Each Client communicates with the Server using a unique Socket, ClientConnection and ServerConnection. Clients logging into the Server provide a username and are assigned a userID; a User object is then created and stored onto the Server to represent each logged-in Client. A channel has a unique title and is started by a User; any other User can join the channel by adding themselves to the list of its members (issuing a *join* command). A channel cannot be ‘killed’, even if has no members. The client sends commands to the server through an instance of ClientConnection and an instance of ServerConnection. They handle the Sockets at either end of the connection between client and server. Each constantly reads in input to process, and has a queue for writing.

GUI Summary

In the GUI, we have a few major components. There is a roomList of rooms the user is currently in, there is a chatList that represents the messages being sent in the user’s active room, there is the userList of users currently in the active room, and there is a Tree of every user logged in to the system.
There are two text boxes: one for the user to type in a room to join, and one for a user to type in a message.

Below are the major methods of each of our main classes.

public class Client

The client runs on the client’s computer.

```
    //Main GUI for this Client.
    private MainApp gui;
    //List of Rooms the Client currently in
    protected List<String> currentRooms;
    //HashMap of room names to the rooms’ display messages
    protected ConcurrentHashMap<String, List<String>> roomMessages;
    //ClientConnection used by this Client to manage its Socket
    protected ClientConnection conn;
    //this Client’s username
    private String user;
```

```
	//Sign in GUI for this Client.
	private Signin signin;

	public Client();

	//Attempt to login to the server.
	public void login();
```

public class Connection

A superclass of ClientConnection and ServerConnection; handles a Socket and all Client-Server communication across it. It has two Runnables, ConnectionReader and ConnectionWriter to read in and write out Packets, and a BlockingQueue of Packets to send across the socket.

```
	protected BlockingQueue<Packet> messageQueue;
	protected Socket socket;
	protected User user;
	protected Thread readerThread;
	protected Thread writerThread;

	//Creates a Connection instance to handle communication across this
socket.
	public Connection(Socket socket)

	//Reads in Packets from the incoming stream and processes them.
	public class ConnectionReader implements Runnable


	//Takes Packets from the messageQueue to write out.
	public class ConnectionWriter implements Runnable


	//Offers a Packet to the messageQueue
	public void sendMessage(Packet output)
```

```
//Interrupts the current writer thread
public void closeConnection();

// Use the type of the Packet to determine the appropriate action.
public void processMessage(Packet Packet);

//Interrupts the current thread
public User getUser();

//set the current user
public void setUser(User user);
```

public class ClientConnection extends Connection

A subclass of Connection; handles Client-Server communications at the Socket on the Client's end. Overrides Connection methods such as processMessage() to call Client's methods.

```
      private Client client;
      private MainApp gui;
      public ClientConnection (Socket socket, Client client)
      public void setGUI(MainApp gui)
      public List<String> getMessages(String room) //get messages in this room
      public boolean roomExists(String room) //see if client is in this room
      public void join(String room) //attempt to join this room
      public void login(String username) //attempt to log in with username
      public void message(String message, String room) //send a message to 
this room
      public void listUsers() //send request for a list of online users
      public String getUsername() //get Client's username
      public List<String> getRoomMessages(String roomName) //get messages from 
this room
```

public class ServerConnection extends Connection

A subclass of Connection. Overrides Connection's processMessage() method to handle incoming communication (commands) from the client by delegating any actions to be performed to the running Server instance. Each user has a single ServerConnection that handles server-side socket input/output relating to that particular user.

```
    private Server server;
    private int userId;
    public void processUserDisconnect()  //deal with a Client disconnecting
    public void closeSockets() //close down Connection by stopping threads
and closing the socket.
```

