CONCURRENCY & TESTING STRATEGY

CONCURRENCY STRATEGY

Client-Side

Within the client, we try to ensure thread safety by only letting one thread run per client. This thread either sends commands to the server from a queue, or receives commands and puts them into a queue. The adding to and removing from the queue is handled by two methods in the Controller class. In order to make the queues thread safe, they are synchronized in these two methods.

Also in the Controller, the GUI is mutated. handleMessage(String message) parses the command and accordingly mutates the GUI. In order to do this in a thread-safe manner, without causing deadlock, each individual GUI object is synchronized. The handleMessage method itself does not need to be synchronized, since it pulls up messages from the outputQueue one-by-one.

Server-Side

Each ClientHandler maintained by the server runs on a separate thread. They each call methods on the server, in themselves, and in their ConversationHandlers. All clients have access to the server's methods at all times, but the server methods that modify server data such as the registered clients and conversations, are synchronized accordingly. All mutator methods in the conversationHandler class are synchronized since the clients in each conversation must be set before messages are sent. All data will be as private as possible but each class will have some public methods.

The goal for this server is to allow all threads to independently access server methods without receiving incorrect data in response.

TESTING STRATEGY

Client-Side

1. ChatClient

In order to test ChatClient, one needs to open a socket and communicate with a Server. Since we don't care about what commands are passed, only that the communication occurs safely, we can initially use method stubs for the getMessageToServer() method (which reads from the queue in Controller) and handleMessage() method (which calls handleMessage in Controller).

getMessageToServer() would pull out commands from an arraylist of commands, while handleMessage simply prints to System.out. This way we could test if elementary communication is occurring.

Another important thing to check is Server disconnect. If the server disconnects, we'd normally be passing on a message to the GUI. in this case we could just print to the System.out and make sure disconnecting the server abruptly causes this to be printed.

2. Controller

We will write unit tests to this that imitate ChatClient behavior and call the Controller methods and check the response handleMessage gives. This should test that the Controller is indeed thread-safe and that it parses messages correctly. Since it's hard to test on the GUI, we will start out with a method stub for handleMessage that instead of updating the GUI prints a String with the parsed command - which we can test using JUnit assertions. We also check the queue implementation and handle for all edge cases of the queue (empty, has some elements, has repeated elements etc.) using JUnit Tests.

3. Integrating ChatClient and Controller

Writing JUnit tests that create new controllers by adding elements to the queue using the addInputMessage() method, and then pass these controllers to clients that communicate with the server. At the end we must assert that the messages in the outputQueue from getClientMessage() matches the expected output.

testOneUser(): this is the basic test that just creates one user with one queue command "SIGNIN USERNAME x"

testBasicUsersConversation(): tests starting a conversation between two users and sending messages back and forth

testMultiUserConversation(): tests having a multi-user chat (so 2+ clients created)

4. GUI

There are two major communication protocols we need to test to ensure GUI correctness: messages from the GUI to the Controller, and messages from the Controller to GUI. Messages from the GUI i.e. user input to the Controller can be done by writing manual test cases for the GUI, and then checking whether the input queue in the Controller matches the expected text generated from the GUI's message, that can be tested by a simple System.out.println() message to the console.

A different set of tests will need to be written while communicating from the Controller to the GUI. This too can be tested by writing JUnit tests that call functions in Controller and then visually checking if the appropriate GUI changes are being made. In the final testing document, we shall paste in screen shots of these tests.

Server-Side

4. Server

We will write unit tests to simulate a multitude of users connecting to the server. They will then

try to start conversations with one another and add other users to their conversations. We will ensure that the correct messages are sent to the users and if order is of importance, we will ensure that the order the messages are sent is also correct. We will test to see how the server responds to users suddenly disconnecting while they are sending messages and while messages are being sent to them.

We will be using a black box testing strategy. For the server we will write:

testSignIn() : Tests if multiple users can connect to the server. Tests to make sure no username can be used twice.

testStartConversation() : Tests if a user can start a conversation with another user. Tests that messages can be sent back and forth correctly. Tests how the server responds to one or more of the users suddenly disconnecting.

testAddUser() : Tests if a user can be added to an already established conversation. Tests if a user can be removed from an established conversation cleanly. Tests how the server responds to a user being suddenly disconnected.

Some white box tests as well:

testConversationRemoved() : Tests that a conversationHandler object is removed when there are no users in it.

GUI Frames:

Figure 1: Login window

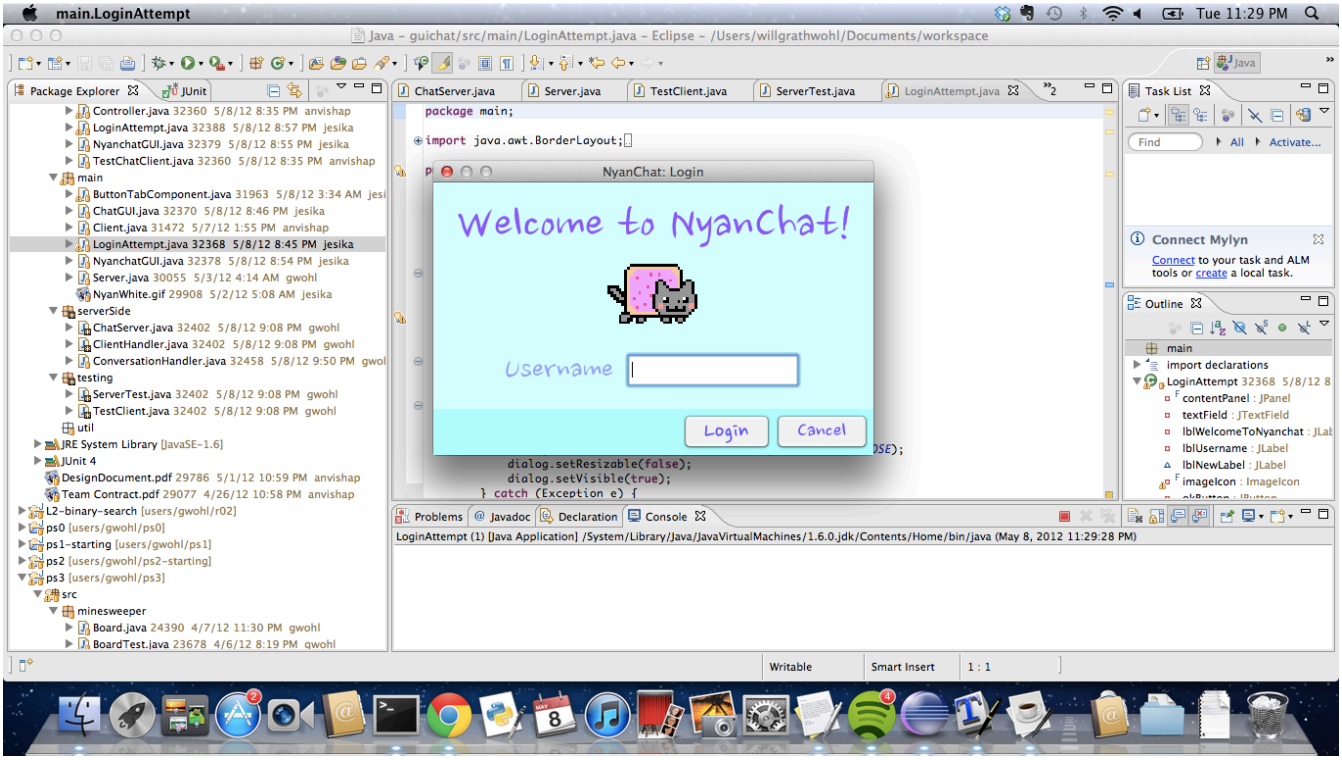

Figure 2: Welcome window (login success)

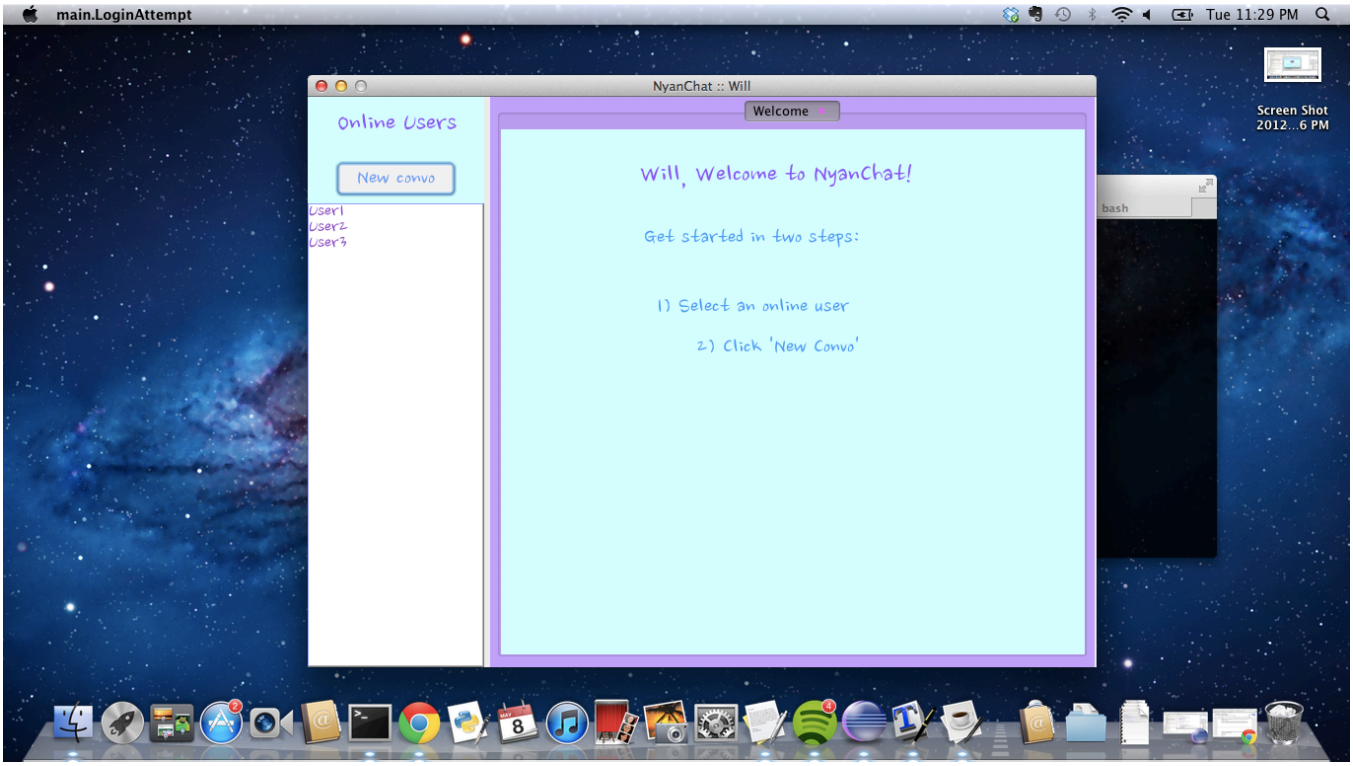

Figure 3: Chat window