Conversation Design

A conversation will be represented by a Conversation class. There are two ways to start a Conversation:

- Click on a user on the buddy list. This will start a Conversation between that user and the clicked user. This Conversation will only even consist of those two users. This type of Conversation will always have isGroupChat set to false.
- Click on the “Group Chat” button, which will allow the user to choose other online users to invite to a group chat. This type of Conversation will always have isGroupChat set to true. The people that are invited to the Conversations are the only ones that can ever be in the Conversation, but the Conversation can continue if users leave.

The server will hold a HashMap of the current conversations. The hashmap will map conversation IDs to Conversation objects.

Conversation Class:

`public class Conversation`

- `private final int id;`
  - `The unique ID associated with this conversation.`
- `private final Set<User> users;`
  - `The list of users in the conversation.`
- `private final set<User> inactiveUsers`
  - `The list of users from the Conversation that logged out.`
- `public Conversation(Set<User> users, int id);`
  - `Creates a new conversation with the given user list and ID.`
- `public void addUser(User user);`
  - `Adds a user in a thread-safe manner.`
- `public void removeUser(User user);`
  - `Removes a user in a thread-safe manner.`
- `public void deactivateUser(User user);`
  - `Deactivate user in thread-safe manner so history is saved.`
- `public synchronized Set<User> getUsers();`
  - `Returns the user list (a copy of, not a reference).`
- `public synchronized Set<User> getInactiveUsers();`
  - `Returns the inactive user list.`
- `public boolean isGroupChat();`
  - `Returns isGroupChat.`
- `public int getID();`
  - `Returns the ID of the conversation.`

User Class:

public class User

- private final String username
  - Stores the username of this user. The User class behaves essentially as if it were a String (for comparison and hashing purposes).
- private final int avatar
  - Stores an integer corresponding to the avatar chosen by the user.
- public User(String username);
  - Creates a user with the given username (avatar is -1).
- public User(String username, int avatar);
  - Creates a user with given username and avatar.
- public int compareTo(User other);
  - Compares users by username
- public int hashCode();
  - Two Users have the same hash code if their usernames are identical.
- public boolean equals();
  - Two Users are equal if their usernames are the same.
- public String getUsername();
  - Returns the username of the user.
- public int getAvatar();
  - Returns the avatar number of the user.

Client-Server Protocol

The basic structure of our instant messaging system will consist of a client (used through a GUI) and a server. The chat server and each chat client will send messages through a plain text protocol (telnet) using basic socket communication in order to update each other on the overall state of the instant messaging system. The GUI and its controller will automatically send messages to the server and interpret messages from the server. Each message sent from the server to a client and vice-versa will be a single one-line string.

Protocol: Client to Server and Server to Client

There exists a text protocol for one-line messages from the client to the server and from the server to the client.

Each message sent from the server to the client or from the client to the server must be of the "message" type, defined in the grammar below. Any message must be one line and cannot contain any newline characters.

In general, a username is defined to be any sequence of alphanumeric characters. An id represents a conversation id, which is unique for each conversation at any given time. An id is any positive integer. An avatar is an avatar id, which is an integer. A line of text sent in a conversation can be anything that is one line. Our chat client allows multi-line chat messages, but they are sent over as multiple messages so that each message that represents a line of conversation only holds one line.

Grammar and Protocol: Server to Client

```
login_success :== "login_success"

login_invalid :== "login_invalid_user"

user_joins :== "user_joins" SPACE username SPACE avatar
user_leaves :== "user_leaves" SPACE username

chat_start :== "chat_start" id SPACE SPACE username SPACE username

group_chat_start :== "group_chat_start" SPACE id
group_chat_join :== "chat_join" SPACE id SPACE username
group_chat_leave :== "chat_leave" SPACE id SPACE username

say :== "say" SPACE id SPACE username SPACE text
typing :== "typing" SPACE id SPACE username
cleared :== "cleared" SPACE id SPACE username
```

login_success: this is sent from the server to the client after a successful login_attempt message.

login_invalid: this is sent from the server to the client after an unsuccessful login_attempt message (typically with a username conflict).

user_joins: this is sent from the server to all connected clients after a new user successfully logs in. The avatar is specified so that other clients know which avatar the new user picked.