public class Channel

This class represents a channel in guichat, and stores a list of users and a list of Packets. The channel has functions to process client commands such as "join #channel" and "leave #channel". An instance of Channel is created when at least one person joins the Channel, and then removed when it has no more members.

```
    private String name;
    private User owner;
    private List<User> users;
    private List<Packet> messages;

    //Construct a Channel with name channelName, started by User owner
    public Channel(String channelName, User owner);

    //Returns true if this Channel has User u as one of its current members;
    //false otherwise
    public boolean hasUser(User u);

    //Add User u to this Channel's list of members
    public void addUser(User u);

    //Remove User u from this Channel's list of members
```

```
	public void removeUser(User u);

	//Send a Packet to the group by adding Packet m to this Channel’s
	//List of Packetsalerts all Users in this Channel
	//other than author of this addition
	public void addMessage(Packet m, User u);

	//Returns the users in this Channel
	public List<User> getUserCount();

	//Returns the number of users in this Channel.
	public int getUsers();

	//Returns String representation of all the Packets in this Channel.
	public String getMessages();

	//Returns String of all the nicknames of the Users in this Channel.
	public String getUserNames();
```

public class Server

The chat server. Stores a HashMap linking Channel names to Channels, a HashMap of nicknames and their respective User classes, and a HashMap of Users to the Thread that processes them. Also has functions to carry out User commands.

```
	private ServerSocket server;
	private HashMap<int, User> userMap;
	private HashMap<String, Channel> ChannelMap;

	// Instantiate a server on the specified port.  Create Server fields of
	// a ServerSocket, a HashMap<String, User> and a HashMap<String,
Channel>
public Server(int port);

// Listen for connections on the specified port; upon accepting create a
ServerConnection to handle the Socket and a User to represent the
logged-in Client.
public void listen();

// Process a user “joining” a Channel
public void addUserToChannel(String nickname, String channelName);

// Process a user “leaving” a Channel
public void removeUserFromChannel(int userID, String channelName);

// Process a user creating a Channel, returns that new Channel
public Channel createChannel(int userID, String channelName);

// Return a Channel on the Server.
public Channel getChannel(String channelName);

// Return a User on the Server.
public User getUser(int userID);

// Get a String representation of all Users on the server.
public String getUserList();

// Get a String representation of all Channel names on the server.
public String getChannelList();

//Send a Packet to a specified Channel
```

```
	public void sendMessageToChannel(int userID, Packet message);

	//Get String representation of all Users on a Channel
	public void getChannelUsers(String channelName);

	//Get String representation of all Packets on a Channel
	public void getChannelMessages(String channelName);

	//Remove a User that has disconnected from the Server
	public void processUserDisconnect(int userID);

	//Closes the Server.
	public void terminate()
```

public class User

A class representing a user

```
	public int id;
	public String nickname;
	public Connection connection;

	// Create a new User given a server, socket, and nickname
	public User(String nickname, int id, Connection connection);
```

public class Packet

A Packet is a discrete unit of communication between the client and the server. Packets are sent through a socket.

```
	private Command command;
	private String channelName;
	private Calendar calendar;
```

```
  private String messageText;
  private String author;

 // Create a new Packet given a command, Channelname, date, author and String messageText
 public Packet(Command command, String channelName, Calendar calendar, String messageText, String author);

  // Returns the Command type of this Packet
  public Command getCommand();

  // Returns the text contents of this Packet
 public String getMessageText();

  // Returns the date that this Packet was sent
  public Calendar getCalendar();

  // Returns the sender of this Packet
  public User getSender();

  // Returns the Channel of this Packet
  public Channel getChannel();
```

public class MainApp

GUI for the main IM chat view.

```
    private JFrame frame;
    private JTextField roomText;
    private JTextField type;
    private JTextField SigninText;
    private ClientConnection conn;
    public JTable roomTable;
    public JList chatList;
    public JLabel roomLabel;
    public JList userList;
    public JTree tree;
//Launch the application
public void init()
//Initialize contents of the frame
private void initialize()
```

public class Signin

GUI for the Login page that opens when a Client is initialized to log into the chat.

public class SigninListener implements ActionListener

Listener that listens to for a Client to sign into the server.

public class ButtonRenderer extends JButton implements TableCellRenderer

Represents a Button on our GUI.

public class RoomTextListener implements KeyListener

Listener that listens to the text box where users type in the name of the room they wish to join.

public class TypeListener implements KeyListener

Listener that listens to the text box where users type to send a message in a chat room.

