Concurrency Strategy

Due to the fact that we have multiple clients and several instances of multi-threaded programming, our chat client must be careful to avoid concurrency issues such as race conditions and deadlocks. We avoid these problems through two primary ideas. Firstly, all modifications of the GUI (as well as many other variables on the client side) are performed on the same thread, the event thread, through the usage of SwingUtilities.invokeLater() or through event listeners. Secondly, all other instance variables, if they are modified, are always modified in a thread-safe manner, by either ensuring all modifications are on the same thread or a lock is used.

The thread-safety is only guaranteed if the preconditions for all the methods are followed. This includes only creating a GUI element

Race conditions:
Race conditions are avoided using the general strategy, implemented in each of the classes. Every instance variable is always modified in a thread-safe manner such that race conditions are avoided. Either the variable is locked before modification or all the modifications are performed in the same thread (or the modifications cannot logically produce a race condition).

Deadlocks:
Deadlocks cannot occur in our code because at no point in our code is more than one lock obtained.

Individual Classes:

- User: thread-safe due to immutability.
    - String username: never changed, so this is thread-safe.
- ChatMessage: thread-safe because no variables can be modified.
    - User origin: never changed, so this is thread-safe.
    - String message: never changed, so this is thread-safe.
    - Time time: never changed, so this is thread-safe.
- Conversation: thread-safe through a combination of having immutable variables and synchronized methods
    - int id: never changed, so this is thread-safe.
    - Set<User> users: all accessor methods and mutators are synchronized through a lock on the set.
- ChatServer: thread-safe due to the usage of thread-safe data structures
    - ServerSocket serverSocket: excluding the constructor, only one method uses the serverSocket. This serve() method is generally only run once and only calls serverSocket.accept().
    - ConcurrentMap<User, Socket> clients: thread-safe because the data structure is

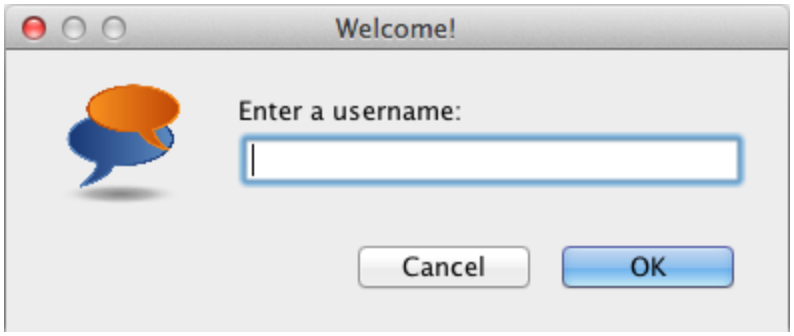
thread-safe. To avoid race conditions, we lock clients whenever we modify it.
    - ConcurrentMap<Integer, Conversation> conversations: thread-safe because the data structure is thread-safe. The individual conversations are thread-safe as well as described before. To avoid race conditions, we lock conversations whenever we modify it.
    - BlockingQueue<String> queue: thread-safe because the data structure is thread-safe.
    - Sockets (stored in the ConcurrentMap clients):
- ChatServerClientThread: thread-safe because no variables can be modified.
    - Socket socket: only one method reads from this socket, run(), reads from this socket. Writing to this socket happens in two places. When the server receives a "login" message, it checks whether a username is valid and responds on the ChatServerClientThread directly by printing to the socket's output stream in the server's tryAddingUsername() method. The only other times the socket is written to is if the ChatServer's sendMessageToClients() method is called. These two will never happen at the same time because sendMessageToClients() only sends messages to clients that have already had their usernames added to the clients ConcurrentMap. Meanwhile, tryAddingUsername() only sends the output before adding the Socket to the ConcurrentMap, so there will never be an instance where these two methods are both trying to output to the same Socket. In addition, sendMessageToClients() is only ever called from one thread (in the work() method), so there will be no race conditions there. tryAddingUsername() is also only ever called from one thread for a specific Socket, in run() in ChatServerClientThread.
    - ChatServer server: the state of the server is never modified by the ChatServerClientThread.
- UserListener: thread-safe because everything is modified exclusively in the event thread.
    - JLabel userLabel: only modified through the MouseListener, which happens in the event thread.
    - ChatClientModel model: uses model.addChat(), which is thread-safe when run in the event thread.
    - User friend: not modified, so this is thread-safe.
- ClientListeningThread: thread-safe because the only method in this class calls an outside method that is thread-safe.
    - ChatClientModel model: uses model.listenForResponse(), which is thread-safe.
- ChatBox: thread-safe because most instance variables are GUI elements, and these are only modified on the event thread. The only other instance variable only has thread-safe methods called.
    - JTextArea display: only modified in the event thread.
    - JTextArea message: only modified in the event thread.
    - JScrollPane displayScroll: only modified in the event thread.
    - JScrollPane messageScroll: only modified in the event thread.
    - ChatBoxModel model: only an accessor method is ever used.

