSAGEPROMPT | DISCONNECTSERVER | CONNECTROOM | DISCONNECTROOM | INVALID ROOM | SERVERUSERLIST | CLIENTROOMLIST | CHATUSERLIST | SERVERROOMLIST | MESSAGE) NEWLINE
CONNECTEDSERVER :== ‘Connected”
INVALIDUSERNAME :== “invalidUsernameSent”
DISCONNECTEDSERVER :== “disconnectedServerSent”
ERRORMESSAGEPROMPT :== [\p{Print}]+
CONNECTROOM :== “connectedRoom “ CHATROOMNAME “ “ USERNAME
DISCONNECTROOM :== “disconnectedRoom ” CHATROOMNAME “ “ USERNAME
INVALIDROOM :== “invalidRoom” CHATROOMNAME MESSAGETEXT
SERVERUSERLIST :== “serverUserList” (“ “ USERNAMELIST)?
CHATUSERLIST :== “chatUserList” CHATROOMNAME (“ “ USERNAMELIST)?
SERVERROOMLIST :== “serverRoomList“ (“ “ CHATROOMLIST)?
CLIENTROOMLIST :== “clientRoomList “ USERNAME “ “ CHATROOMLIST
MESSAGE :== “message “ CHATROOMNAME “ “ USERNAME “ “ MESSAGETEXT
USERNAMELIST :== (USERNAME “ “)* USERNAME
CHATROOMLIST :== (CHATROOMNAME “ “)* CHATROOMNAME
USERNAME :== [\p{Graph}]+
CHATROOMNAME :== [\p{Graph}]+
MESSAGETEXT :== [\p{Print}]+
NEWLINE = "\r?\n"

Notes: \p{Graph} is any alphanumeric character or punctuation and \p{Print} is any printable character (alphanumeric, punctuation, or space).

Before a client attempts to connect, the client will need to provide a username first. This will establish the connection and send the string “connect [username]”. After the client side checks to make sure that the username contains valid characters only (no spaces), the server will check the available list of usernames and if the username has not been taken yet, the server will construct a ClientHandler for the particular client and add the client to the UserListHandler. The server will send a “connectedServer” command to let the client know that the client is connected and then display the UserListHandler and the ChatHandler to the client as a list of current users and chats by sending a “serverUserList” command. Otherwise, if the username is taken, the server will send a ‘serverUserList’ message with a list of users in the chat displayed in a separate window and an ‘invalid’ message to the user. This will state to the user that the username is already taken and that the user should select a different chat name.

Whenever the list of all the server users is updated, the server sends a chatUserList command to each of the clients.

When a user requests to make or join a chatroom, the client will send the respective command “make” or “join” to the server. The GUI will create a new window for the specific chat and the user will be able to send messages through a text box. The server will send a “connectedRoom” command to let the client know that the client is connected to the chatroom. When making a chatroom, the chatroom’s name cannot already be taken, otherwise an ‘invalid’ message will sent to the user, prompting them to select a different name for the chatroom.

When a user sends messages, the client sends a “message” command to the server. When a server receives the “message” command, it will process it and place the message into the proper conversation queue. The conversation queue will consume the messages and the server will send “message” commands to each of the clients in the particular chat room.

When the list of users in the particular chat room changes, the server will send a “chatUserList” command to each of the clients in the particular chat room. This will update each client and GUI with a new list of users in the chatroom.

When a user disconnects from a chatroom, the client will send an “exit” command to the server. The server will process the disconnection and relay a “disconnectedRoom” command back to the client if the client is still connected to the server.

When the user disconnects from the server, a “disconnect” command is sent to the server. The server will then remove the user from all chatrooms and from the list of all users on the server and return a “disconnectedServer” command. All open GUI chat windows will be closed except for the original username prompt window. The user will need to login again in order to reconnect to the chatroom.

Conversation

The ChatRoom will be a class that serves as one ‘room’ of people chatting. It will contain a list of all the connected clients in that room (subscriber) and upon receiving a message from the server with text, update all of the connected clients of the change (publisher). People are free to come and go from this object (subscribe/unsubscribe). As long as one person remains in the room, the ChatRoom will still be valid and upon becoming invalid, the ChatRoom will deregister itself from the server and all connected clients will be notified of this change.

ChatRoom classes will contain an instance of UserList that is a subset of the UserList that the server has. The Conversation will need at least one user as a ConnectionHandler in the list of ConnectionHandlers for the ChatRoom to hold. Otherwise, the server will delete the ConnectionHandler by calling the respective command. The ChatRoom will have a blocking queue of the messages to be passed to the members of this conversation. This class will be threaded such that the thread checks if anything is in the queue and if so, alert all parties. This class will also have provisions to add or remove a client from this object, updating the respective lists and informing all members of this change.