Client-Server Protocol

Shared grammar:

WORD ::= [A-Za-z0-9_]+

SPACE := " "

NEWLINE := "\n"

COMMA ::= ","

PERIOD := "."

NAME := [WORD]+

nickname = CHANNELNAME = AUTHOR = NAME

ID := [0-9]+

LINE := [WORD SPACE COMMA PERIOD]*

TEXTLINE := [LINE]* NEWLINE

<u>Client --> Server</u>

protocol ::= command*
command ::= (login | logout | list-channels | list-users | user-channels | join | leave | post)
NEWLINE
login := "login" nickname //logs in a user with that nickname
logout := "logout" //logs the current user out
list-channels := "list-channels" //list all open channels (channels with >0 users in them)
list-users := "list-users" //list all users currently online
user-channels := "user-channels" //list the channels the user is in

join := "join" CHANNELNAME
            //have user join the channel with name CHANNELNAME

leave := "leave" CHANNELNAME?
            //have user with name nickname leave the channel with name CHANNELNAME
            //If no channel argument supplied, user leaves current channel

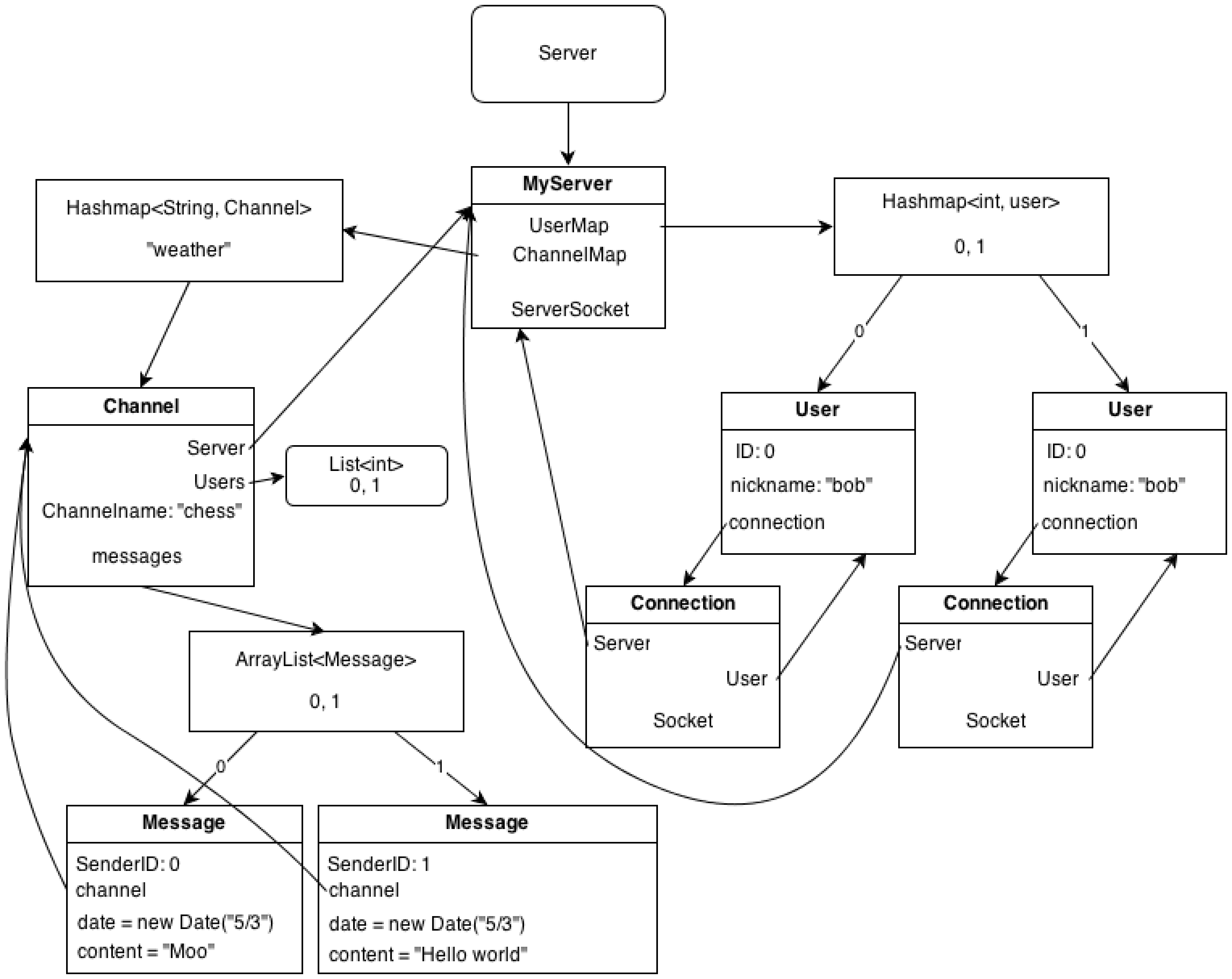
Message := "message" CHANNELNAME TEXTLINE
// have user with name nickname post a Message comprising of TEXTLINE in the channel with name CHANNELNAME