- ChatBoxModel: thread-safe because everything is only modified through the event thread.
    - ChatBox chatBox: only ever modified through the event thread.
    - int conversationID: never modified.
    - ChatClientModel model: only ever modified through the event thread.
- ChatClient: thread-safe because the GUI elements are only modified in the event thread and other variables are either never modified or are accessed in a thread-safe manner.
    - User user: never modified after being initialized in the constructor.
    - Map<String, JLabel> userLabels: only ever modified in the EventThread (same goes for the individual JLabels).
    - JPanel users: only modified in the event thread.
    - JPanel onlineUsers: only modified in the event thread.
    - JScrollPane userScroll: only modified in the event thread.
    - JLabel welcome: only modified in the event thread.
    - JPanel welcomePanel: only modified in the event thread.
    - ChatClientModel model: only accessed in a thread-safe manner.
- ChatClientModel: everything is done in the event thread, so there are no thread-safety issues.
    - User user: only accessed and modified in the event thread, assuming that tryUsername() is called from the event thread.
    - Socket socket: the socket is only read from the listenForResponse() method, so there are no thread-safety issues with reading. In terms of writing, all writing is done in the event thread so there should be no concurrency issues. Writing always stems from either tryUsername() or handleRequest(), both of which run in the event thread when modifying variables. The initialization is only performed once and before any listening occurs, so there are no concurrency issues there.
    - ChatClient client: only ever modified from the event thread.
    - HashMap<Integer, ChatBoxModel> chats: thread-safe due to the usage of a thread-safe data structure and the fact that the modifications are performed from the same thread.
    - BlockingQueue<String> messages: only ever modified from the event thread.

Preliminary GUI

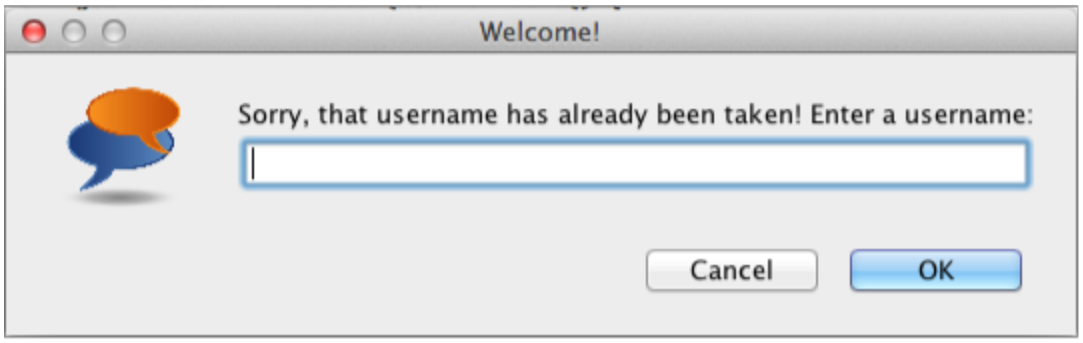
Welcome

Start Client:



Enter “Ben”:

If the username “Ben” is already taken:

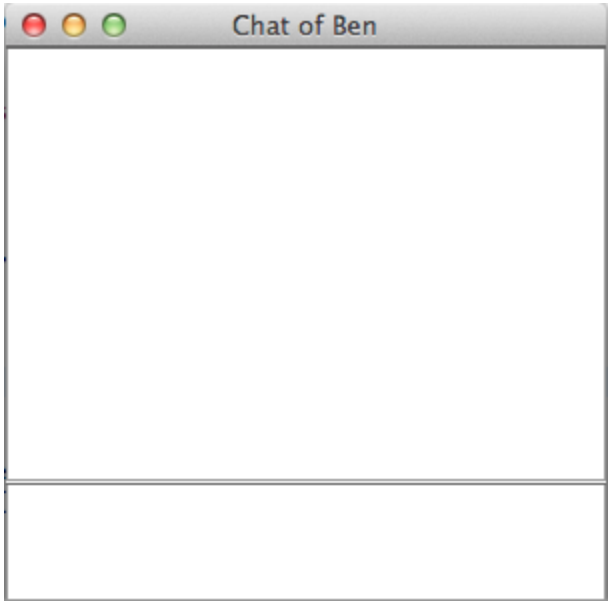


ChatClient

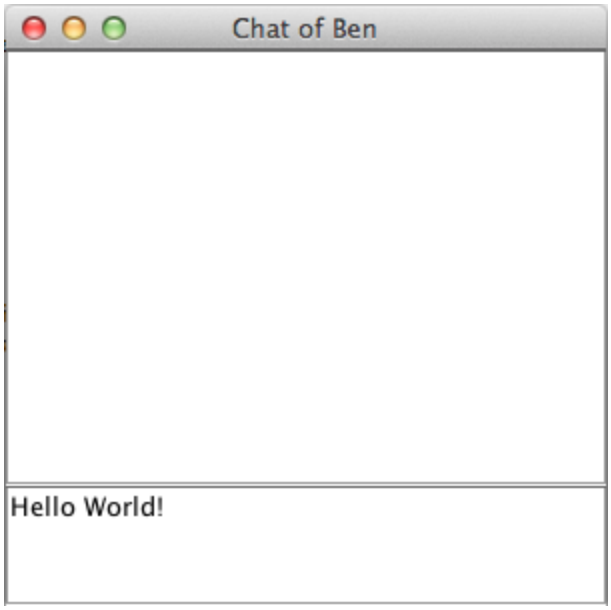
If username “Ben” is not taken:

ChatBox

Click on “Alyssa” (note that ChatClient remains open):



Type “Hello World!”:



Press Enter:


Chat of Ben
Ben: Hello World!

Testing Strategy

Integration Tests:

These tests are a series of manual tests that verify that everything works together as a system. We plan to do this by running a chat server on one person's computer and then using two computers to set up multiple clients and verify that everything we see the GUI do is what we expect. These are not nearly as rigorous as the other tests, since every feature is pretty much already tested, but we will verify that basic functionality works:

- General logging in and logging out tests:
    - Start the server.
    - Connect to the server by logging in with 4 users. Users should successfully show up on the list of logged in users for all 4 users.
    - Have one user logout. This user should disappear from all the user lists of the other three users.
    - Have the user log back in with a username that is already taken. Regardless of how many times this user tries to log in with that username, it should not be allowed.
    - Close one client connection forcefully. This user should be logged out on all screens.
- Conversation testing:
    - Start the server.
    - Connect two users to the server.
    - Start a conversation. Both users should have a new chat box open up.
    - Enter lines of conversation on both users. The messages should show up in the same order on both chat boxes, even if they are sent at the same time.
    - Invite a user to the conversation. A new message should appear for all users notifying them that the new user has joined.
    - The new user should see the list of users already in the conversation.
    - Have a user leave the conversation by logging out. Make sure that the users remaining in the conversation are notified.
    - Start another conversation between a different set of users and make sure that the system can handle multiple conversations at a time.

Conversation Class:

To test the Conversation class we need to ensure that adding and remove users from a Conversation is done safely. Two the methods to be tested are addUser and removeUser.

- addUser:
    - If the new User is not in the Conversation then it should be added.
    - If the new User is in the Conversation then nothing should happen.

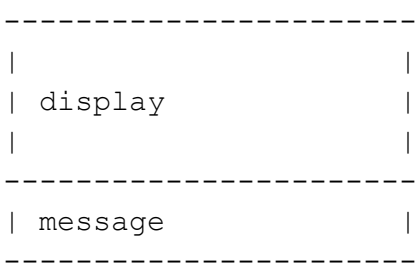
- removeUser:
    - If the User is in the Conversation then it should be removed.
    - If the User is not in the Conversation then nothing should happen.

Finally both of these methods will be tested for thread safety.

