

6.005 Project 2: ChatTrollette Final Design Document and Testing Report

Team Members: Rebecca Zhang (ryzhang), Katie Lee (katielee), Birkan Uzun (birkanu)

TA: Joshua Oremán

Conversation Design

Definition of Conversation: Communication of text between two or more clients on the server.

Our Instant Messaging System will allow a conversation between two or more people in the chatroom. All clients automatically join the a main chat room (“Lobby”) when connected to the server and can see all other existing clients and send messages to all other clients through this main chatroom. Each client can also choose to have 1-on-1 conversations with another online client. A conversation is more like e-mail than like a phone call-- every client can send messages to other online clients, and all other clients will see these messages, but it is up to the other client to decide whether he or she wants to respond with their own message.

Timeline:

1. Server starts with given host and port.
2. When LoginWindow starts, user can type in host and port, and username, to join the chatroom.
3. When the submit button is pressed, an instance of Client is created that connects to the Server and starts listening to the Server’s messages. An instance of clientThread is created as a one-to-one server thread to handle the Client’s requests. LoginWindow disappears.
4. MainWindow is opened to the Lobby section, which allows the user to start typing into the main chatroom of all online users by pressing the “send” button.
 - 4a. The user can select one online user to chat with privately by double-clicking on the user’s name on the right-hand side.
 - 4b. The user can navigate to the chat histories he/she has with all previous users by clicking on the History tab and clicking on the user’s name on the right-hand side.
5. User can disconnect by pressing the “disconnect” button.
 - 5a. If so, then the Client relays the disconnect request to the Server and closes the connection.

Note: To be able to see the chat program running, one has to run the Server.java file to wait for user connections, and then the LoginWindow.java to be able to login to the Chatroom.

Chat History:

Chat conversations are recorded by a private HashMap stored in each clientThread. The keys are

String usernames of the other Clients they are chatting with, and the values are a String instance of the messages they have sent to each other so far. We update and store the chat history by appending each message that is sent to the String of the corresponding HashMap value.

Log Out & History:

When users log out of the chatroom, their name disappears from the list of online users who are available to chat, and they lose access to their chat histories. However, the users who have had conversations with them and are still connected to the server can still view their chat histories with that particular user.

If a user logs out of the chatroom, any user who was chatting individually with them can still type messages into the conversation, but the user who logged out (disconnects from the server) will no longer be able to see those messages. When that user logs back in with the same username, the two users can resume their conversation. However, the user who just logged back in will have a clean-slate history and will no longer have records of the past conversation history. On the other hand, for the user who has been logged in this whole time, the new conversation initiated with the user who just logged back in will be added to the chat history from the previous conversation (given the user who logged back in logs in with the same username).

Special Feature : Trolling with Music:

During a conversation, if specific keywords or phrases (see below) are contained in the message that is being sent, our chat will play a short 2-3 second sound effect to the sender and all recipients of that message.

We understand that currently our music is playing within `invokeLater()` so any following messages won't show up in the users' chat conversation until the music has finished playing. We considered having the music play separately in its own thread and blocking queue but as a design decision, we wanted it to appear right after the corresponding messages that triggered the music (since that's part of the "trolling" effect), instead of having the messages decoupled from the music and show up as soon as they are sent and potentially having a growing queue of music to play.

- Keywords:

nyancat, sexy, awkward, groovy, rickroll, synchronize, noooo, 6.005, yesss, mastah, master, pokemon, number, debug, help.

**we considered variations of the keywords that make the most sense.

ex. -it is difficult to accidentally type nyancat, synchronize, and pokemon, so any appearance of that text triggers

-noooo and yesss are components of common words, so they must exist exclusively

-some phrases are commonly typed with extra letters or no spaces, ex. sexy → sexxxxxy, nyancat → nyan cat, etc. so

we allow for some flexibility there

-only when you end a phrase with number will we call you maybe.