<u>Server --> Client</u>
protocol ::= command*
command := success | failure | error | reply-list-channels | reply-list-users
success := "reply-success"
failure := "reply-failure" TEXTLINE* // TODO: have TEXTLINE contain helpful info about failure
error := "error" TEXTLINE // When server throws Exceptions

```
reply-list-channels := "reply-list-channels" CHANNELNAME*
        //info Message sent in response to a list-channels or user-channels request
reply-list-users := "reply-list-users" nickname*
        //info Message sent in response to a list-users request

Message := "message" CHANNELNAME AUTHOR TEXTLINE
```

Snapshot Diagram

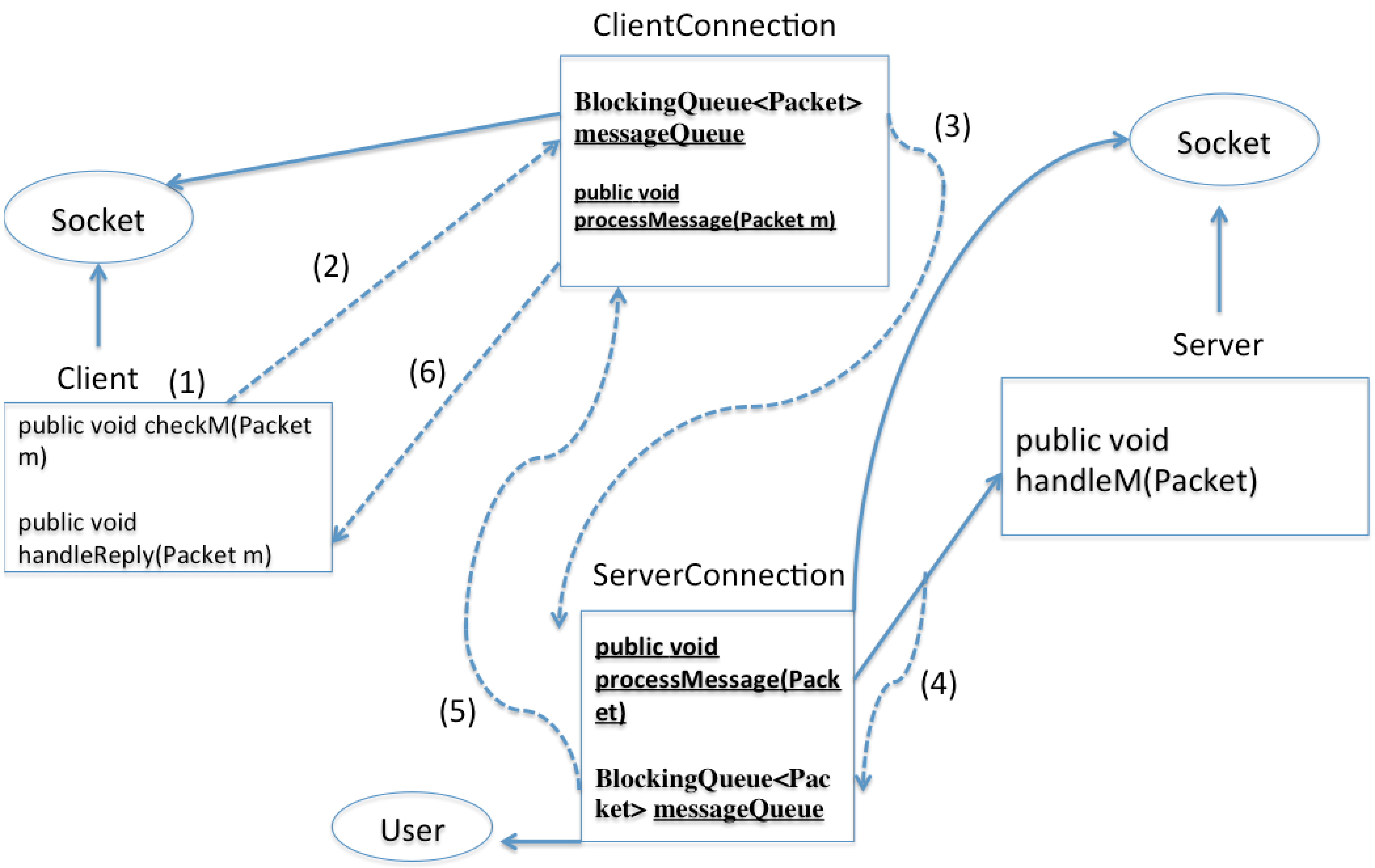


** in this diagram, we treat the Connection as a 'black box'. See below in-depth diagram for details on what the Connection box is actually representing.

In-depth Snapshot Diagram

The Inner Workings of the “Connection” Box from the Above Diagram

(the steps of a Client request, represented by Packet m)


ClientConnection
BlockingQueue<Packet> messageQueue
public void processMessage(Packet m)
Socket
(2)
(3)
Socket
(6)
Server
Client
(1)
public void checkM(Packet m)
public void handleReply(Packet m)
public void handleM(Packet)
ServerConnection
public void processMessage(Packet)
BlockingQueue<Packet> messageQueue
(4)
(5)
User

(1) The Packet m is checked for proper protocol grammar using Client’s checkM() method.
(2) If properly formatted, m is sent to the Client’s ClientConnection’s queue of Packets waiting to be sent across the socket connection.
(3) m is taken from the queue in ClientConnection and sent to the ServerConnection at the ‘other end’ of the socket connection.
(4) Immediately after arriving at the ServerConnection, m is processed using ServerConnection’s processMessage method which calls some method in Server handleM. processMessage may produce a Packet r, representing the Server’s response to m. r is now put into the ServerConnection’s queue.
(5) r is taken from the ServerConnection’s queue and sent back to ClientConnection for

processing.
(6) Immediately after arriving at the ClientConnection, r is processed using the processMessage method in ClientConnection which calls on some handleReply() method in the Client.

Concurrency Strategy

Each connection (Socket) made between a Client and a Server is managed by a ClientConnection and a ServerConnection; both are subclasses of the Connection class. Both ClientConnection and ServerConnection are constantly waiting for incoming Packets, which they then immediately process. Thus, each incoming Packet is processed one at a time (in the same Thread), so we do not have to worry about concurrency issues there. Each also has a blocking queue (the “writing” queue, consisting of Packets to be written to the Socket output stream); the elements of the queue are popped off and written using a separate Thread.