- addUser: If two threads attempt to add a new User to the at the same time, the User should be added only once.
- removeUser: If two threads attempt to remove a User at the same time, the User should be removed.

ChatBox Class:

Manual Testing:
Testing for the ChatBox will be completed manually. This will essentially consist of checking that the ChatBox is formatted properly. We want the ChatBox to appear as follows:

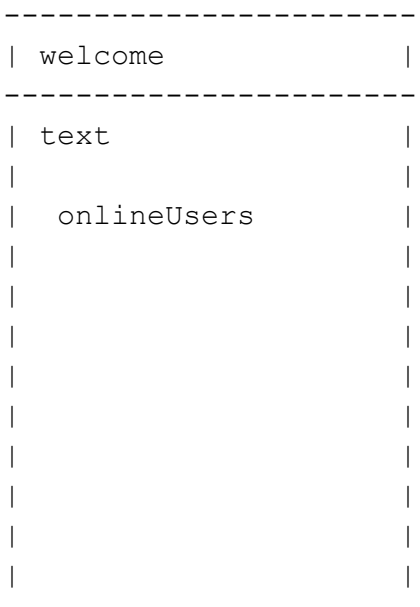


Things to check for:

- display and message are formatted as depicted above.
- The text in display cannot be edited
- The text in display wraps around
- A scroll bar appears when display contains sufficient text
- The default position of the display scroll bar is at the bottom
- The text in message can be edited
- The text in message wraps around
- A scroll bar appears when the user types sufficient text
- The default position of the message scroll bar is at the bottom
- The height of message does not change as the size of the window increases
- All other dimensions increase proportionately with size of window
- Initial window size is 300 x 300
- Both message and display have black foreground and white background

ChatClient Class:

Manual Testing:
Testing for the ChatClient will be completed manually. This will essentially consist of checking that the ChatClient is formatted properly. We want the ChatClient to appear as follows:



Things to check for:

- welcome displayed above onlineUsers
- welcome displays the message "Welcome, username!"
- welcome message is centered
- welcome has white foreground and grey background
- onlineUsers displays usernames (one per line)
- Scroll bar appears for onlineUsers once there are sufficient users
- onlineUsers has black foreground and white background
- Appropriate spacing between text and edges of window
- Usernames are tabbed over slightly from instructional text
- Window has initial size 200 x 400
- Height of welcome does not change as window is resized
- All other dimensions increase proportionally as window is resized

UserListener Class:

Manual Testing:
Testing for the UserListener class will be done manually. To do so involves creating a ChatClient GUI and making sure that following effects occur:

- Hovering over a username changes the color of the username to blue
- Leaving the username changes to color of the username back to black

- Clicking on the username results in a new ChatBox being opened
- Pressing on a username (without releasing) has no effect
- Releasing over a username (without having pressed there) has no effect

ChatServer Class:

We can use automated testing here to test the ChatServer because there are sufficient public methods. Essentially, we need to test to make sure the server's state is always what we should expect it to be given certain messages (because only messages can change the server's state) and we need to test to make sure that the server is thread-safe.

- serve:
    - Make sure that the ChatServer can accept multiple client connections and process them all concurrently.
    - Make sure that the ChatServer continues serving even if one of the client sockets closes unexpectedly.
- tryAddingUser:
    - Branch 1: make sure that the client socket receives the correct output if the username is invalid. Example: the client socket sends "login asdf" and then another client socket also sends "login asdf". The second client should receive "invalid".
    - Branch 2: make sure that the client socket receives the correct output if the username is valid, and that all future attempts to try that username are invalid. Example: the client socket sends "login asdf" and should get "success" as a response, followed by a "login asdf" message. Another client trying "login asdf" should receive "invalid".
- addMessageToQueue:
    - Can be tested by having a dummy client connect to the server and then putting messages into the queue, making sure the dummy client receives the messages it is supposed to. Example: the dummy client should receive "login asdf" if "login asdf" is added to the queue. If "start asdf qqq" is added to the queue after a "login qqq" command, then "join 1 asdf" and "join 1 qqq" should be received by the dummy client.
    - Ensure that an exception is thrown when one of the users in the targets list is not a user that is logged in.
    - Ensure that the method correctly runs if the target users list is empty.
    - Can also be tested by adding methods that allow us to query the state of the server. Essentially, there can be accessor methods that access the server's clients (map from User to Socket with getClients()) and the server's conversations (map from Integer to Conversation with getConversations()). Then, we use dummy sockets (a class that extends Socket but does nothing but deliver output) to receive server output and use this method to add input to the server, checking the state of the server with every input command to make sure it is

valid. This can be combined also with tryAddingUser to add new users. The accessor methods allow us to easily generate unit tests for the server without any real networking.