Overview of Classes

main Package:

- **<class> Server**
 - Handles connections for and communication between multiple clients
 - Variables:
 - clientThreads: ArrayList of clientThread instances that keeps track of current clients that are connected to the server. Each clientThread is the corresponding server thread that handles that client's requests.
 - Public Methods:
 - serve(): Runs the server, listening for client connections and handling them by creating an instance of a clientThread. Never returns unless an exception is thrown that the main server socket is broken.
- **<class> clientThread**
 - Individual thread of the server that handles that specific client's requests.
 - clientThread takes in the socket, port, and clientThreads list.
 - Thread responsible for client involved broadcasts to every other client.
 - LoginWindow pops up after user connects to Server's port.
 - clientThread is started before MainWindow.
 - clientThread has its own username associated with it.
- **<class> Client**
 - Creates a clientThread, but is NOT a thread.
 - Chooses an individual user alias to take for the chat time.
 - Works with a publisher class to publish events to the GUI.
 - Public Methods:
 - startDialog():
 - Opens a new Socket and connects to the ServerSocket (given the host and port).
 - Creates an output (ObjectOutputStream) to write to the Server and input (ObjectInputStream) to read from the Server
 - Starts the Publisher which announces Events to the GUI
 - closeConnection():
 - closes the input, output, and clientSocket
 - getClientSocket():
 - returns the clientSocket of the Client
 - getPublisher():
 - returns the Publisher of the Client

pubsub Package:

- **<class> Publisher**
 - Publishes communication to listeners (we will be using it to publish from Client to GUI)
 - Methods:
 - **addInputStream(BufferedReader inputStream)**
 - defines the input stream from which the publisher reads its messages from
 - **addListener(Listener listener)**
 - adds listener to ArrayList of all the listeners of the Server
 - **announce(String event):**
 - broadcasts the event to all listeners of the publisher
 - **run():**
 - Listens for communication from inputStream (sees server communication to client)
 - Publishes inputStream messages to the listeners (sends messages from client to GUI)
- **<class> Event**
 - has enum Types {AddUser, RemoveUser, Message, AllUsers}
 - Public Methods:
 - **getType():** returns the type of event
 - **getMessage():** returns the message
 - **getThirdToken():** returns third token of the message (useful for parsing messages as defined by grammar)
 - **getSendTo():** returns ArrayList of all users that this message is being sent to
 - **StringToType():** turns a String into a type
 - **StringToEvent():** turns a String into an Event
 - **toString():** turns an Event into a String
- **<interface> Listener**
 - Implements event listener
 - Implemented by the MainWindow and ChatWindow classes
 - required methods:
 - **event(String serverResponse)**

gui Package:

- **<class> LoginWindow**
 - Created when the program begins
 - User must connect to valid host and port with a username that is not already taken.
 - ActionListener
 - listens for Start Button to be pressed.
 - if inputs are valid, then an instance of Client is created, the MainWindow is created, and the LoginWindow is closed.
 - if inputs are invalid, then an error message pops up.

- **<class> MainWindow**
 - Creates Window for a conversation
 - Automatically creates tabs for the Lobby (which has the groupchat + all online users) as well as the History
 - Implements the Listener class in order to read events from the Client
 - Implements ActionListener in order to respond to actions in MainWindow
 - Lobby Tab:
 - Displays text exchange that is meant for all clients
 - Shows list of all clients connected to the server
 - Can send messages to all clients for other clients' Lobby tabs
 - Option to disconnect from ChatTrollette
 - History Tab:
 - Displays history of conversations for all clients who are/were online at some point during the server connection session
 - Shows list of all clients who were/are connected to the server since the beginning of when you were connected to the server
 - addChat(String username) → adds tab for a 1:1 chat with an online user.
 - ActionListener
 - send Text (either by pressing Enter in the sendbox or pushing the “submit” button) - writes Event to the Server
 - disconnect - writes to Server saying this user has requested to log out
 - select online user - begin 1:1 chat with another user
 - select chat history user - see chat history with (previously) online user
 - Event method from implementing the Listener class:
 - Parses messages from the Client to do associated actions
 - Only method that manipulates GUI after creation
 - SwingUtilities.invokeLater to ensure thread safety for GUI manipulations

- **<class> ChatWindow**
 - JPanel for a conversation between two users
 - Implements the Listener class in order to read events from the Client
 - Implements ActionListener in order to respond to changes in ChatWindow