user_leaves: this is sent from the server to all connected clients after a user logs out or is disconnected for whatever reason.

chat_start: this message notifies a connected client that a private conversation has been started between the first user and the second user. The ID is also specified. The first user is the one that initiated the conversation. Our client will only open the chat box if the first user corresponds to the client's username. Else, the box will remain hidden until the other user says something. Only members of the conversation are notified.

group_chat_start: this message notifies a connected client that a group conversation has been started with the list of users. The ID is also specified. The chat box is opened on all clients. Only members of the conversation are notified.

group_chat_join: this message notifies a connected client that a group chat now has a new user joined into the conversation (with the specified ID). Only members of the conversation are notified.

group_chat_leave: this message notifies a connected client that within a group chat (with the specified ID), a user has left. Only members of the conversation are notified.

say: this message notifies all users participating in the given conversation (with the specified ID) that someone has said a message.

typing: this message notifies all users participating in the given conversation (with the specified ID) that the user has typed a character.

cleared: this message notifies all users participating in the given conversation (with the specified ID) that the user has cleared the text box of characters.

Grammar and Protocol: Client to Server

```
login_attempt :== "login_attempt" SPACE username SPACE avatar
logout :== "logout" SPACE username
```

```
chat_start :== “chat_start” SPACE username SPACE username

group_chat_start :== “group_chat_start” (SPACE username)+
group_chat_leave :== “chat_leave” SPACE id SPACE username

say :== “say” SPACE id SPACE username SPACE text
typing :== “typing” SPACE id SPACE username
cleared :== “cleared” SPACE id SPACE username
```

login_attempt: this is sent from the client to the server when trying to log in. The avatar is also specified so that after logging in, other clients can be notified which avatar the new user chose. Upon successful login, the server responds by sending a user_joins message to all connected clients (in addition to either sending login_success or login_invalid back to the original client).

logout: notifies the server that a particular client wishes to log out. This is also sent by one of the server’s client threads when a client forcefully disconnects. The server responds by sending a user_leaves message to all connected clients.

chat_start: notifies the server that the first user wishes to start a conversation with the second user. The server responds by sending the chat_start command, now with a conversation ID generated, to the two users involved in the conversation.

group_chat_start: notifies the server that a group chat is about to begin. The server responds by finding a conversation ID and then sending the group_chat_start command, with the conversation ID, back to the users involved in the conversation.

group_chat_leave: notifies the server that a user wishes to leave a group conversation. The server responds by sending the group_chat_leave message to all users remaining in the group conversation.

say: notifies the server that a user wishes to say a message in a conversation. All users in the conversation are notified.

typing: notifies the server that a user is currently typing into a conversation. All users in the conversation are notified.

cleared: notifies the server that a user has cleared text in a conversation. All users in the conversation are notified.

Catching Username Conflicts

When a new client is started, it will ask the user for a username. This username is then sent to the server using the “login_attempt” message. If the username is valid, the server will reserve it

and then send the “user_joins” message to all clients that are already with usernames, as well as the newly connected client. The newly connected client is also notified by the “login_success” command. Else, the username will immediately send back the “login_invalid” message to the specific client that requested the bad username.

Typing and Cleraed

These messages are used to implement the “User is typing...” and the “User has entered text...” features. When a user types into a conversation, all participating users are notified and will display the “User is typing...” message for 1 second. After 1 second, if the user has entered text but has not submitted it, the text will switch to “User has entered text...”. This is done through the use of a timer. If at any point, the user clears the text box, the user is no longer typing and should not display these messages. This is the purpose of the “cleared” command.

State: Server

At any point, the server will know its socket, the sockets (and usernames) of all currently connected clients, and the list of ongoing conversations. In addition, it will run a thread for each client connected. A conversation consists of a list of users and a unique ID number.