The rest of the methods are private and thus not testable but the client-server communications tests will test the ability of the server to communicate with the client through the protocol, which effectively tests the rest of the methods in the ChatServer.

ChatServerClientThread Class:

The best way to test the ChatServerClientThread is to make sure it correctly adds messages to the server's queue, as this is the primary purpose of the ChatServerClientThread. In order to do this, we can use a series of tests that uses two dummy clients. Both connect to the server but only one of them sends messages. The other merely waits for responses to make sure that it and the other client are receiving the correct responses. The second client can log on and off as well as leave and join conversations, because each of those actions involves the server revealing part of its state (the list of users logged in or the list of users in a given conversation).

Sample tests:

- Connect clients 1 and 2. Client 1 sends "login asdf". Client 2 should receive "login asdf". Client 1 sends "logout asdf". Client 2 should receive "logout asdf".
- Connect clients 1, 2, and 3. Client 1 sends "login asdf". Client 2 sends "login qqq". Client 2 should receive "success", "login asdf", and "login qqq" in that order. Client 1 sends "start asdf qqq". Client 2 should receive "join 1 qqq" and "join 1 asdf". Client 1 sends "say 1 asdf 123456 hello this is a test message". Client 2 should receive "say 1 asdf 123456 hello this is a test message." Client 1 should send "leave 1 asdf". Client 2 should receive "leave 1 asdf". This is more of an overall test but serves to make sure that all these messages are being added to the blocking queue (because if they are not, then it is impossible for the second client to be receiving the messages that it does).

In addition, the ChatServerClientThread handles part of the addition of a new username. The same tests for tryAddingUser should be applied here, except using an actual connection instead of just passing things into the method directly. This involves testing both a valid login and an invalid login.

Finally, we must test to make sure that a ChatServerClientThread responds correctly if the client disconnects. The ChatServerClientThread is responsible for sending the "logout" command if the client leaves without sending the "logout" command. To do this, we can test a case where two clients connect and the first exits by force quitting the chat client program. The second client must receive a logout message saying that the first client has left.

Client-Server Communications Testing:

This is the bulk of the testing and is an integration test. Here, we essentially test the private work() method of the ChatServer that we were not able to test in the ChatServer tests. To do this, we will continue to use the dummy client class that does nothing but send messages and verify received messages. The general testing strategy is to verify the correctness and the thread-safety of the server's handling of the protocol. Each message is only sent to a subset of the logged in users, so we should always make sure that the correct set of users receives the message.

Here, we verify that the server and the server's client communications thread correctly handle all the messages in the protocol:

- "success" and "invalid" tests:
    - These have already been tested when testing the ChatServerClientThread.
- "login" tests:
    - Whenever a user logs in, every user should receive a "login" message. Start clients 1, 2, and 3. Client 1 sends "login a". Client 2 sends "login b". Client 3 sends "login c". Client 1 should receive "success" and "login a". Client 2 should receive "success", "login b", and "login a". Client 3 should receive "success", "login c", "login a", and "login b". To make sure that the server continues to properly maintain the list of logged in users, we can have client 1 logout ("logout a") and then have client 4 log in with "login a". Client 4 should receive "success" (no username conflict), "login a", "login b", and "login c".
    - We can run the same test above for more complicated usernames, such as "asdfasdfasdfasdfasdfasdfasdfasdf32134123412341234123412341234123412341234;;1;1;2;43;1234;" and everything should still work, as long as there are no whitespace characters in the username.
- "logout" tests:
    - Whenever a user logs out, all logged in users should be notified. Start clients 1, 2, and 3 and have them all login with usernames "a", "b", and "c" respectively. Log user 1 using "logout 1". Clients 2 and 3 should both receive "logout 1". Close client 2 by closing the socket. Client 3 should receive "logout 2".
    - We can have a dummy client send "logout qqq" (for a user "qqq" that is not logged in). An error should be thrown.
    - We also need to make sure that "logout" causes the user to leave all conversations that the user is involved in. To do this, we start a conversation among three clients and have one of them logout. The other two clients should be notified that the first client has left the conversation.
    - To test thread-safety, we have a conversation with users 1 and 2. We then have both users 1 and 2 logout at the same time. Only one of them should receive the "logout" command and the "leave" command (because the other should have logged out first).
- "start" tests:
    - We need to correctly test whether a conversation can be started with any number