- Constructor takes in client instance and string of username to communicate with
- Methods:
 - actionPerformed (ActionEvent e) → listens for actions in the GUI
 - ChatThisWindow (String message) → adds chat text to this window.
 - SwingUtilities.invokeLater to ensure thread safety for GUI manipulations
 - event (String response) → listens to publisher with corresponding client instance, parses messages to do associated actions
 - SwingUtilities.invokeLater to ensure thread safety for GUI manipulations
 - getHistory() → returns text of the conversation that took place within this ChatWindow
- **<class> ButtonTabComponent**
 - Window for a conversation between two or more users
 - Methods:
 - closeTab(String tabName) → closes tab with the given name
 - getAllTabs() → returns all existing tabs of the pane
 - addClose() → adds a close button to the tab
 - TabButton class that makes a close button for a tab

funstuff Package:

- Various .wav files played by the Music.java class.
- Different icons (.png files) for the LoginWindow.
- **<class> Music**
 - Constructed with a keyword, which then plays corresponding .wav file (it can only play .wav files)
 - Public Methods:
 - keywordIn(String message) → static method that returns the keyword within the message. If there is no keyword in the message, returns null.
 - playSound(String filename) → takes in a filename which is the name of the file that is going to be played and plays it.

Concurrency Strategy and Thread-Safety Argument:

Concurrency on the Server Side:

The Server's serve() method has a while loop that loops forever unless the main server socket is broken,
 listening for client connection requests on a ServerSocket. When a request comes in, it accepts the

connection, starts a new clientThread for that connection which takes in the socket returned from accept as well as the clientThreads list (which is a list of all connected clients), and starts the thread. Then the server goes back to listening for connection requests. The clientThread.java class communicates with the client by reading from and writing to the socket and handling all the requests. So, multiple clients can connect and disconnect to the Server in a thread-safe environment.

Inside the clientThread class, any code written in synchronized block in java will be mutually exclusive and can only be executed by one thread at a time. Therefore, by using synchronization on getUsername(String possibleUsername), broadcastNewUser(), updateUserList(), broadcast(String message, ArrayList<String> user), broadcastUserLogout() and removeUser() methods to lock the whole clientThreads list. So, when the clients requests something from the server (like typing a text, joining the chat, disconnecting etc.) and these methods are called, only one method can be executed at a time. This means that, when one client is using a method, that method is locks the list of other clients and they can't use it. Once a client is done with the method, the other clients can now execute it. Since synchronization is not an expensive process, this doesn't slow down the Chatting process at all.

Some facts:

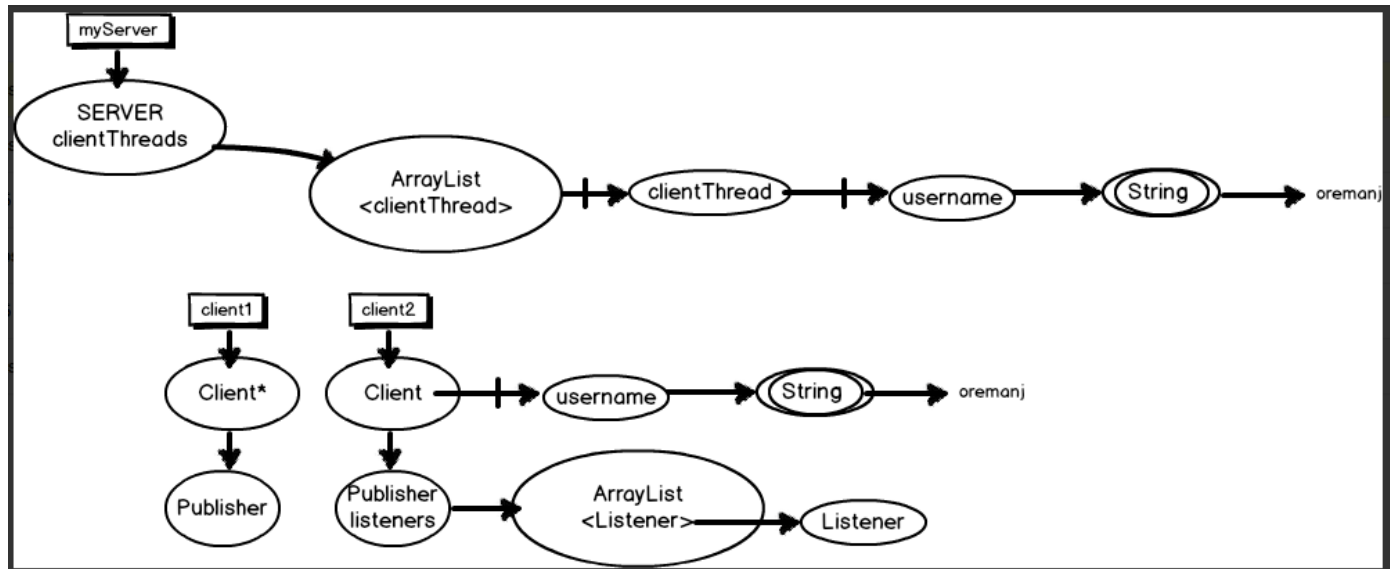
- Multiple users creates an instance of Client with a unique username.
 - the *checkUsername*(String username) method in clientThread checks the list of all existing usernames to make sure that a client with the given name is not already taken by another client already connected to the server.
 - *checkUsername* is the method that creates an instance of Client.
- Multiple clients connect to the server concurrently.
 - Each client connects to the server on its own thread of the server.
 - Server keeps listening for new connections, and when there is a new connection, it starts a new clientThread for the specific client and handles its requests concurrently.
- Multiple messages are sent to the server concurrently.
 - When we call the broadcast method to deliver a message to all users or to a particular user, we lock the clientThreads so that the server can only receive messages from one person at a time.
- Multiple messages are sent to a client from the server concurrently.
 - server (clientThread) only sends one message at a time to a client because all broadcast methods from server to client are synchronized to each instance of clientThreads.
- All other requests (such as logging in,disconnecting) are distributed to each client concurrently as the methods for handling these requests are synchronized.

Concurrency on the Client Side within the GUI:

All methods that update the GUI (i.e. all the Listener events) use SwingUtilities.invokeLater such that only one thing is written to the GUI at a given time. This applies to the MainWindow and ChatWindow

classes because they are the only ones with clashing action events.

Snapshot Diagram:



Client-to-Server & Server-to-Client Protocol:

The Client and Server classes need to communicate in order to log into a specific User and to send messages within a Chat. We have specified protocols to allow for the following:

- a new client connecting to the server
- a client disconnecting from the server
- exchanging messages between users

Note that this protocol works both ways, so we didn't specify separate Client-to-Server and Server-to-Client protocols.

GRAMMAR:

MESSAGE ::= (ADDUSER | REMOVEUSER | MESSAGESEND | ALLUSERS) NEWLINE

ADDUSER ::= "AddUser" SPACE USERNAME SPACE "has joined the chatroom."

REMOVEUSER ::= "RemoveUser" SPACE USERNAME SPACE "has left the chatroom."

MESSAGESEND ::= "Message" SPACE USERNAME (DASH USERNAME)* TEXT

ALLUSERS ::= "AllUsers" SPACE USERNAME (SPACE USERNAME)*

TEXT ::= .+

USERNAME ::= [a-zA-Z0-9]

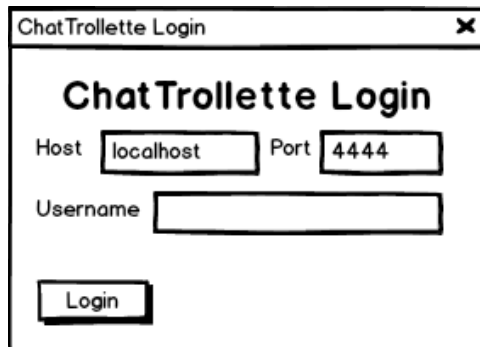
DASH ::= "-"

SPACE ::= \s

NEWLINE ::= "\r?\n"

UI Sketches:

Initial LoginWindow:



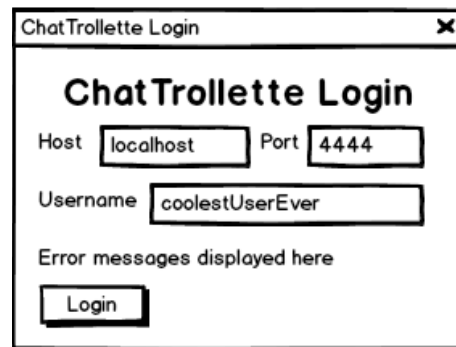
ChatTrollette Login

ChatTrollette Login

Host Port

Username

Error for LoginWindow:



ChatTrollette Login

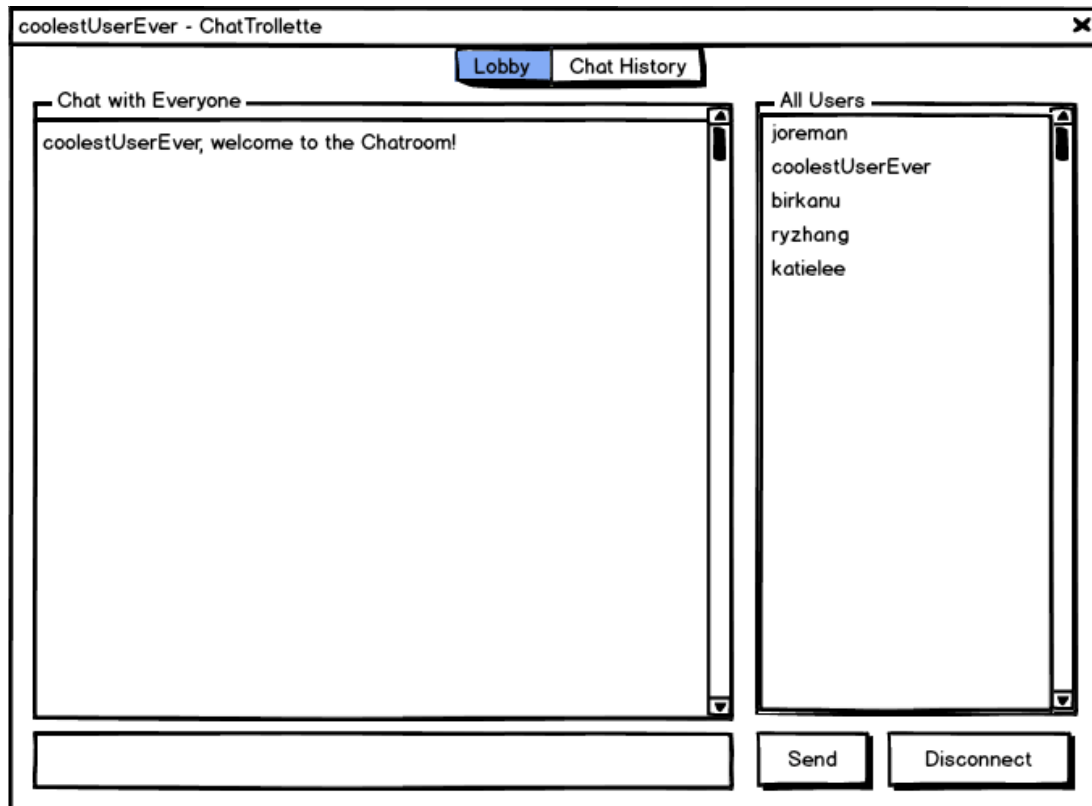
ChatTrollette Login

Host Port

Username

Error messages displayed here

Lobby Tab of MainWindow (initial successful login):