The server starts by creating a thread that does nothing but process messages from its blocking queue. Then, the server will loop infinitely to wait for client connections. With each client connection, a ChatServerClientThread is created, which puts messages from the client into the blocking queue so that the server can process it.

```
public class ChatServer
```

- `// this class serves as the chat server`
- `private final ServerSocket serverSocket;`
  - `Used to communicate with clients.`
- `private final Map<User, Socket> clients;`
  - `Holds a list of all connected clients.`
  - `Updated whenever a user logs in or logs out.`
- `private final Map<Integer, Conversation> conversations;`
  - `Integer corresponds to conversation ID.`
  - `How do we assign conversation IDs? Assign smallest possible positive integer that is not currently used.`
  - `Updated whenever a new conversation begins, or when a conversation ends (when all participating users leave).`
- `private final BlockingQueue<CommunicationsData> queue;`
  - `Stores messages from clients that run() uses to make changes to the current state and send messages back to clients.`

- public ChatServer(int port);
  - Creates a new chat server with some socket for clients to connect to. Does not start listening until the serve() method is called.
  - Creates a new thread to process messages using the run() method.
- public void serve();
  - Sets up the server socket and accepts connections, delegating a new thread for each new connection (using a ChatServerClientThread).
- private void sendMessageToClients(String message, List<User> targets);
  - Sends the message to all the clients listed, with help from the HashMap clients.
- private int nextConversationID();
  - Returns the next valid conversation ID (the smallest positive unused ID).
- private void work();
  - Run in its own thread to process messages from its BlockingQueue as they come in.
  - Updates instance variables accordingly.
- public boolean tryAddingUser(String username, Socket socket);
  - Used by ChatServerClientThread to add a new username.
  - Returns true if the username is unused (and adds it).
  - Also does all communications with the socket.
  - Updates new user and logs him or her in.
- public void addMessageToQueue(String message, Socket socket);
  - Used by ChatServerClientThread to add a message to its queue for future processing.
- public void forceLogout(Socket socket);
  - Notifies the server of a force quit.
  - The server disconnects the user from the socket.
- public void writeMessageToSocket(String message, Socket socket);
  - Sends a message through a given socket.
- public void sendMessageToUser(String message, User user);
- public void sendMessageToUsers(String message, Iterable<User>targets);
- public void work();
  - Read messages from the blocking queue.
  - Act according to the message retrieved from the blocking queue
- Methods used to take the appropriate action depending on the nature of a command retrieved from the blocking queue:

        - `public void processLoginAttemptCommand(String message, Socket socket);`
        - `public void processLogoutCommand(String message);`
        - `public void processChatStartCommand(String message);`
        - `public void processGroupChatStartCommand(String message);`
        - `public void processGroupLeaveCommand(String message);`
        - `public void processSayCommand(String message);`
        - `public void processTypingCommand(String message);`
        - `public void processClearedCommand(String message);`

`public class ChatServerClientThread implements Runnable`

- `// A thread run for every client (username) connected to the server.`
- `private final Socket socket;`
- `private final ChatServer server;`
    - `Updated based on messages received from the client.`
- `public void ChatServerClientThread(Socket socket, ChatServer server);`
    - `Creates a new ChatServerClientThread with the given socket and associated server.`
- `public void run();`
    - `Reads messages from the client socket and has the server respond accordingly using thread-safe methods.`
    - `Adds messages to server's message queue.`

State: Client

At any point, the client's state holds its current socket, a list of online users, as well as a list of ongoing conversations. A conversation consists of a list of users and a unique ID number. The client consists of the models for the GUI elements.

Both models are ActionListeners, meaning that they will react to things that happen in the GUI.

`public class ChatClientModel implements ActionListener`

- `// The model that communicates with the server and updates the chat client GUI.`
- `private User user;`
    - `Corresponds to the current user.`
- `private final Socket socket;`
    - `Used to communicate with the server.`
- `private final ConcurrentMap<Integer, ChatBoxModel> chats;`
    - `IDs of conversations mapped to the ChatBoxModels corresponding to the conversations.`

- private final ChatClient client;
  - The GUI that is updated by this model.
- private final BlockingQueue<String> messages;
  - A blocking queue for the temporary storage of messages.
- public ChatClientModel(ChatClient gui, User user);
  - Creates a new ChatClientModel with the specified GUI and user.
  - Connects to a socket.
- public void startListening();
  - Starts reading from the socket and interpreting messages.
- public boolean tryUsername(String username);
  - Queries the server to see if the given username is valid.
  - Sets the current username if this username is valid.
  - Uses the BlockingQueue messages to retrieve messages from a different thread.
- public void quitChats();
  - Closes all ChatBoxes.
- public void addChat(User other);
  - Starts a chat with a given user.
- public void submitCommand(String command);
  - Sends the command to the server.
- public void listenForResponse();
  - Waits for server messages and responds accordingly by either updating its GUI or by notifying one of its ChatBoxModels.
- public void handleRequest(String output);
  - Parses a message from the server and updates the GUI accordingly.
- public void connect();
  - Connects to the server.