of users that is greater than 0. Start clients 1, 2, and 3 and have them all login with usernames “a”, “b”, and “c” respectively. We then use “start a b c” from any of the clients. Each client should receive “join 1 a”, “join 1 b”, and “join 1 c” to signify the joining of the users into the conversation. Conversation ID generation will be tested later.

- “say” tests:
    - We create a conversation and have users communicate using the “say” command. We verify that the correct users are attributed to each statement and that the statements are not modified when they are sent back to all users involved in a conversation. We should also verify that only the users involved in a conversation receive the “say” message for that conversation.
- “join” tests:
    - We start a conversation between two users and then send a “join 1 c”, where “c” is a third user, to the server. “join 1 c” should be sent to all users now involved in the conversation and not to a fourth user “d” who is also logged in.
    - Joining a conversation with an invalid ID should throw an error.
- “leave” tests:
    - We start a conversation with three users and send a “leave 1 c”, where “c” is the third user, to the server. “leave 1 c” should be sent to the remaining two users involved in the conversation but not to “c” or a fourth user “d” who is also logged in.
    - We have a user leave and then rejoin the conversation. This should work as expected.
    - The correct messages should be sent out even if two users leave in quick succession. This is a thread-safety test.

For all the tests, we should verify that an error is thrown if an invalid username is used. This is behavior not allowed by the specifications (and should not happen if the client and server are correctly implemented) so these tests are not crucial.

Finally, we indirectly test a method that was not testable earlier. To test nextConversationID, which should always return the smallest positive integer ID that does not belong to a current conversation, we try the following test to verify the correctness (and thread-safety) of this method:

- Start 1000 conversations. These should have IDs 1 through 1000 in that order.
- Quit conversations 4, 123, and 999.
- Start three new conversations. These should have IDs 4, 123, and 999 in that order.
- Quit conversations 123, 12, and 417 in that order.
- Start three new conversations. These should have IDs 12, 123, and 417 in that order.
- Quit every conversation.
- Start a new conversation. This should have ID 1.

ChatClientModel Class Testing:

- startListening: this will be tested via manual testing
- quitChats:
    - Programmatically initialize multiple chats, and then programmatically call quitChats. All of the chats should close.
    - Manually start multiple chats. Then, close the main ChatClient window. All of the ChatBoxModel windows associated with that ChatClient should close.
- tryUsername:
    - Programmatically try
    - Manually try entering the following usernames:
        - A username of length 0
        - A username made of both letters and characters
        - An extremely long username
        - A username that has already been used
- addChat:
    - Test whether manually adding a chat has the appropriate effect; that is, the chat shows up on the correct ChatBox GUI.
    - Programmatically call addChat and test whether the chat shows up on the correct ChatBox GUI.
- sendChat:
    - Test whether manually sending a chat has the appropriate effect.
    - Test whether programmatically sending a chat has the appropriate effect.
- submitCommand:
    - Test whether manually submitting a command has the appropriate effect on the GUI (in manual testing)
    - Test whether programmatically submitting a command has the appropriate effect on the GUI.
- listenForResponse: This method will be tested in the client-server
- handleRequest: programmatically submit requests and see if they have the appropriate effect on the GUI.

ChatBoxModel Class Testing:

- addChatLine: programmatically test whether the correct message is sent.
- addChatToDisplay: programmatically test whether the chat is correctly added to the display. Add multiple chat lines to the display at once, and see whether threading issues occur.
- keyReleased:
    - Programmatically test whether the correct chat message is sent when the KeyEvent is VK_ENTER.
    - Test that no message is sent when the chat message contains nothing--i.e. an empty string. Test this by repeatedly sending VK_ENTER KeyEvents without typing anything into the ChatBox in between.