coolestUserEver - ChatTrollette

Lobby Chat History

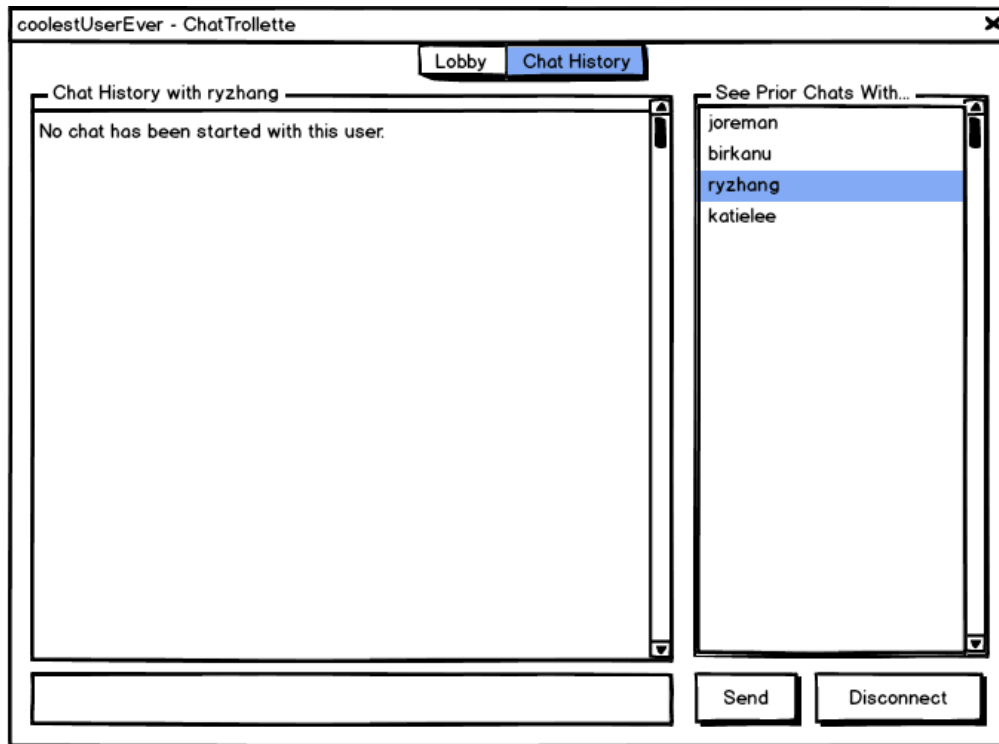
Chat with Everyone

coolestUserEver, welcome to the Chatroom!

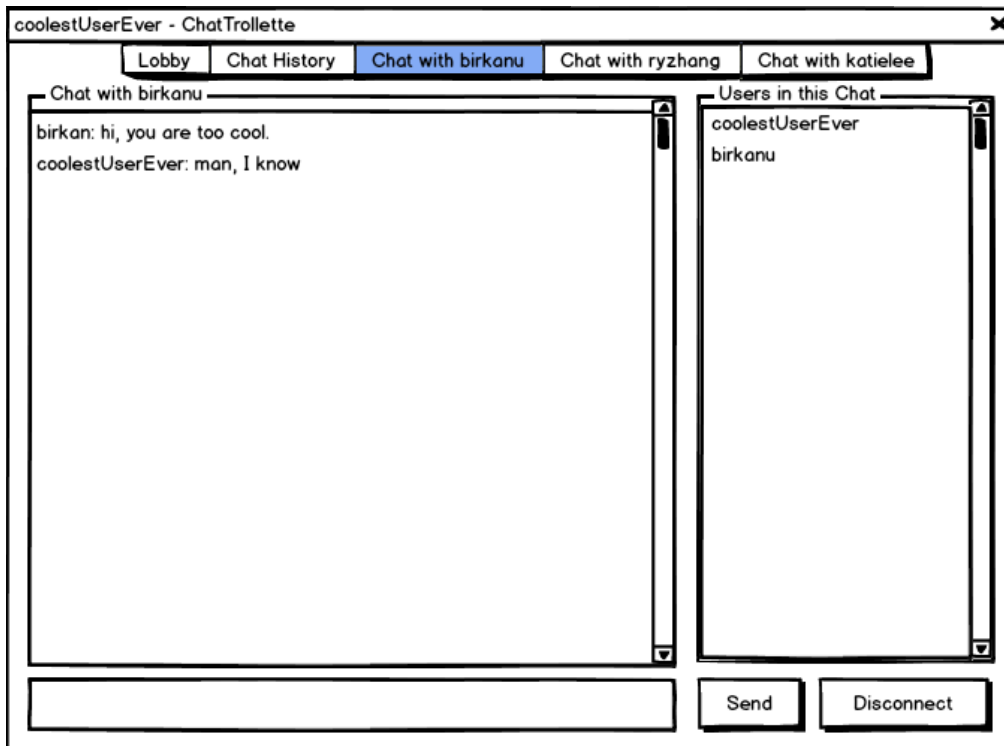
All Users

- joreman
- coolestUserEver
- birkanu
- ryzhang
- katielee

History Tab of MainWindow:



ChatWindow Tab Instance of MainWindow:



Testing Strategy

Overview of Testing Strategy:

We will have three types of tests: automated tests using JUnit, manual tests using telnet, and manual tests using the GUI.