Concurrency

On the server side, the ConnectionHandler and the ChatRoom objects both receive individual threads. This is so that multiple ChatRooms can process messages at the same time, and multiple ConnectionHandlers can handle connections at the same time. This works because the connection handler will have an input and output buffers. any message that has to be sent to the client will be queued in a blocking queue which is thread safe. One thread will consume from this queue in its own time and push the messages to the network. The other thread will similarly check the buffered reader for data and parse it. This makes the connection handler modular such that multiple people can queue writes to it. Similarly, multiple people can write to a chat room using this queue strategy.

In order to prevent race conditions from occurring, we will make sure that the different list objects are all threadsafe, since the list objects are the only modifiable objects shared between the different threads. In RoomList, getRooms iterates over the list of Rooms, but it is a private method that is only called by add and remove, so we do not need to worry about RoomList’s hashmap getting modified while we are iterating over its values. This means we can make RoomList threadsafe just by synchronizing the add and remove methods. In UserList, however, informAll is a public method, so we are not allowed to directly iterate over the HashMap’s values, as the add or remove method could be called during iteration, modifying the collection of Connection Handlers and producing incorrect output. Therefore every time a call to informAll is made, a copy of the HashMap’s values is made for informAll to iterate over. Therefore if someone is added to the userList after informAll has been called, they simply will not receive the message. If someone is removed from the ServerUserList after informAll has been called, that ConnectionHandler’s queue will still receive a message, but because the client for that ConnectionHandler has left the entire server, there will no longer be a thread to read from the queue, so the message will just sit there unread. If someone is removed from the ChatUserList after informAll had been called, the client will just receive a message for a chat they are no

longer a member of, and the message will be discarded clientside. The copying of the Hashmap's values, the add method and the remove method are all synchronized on UserList's HashMap to prevent concurrency issues.

When the client tries to connect to the server, a thread is made that attempts to create a connection Handler. This is to allow multiple clients to connect to the server at the same time. The only potential concurrency issue comes when the thread adds the new connectionHandler to the server's userList, and that issue was handled when we made the Lists theadsafe. Once the connectionHandler is successfully made, the thread terminates.

Aside from the event dispatch thread that handles all the GUI issues, the client will also have a background thread that reads messages from the server and calls the appropriate method to modify the appropriate model. Each method that modifies a method will work by using InvokeLater so that the actual modifying of the model will occur on the event dispatch thread, making the overall chat threadsafe. The Gui will then update itself based on changes in the model. There will not be a separate thread for sending messages to the server, as we will just use action listeners to send messages when the appropriate action event occurs and then immediately go back to dealing with the Gui.

Testing Strategy

Serverside Testing

- Test Connection Handler
  - Make sure that a connection handler can be instantiated properly
    - Should be getting the right output
    - Should be getting added to the userlist *after* username is chosen
    - Should close socket if username is already taken
  - Check that parseInputs does what is expected
    - try disconnecting (both nicely and forcefully)
    - make a chatroom (both existing and non existing)
    - join a chatroom (both existing and non existing)
    - leave a chatroom (one connectionhandler is a part of, one it isn't a part of, and one that doesn't exist)
    - send a message to a chatroom (one connectionhandler is a part of, one it isn't a part of, and one that doesn't exist)
  - Check that removeAllConnections does what is expected
    - try with both empty and full list of connected chatrooms
  - Make sure updateQueue works properly
    - spam a bunch of threads and update the queue all at once to make sure no messages are lost
- Test the List Objects independently
  - Make a bunch of connection handlers to add and remove from a single server user list
  - Make a bunch of connection handlers to add and remove from a single chat user list

        - Make a single chat user list, but make two threads that add/remove from it at the same time.
        - Make a chatroom list and add/remove a bunch of chatrooms from it
        - Have multiple threads add/remove chatrooms from a chatroom list
    - Test the Chatroom Object independently
        - Create a chatroom, add a bunch of users to it and then remove them all
            - Chatroom should add self to list when starting, then terminate thread and remove self from chatroom list once empty
        - Make sure updateQueue is working
            - make a bunch of threads to send messages to the chatroom. then make sure all the users in it got all the messages
    - Test the Server as a whole
        - Start the server, have multiple connections made through socket stubs, and have them communicate with each other. Check output to see that it's as expected