We run into problems with concurrency when two requests from different Clients (through separate Connections) both attempt to mutate the same object on the Server. Since we only mutate objects on the Server by calling Server’s methods, we will make our Server threadsafe by synchronizing methods which mutate Server objects (or alternatively, using a BlockingQueue to store commands to be executed by the Server). Then only one request may mutate the Server’s objects at a time. A bottleneck may then be formed but this is acceptable.

On the GUI side, we maintain concurrency in our Listener objects by using locked objects in a global order.

Testing Strategy

Automated Testing

Our automated testing will focus on ensuring our back-end modules, including the Server, the Client, and the Connection in addition to our Model classes (such as Packet, Channel) function as expected. We will usually do so by using a dummy server, then simulating a client connecting to this dummy server and processing different actions.

We will first test the methods that the Server and Client use to respond to Packets. For example, in the Server we will use JUnit tests to test logging in/out clients; creating/deleting channels; adding a client to/deleting a client from a channel; posting Packets; getting a channel’s history; seeing other clients and channels, all of which may or may not exist when the method is called.

Then, we test that upon a single Client logging in, followed by one or more Clients logging in, the created Client Connection and ServerConnection instances successfully pass Packets in-between them. For this step we may simply have the Client send along dummy Packets, which the Server will respond to with another dummy Packet, which the Client will finally print out. We will test for correct Packets and behavior given interaction between users, such as having one user trying to respond to a conversation that another starts. We may also check the order in which Packets are processed.

Next we test to ensure that ClientConnection and ServerConnection will always pass properly formatted Packet instances to each other; therefore, there will be no problems when it comes time to process the Packets. This checking will occur in our Client, Server, ClientConnection or ServerConnection classes. In general, if a client inputs a request not adhering to the grammar, we will treat it as an undocumented case and unexpected behavior might occur.

Finally, as an integrated test we will simulate a lengthy chat session (by manually forcing certain Packets into the Connection), involving at least two Clients who log in at different times, but always making protocol grammar-adherent requests. At specified points in this session we check the fields of the Server and the Client to ensure everything is being processed correctly. Eventually, we want to test all commands in our Client-Server protocol, and ensure that the Packets sent back to the client from the server are correct.

Manual testing step-by-step

- Logging in
    - Port has to be positive integer
    - Username can't have spaces
- Main App
    - Sign in populates all users
        - for every user
    - Typing in room name
        - Name cannot have spaces, cannot be repeats
        - Proper name adds to room list, sends join message to server
    - Clicking join button:
        - changes room label
        - populates user list
        - loads chat history (since last login if applicable)
    - Clicking leave button:
        - removes room from room list

        - clears messages from stored history
        - clears chat window
        - resets room label
        - resets user list
    - Typing:
        - asks to join a room if not in room
        - adds message to chat window
        - adds message to chat history
        - message appears to other users' chat window

<u>Testing Report</u>

In the end, many classes whose functionality we intended to test with automated testing (including ServerConnection and ClientConnection) we decided to manually test. This was largely because we experienced difficulty working with Sockets in JUnit testing.

In our manual testing, we first tested our login methods, then moved onto joining/leaving rooms, and sending messages. We found that our ServerConnection and ClientConnection's reading and writing methods worked perfectly for all the above cases; upon ServerConnection receiving a message the processing's response message would be correctly received by the ClientConnection to be processed. Within our GUI we would always be sending correctly formatted Packets through the sockets, and even with multiple clients all participating in a conversation the chat GUI and back-end held.

However, in order to keep up GUI functionality we found that our server-end processing methods needed some work. Sometimes this was due to unintentional bugs (for example, due to the loss of one bracket, an entire section of code was unnecessarily repeated). Other times we found that the GUI required information requiring a different type of Packet than what we originally intended the process to send, or required us to send additional information to other logged-in clients.

Next we focused our manual testing on the GUI itself. We made sure (using print statements as well as server object checking statements) that the SigninListener ensured that only properly formatted names and port numbers would be allowed for logging into the Server. Specifically, the names could not contain spaces and the port number had to be positive.