JUnit tests allow us to conduct precise, repeatable unit and integration tests on our non-GUI components. We decided to focus on performing integration tests to see the interaction between our Server, Client, clientThread classes. Theoretically, creating unit tests for our Server for example would use either sockets or JMock objects to isolate the performance of the Server but the tasks it performs are simple (accepts a client socket and creates a corresponding clientThread) and wouldn't need such an elaborate test suite. (We wrote a Unit Testing protocol anyways to demonstrate what we would have tested on. This is labeled as "What we would have done" and is not actually implemented. However, we did write Unit tests for the Event class.) Our manual tests cover the testing that the unit testing of our non-GUI components would have covered; if our manual tests pass, our unit tests would have passed as well. For the integration tests we did write for the JUnit automated testing, we have 5 tests as mentioned below under integration testing.

With our first type of manual testing, telnet serves as the client that connects to our server so that we can do integration tests stripped of the GUI. We thought this would be an efficient way to cover a large area of the integration of our non-GUI components.

We use manual tests through the GUI and automated JUnit tests to test the various parts of our project. The manual tests allow us to test the concurrency of our chat and evaluate the user experience of the GUI and navigate through the interactions of our classes with ease.

Coverage Level:

Through the total efforts of our three types of testing, we aim to test every class (and every line of code) and test them in varying conditions like with multiple clients and multiple requests per client. The GUI should behave the same (not throw unnecessary exceptions) when the traffic of requests is heavier.

Automated Tests: JUnit Testing

We will do JUnit testing on the non-GUI components (i.e. Server.java, Client.java, clientThread.java). Mainly, we will test the communication between the Client and Server classes by creating a socket and sending messages to the server directly.

- Unit Testing:

- [What we would have done]
 - Server
 - serve() method: run the server and make sure that it can receive a socket
 - ClientThread
 - setUsernameTaken() method: to make sure that the clientThread returns the correct boolean *taken* if a user requests a username
 - removeUser() method: to make sure that clientThreads correctly has the removed user's clientThread removed
- [What we did implement]
 - Event
 - tests all public methods of the Event class

- Integration Testing [IntegrationTest.java]

- These tests test the Server and clientThread by having sockets connect to the Server and attempt to assume usernames. By running them, we test clientThread's newUser(), setUsernameTaken(), broadcastNewUser(), updateNewUserList(), broadcast() methods.
 - test handling one connection from a client socket to the server:
basicOneSocketName1Test(), basicOneSocketName2Test()
 - test that server notifies other client sockets when a new user has joined:
basicTwoSocketTest()
 - test that server notifies client socket if a username is taken:
basicTwoSocketSameNameTest()
 - test that server can handle three clients connecting: basicThreeSocketTest()
- **Note:** These tests cannot be run on didit because didit does not run network connections. We tagged the test file with @category no_didit. To automatically run the tests, please run the IntegrationTest.java file in the main package. Then the five tests in the suite should run.

Manual Testing: To test integration of non-GUI components using telnet

After Design Milestone 1, we performed manual tests using telnet to test the interaction between

the server, clientThread, and client. At that time, we had implemented one big chatroom, so when a client successfully connects the Server, they join the chatroom and can type in messages that would be received by all other online clients.

We ran the Server.java class and opened up terminal windows to connect to it. Messages would be visible on the terminal through the PrintWriter and BufferedReader in the clientThread class and calling out.println().

>> Test 1: Test connecting to the server

We tried to connect to the server by typing: "telnet localhost 4444" if we only had one client.

Then, after pressing "Enter," we could type in a username. If the connection worked, then the Server would return: "[username], welcome to the chatroom!"

>> Test 2: Multiple clients and valid usernames

We also tested multiple clients connecting to the server. We could open multiple terminal windows on the same computer and connect using "telnet localhost 4444" and also used multiple computers and connected via IP address: "telnet [IP Address] 4444".