Clientside Testing

- Test individual chatting system
    - Create a valid username and attempt to login
    - Create an invalid username and check to see an indication if the username is taken or invalid
    - Create a chatroom with a valid chatroom name
    - Create a chatroom with an invalid chatroom name (bad characters/already taken)
    - Join chatrooms
    - Exit chatrooms
    - Message all the users in the chatroom
- Test multiple tabbed chatting system
    - Create and join multiple chatrooms and messages to different chatrooms
    - Check for concurrency bugs for a particular client
- Test multiple users in a conversation
    - Multiple users will message in the conversation
    - Users will join and exit the conversation and the output of the messages will be checked against expected chatroom behavior
- Combine multiple chatrooms and multiple users
    - Consider all cases of performing actions on the room, e.g. The last person exits the room and someone attempts to join the room at the same time.
- Deal with sudden disconnects
    - Have a user join multiple chatrooms and then disconnect from the entire chat
- Gui testing
    - Make sure gui components are updated when the clientside model is updated

        - Make sure that the display doesn't freeze up no matter how many messages are being sent from the server (a lot!)
        - Make sure that displays are dynamic; room list and user list updates without the need for refreshing
        - Make sure all the components are sitting in the right place

Testing Strategy (GUI Version)

When testing the gui, we start the server and then run one instance of the chatroom. We check to make sure that the main chat window appears like expected and that a login window also appears. The login window should block all access to the main window (key presses, mouse clicks, etc). We then check to make sure that the appropriate error messages are being shown for invalid login attempts:

- Blank Username = Error: Valid username required to login
- Blank IP Address = Error: IP Address required to connect to server
- Blank Port Number = Error: Port Number required to connect
- Invalid Username (contains spaces/is too long): Username can't exceed 20 characters or contain any whitespace
- Bad Port Number/IP Address = Error: Unable to resolve host
    - The gui locks up while the client attempts to connect to the server and does not unlock until it either connects or acquires a response

Once we do log in with a good username we check to make sure that the welcome message does display the username of the person currently logged in. We now start by creating a chatroom with a variety of names:

- A normal name (ie puppies)
- A normal name with numbers/puncutation (puppies_4ever)
- A not quite normal name with nothing but numbers/punctuation (728492&&*3420)
- A name that is the same as an existing room except for punctuation (ie PUPPIES)
- A name that is exactly the same as an existing room
    - results in an error dialog telling the user that the name is already taken
- A name that is invalid (over 40 characters in length or contains whitespace)
    - results in an error dialog opening up telling us that the name is invalid

No matter what wacky chatnames are, the list of chatrooms should update properly, display them in lexographical order, and a new chat tab for that room should open. If I double click on a chatroom that I'm already in, that should open up a dialog saying that I'm already in the chatroom. We then check each of the tabs to make sure that they have the correct chatname and correct userlist (just me). We then try closing one of the tabs. Because there is only one person in the chatroom (just me again), closing the tab will result in the chatroom being removed from the list of chats. If I then 'make' a chatroom with the exact same name, that will result in the tab being opened. I should still see any messages sent inside of that chatroom before I left.

Next we try sending a bunch of messages inside of a chatroom. Even though the room only has one person, the text area should still update with all the messages in a timely manner. Now is a

pretty good time to make sure the History is working. We go to File and click on the Chat History option, which should open up a new History tab. The history tab should have a list of all chats ever, including chats that have already been left. Selecting that chat from a tab should cause the main text area to display everything that has been said in a chat up until the user left the chat. I then go to a chatroom I'm still in, send a message and see if the chatroom in the history updates that room with that message. I then join yet another chatroom and see if the history updates that chatroom too.

Now I get one of my partners to come and join the chatroom too. If they try to join with my username, they should receive an error saying the username already exists. Once they join, on my end the list of users should update to include my partner. On their end, they should see the exact same list of rooms that I see, and they should see an updated user list. Now they make a room, which should cause the room list to update for both of us. Then they join a room that I am already in. On my end, I should see the user list update with their name. On their end, they should see a user list with both of our names. They should not, however, see any previous messages I may have sent before they joined the room.

The two of us will then chat back and forth for a while. Chats Anything said will have the format **username:** message. If the username is my own, it will be colored in blue. I will then leave the chatroom. This should result in the chat list on their side updating to reflect my leaving. The chatroom should still be in the list of chatrooms in the main window though. They will then continue sending messages inside of the empty room, and I will go check my history. My history should only contain any messages sent up until I left the room. I will then go rejoin the room. I will still only see any messages sent while I was in the room, and not any messages from the period in between.

Now my partner will close the application entirely. They will do so by selecting logout from the pull down menu. This will result in the userlist in the main room being updated to reflect their leaving, the userlists in each of the rooms they are in updating to reflect their leaving and the list in any rooms they are a member of updating. Any rooms they were the only member of should be removed. They then log in with the exact same user name and check their history. It should be empty. Then they logout by clicking on the close button on the window, and I log out by closing the application.

Finally we shut down the server to see if that closes all the windows. It does. Yay.