This test also checked whether usernames were valid. If a user tried to type in a username that was already taken, the Server would recurse over the new user creation method and prompt the user to enter another name by showing another blank line for the user to type in. Again, if the user successfully joined the chat, the server would return: "[username], welcome to the chatroom!"

>> Test 3: Multiple clients and conversation

Once there were multiple users in the chatroom, we tested the chatting mechanisms. To send a chat, the user could type in the terminal and press "Enter". If the chat was successful, all other terminals connected the server would receive the message and display [username]: message. Again, we used multiple terminal windows on the same computer and also used multiple computers to make sure the chat was possible.

Note: Our manual tests with telnet and the terminal were not able to test individual chats since individual chats were not implemented until later on with the GUI suite.

Manual Testing: To test for interaction with GUI/integration/synchronization

We will do manual testing through the GUI components to test both the GUI and perform additional tests on the non-GUI components.

>> Test 1: to test initialization of a Client (testing all functions of the LoginWindow)

1. Run Server.java file to start a Server (with port 4444)
2. Run LoginWindow.java
 - a. Expectations:
 - LoginWindow pops up
 - host and port fields have default "localhost" and "4444" values
3. Check the submit button for various username entries:
 - a. Enter no username (whitespace only)
 - Expectation: error message "Username must be alphanumeric...", username field is cleared
 - b. Enter username with non-alphanumeric characters
 - Expectation: error message "Username must be alphanumeric...", username field is cleared
 - c. Enter username that is already taken
 - Expectation: error message: "Username already taken...", username field is cleared
 - d. Valid username & type Enter
 - Expectation: LoginWindow closes, MainWindow pops up
 - e. Valid username & press "Login"
 - Expectation: LoginWindow closes, MainWindow pops up

>>Test 2: to test MainWindow and ChatWindow (proper chatting, all features of the MainWindow and ChatWindow)

1. Run Server
2. Complete Test 1 steps with valid username "username1"
 - a. Expectations
 - Lobby Tab: "username1" is the only username that is within the lobby's OnlineUsers list (because this is the only client that is connected)
 - History Tab: there are no usernames in the history tab because there are no other users that we could have chatted with.
4. Complete Test 1 steps with new valid username "username2"
 - a. Expectations:
 - Lobby Tab: for both MainWindows, onlineUsers list has both usernames.
 - LobbyTab: print "username2 joined chat..." in textfield of "username1"'s lobby textfield
 - History Tab: each has the other client's username in its history tab of potential histories. Double clicking the should show "No chat has been

created..."

5. In "username2"'s lobby, double-click "username2"
 - a. Expectation: nothing happens.
6. In "username2"'s lobby, double-click "username1"
 - a. Expectation: creates new tab that says "Chat with username1"
 - b. New Chat tab contains both usernames.
7. Complete Test 1 steps with new valid username "username3"
 - a. Expectations:
 - Updated Lobby onlineUsers list with all users
 - Print "username3 joined chat..." in lobby of other two users
 - Updated History tab with history list of all users not including oneself
8. "username2" sends message in individual chatroom to "username1"
 - a. Expectations:
 - message shows up on username2's individual chatroom
 - individual chatroom tab pops up for username1
 - message shows up on username1's individual chatroom
 - message shows up on message history between username1 and username2 for both users.
9. "username1" sends individual chat to "username2"
 - a. Expectations:
 - message shows up on username1 and username2's individual chatrooms
 - message shows up on message history between username1 and username2 for both users.
10. Close MainWindow of "username2" by clicking the "Disconnect" button
 - a. Expectations:
 - individual chat tab with username2 (on the username1's MainWindow) disappears
 - disconnect message shows up on history of username1's chat with username2
 - disconnect message shows up on lobby of both username1 and username3
11. Close MainWindow of "username3" by force quit
 - a. Expectations:
 - disconnect message shows up on lobby of username1

>>Test 3: concurrency and race conditions

- Run multiple clients to connect to one instance of Server
- Have clients try to...

- Login with the same username at the same time → one user is denied use of the same username
- Send messages at the same time → the messages should appear (one after another) in both MainWindows and no message requests are lost