public class ChatBoxModel implements KeyListener

- // One of these is created for each new window that is a chat box.
- private int conversationID;
  - Holds the current conversation ID.
- private ChatBox chatBox;
  - Corresponds to the chat box that is updated by this model.
- private ChatClientModel myClient;
- public ChatBoxModel(ChatClientModel model, ChatBox box, int conversationID);
  - Creates a new ChatBoxModel with the specified GUI, the specified client, and the conversation ID.

- public void addUserToConversation(User user);
- public void removeUserFromConversation(User user);
- public void addChatLine(String username, String time, String text);
- public void quit();
- public void keyPressed(KeyEvent e);
    - Sends a message to the server (through the client) in response to an ActionEvent from the GUI.
- public void keyReleased(KeyEvent e);
    - Sends a message to the server (through the client) in response to an ActionEvent from the GUI.
- public void keyTyped(KeyEvent e);
    - Sends a message to the server (through the client) in response to an ActionEvent from the GUI.

Concurrency Strategy

Concurrency Strategy: Server

The server can easily be made thread-safe by minimizing the usage of threads. The ChatServer class spawns a ChatServerClientThread for each connected client. Then, it has one other thread for processing messages from its blocking queue. The only data structure that is accessed in multiple threads is the blocking queue, which we know is thread safe due to its nature. Aside from that, the only other variables are all modified from the worker thread that processes messages from the blocking queue. Because everything is modified from this one worker thread, there are no concurrency issues.

Individual Classes:

- ChatServer: thread-safe due to the usage of thread-safe data structures and general serialization of operations. Each socket is only read from in one thread (the corresponding ChatServerClientThread) and each socket is only printed to in one thread (the worker thread).
- 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.
- CommunicationsData: thread-safe due to immutability.

Concurrency Strategy: Client

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.

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:

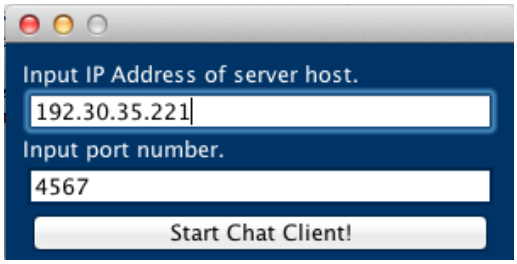
- User: thread-safe due to immutability.
    - String username: never changed, so this is thread-safe.
- ChatMessage: thread-safe because no variables can be modified.
- Conversation: thread-safe through a combination of having immutable variables and only being modified in one thread. There are also synchronized methods in case Conversation is used in multiple threads (although in our usage, it is not).
    - 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.
- 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.

- ChatBoxModel: thread-safe because everything is only 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.
- ChatClientModel: everything is done in the event thread, so there are no thread-safety issues.
    - 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 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.
- ConnectionInfoBox: everything is modified either only once or in the event thread only, guaranteeing thread-safety.
- GroupChatListener: everything is modified either only once or in the event thread only, guaranteeing thread-safety.
- GroupChatSelectBox: everything is modified either only once or in the event thread only, guaranteeing thread-safety.
- HistoryBox: everything is modified either only once or in the event thread only, guaranteeing thread-safety.
- HistoryListener: everything is modified either only once or in the event thread only, guaranteeing thread-safety.
- ChatHistory: thread-safe due to immutability.
- Emoticon: thread-safe due to immutability.

Testing Report (with GUI Images)

In general, look at the testing files for more detail. We documented our manual testing within certain .java files.

Our program passed all of the tests as detailed in the testing strategy. We implemented JUnit tests for the server and the classes used in the client model class, such as User and ChatHistory. We also provide detail on the tests we performed on each class in the Java files for that class. Our overall strategy is detailed in this report.
The following report details a walkthrough of every aspect of our GUIChat, and how we manually test each component for correctness, usability, and concurrency.
We first start a server with a certain host address and port.
We then start the client side program; the GUI opens with the connection window (ConnectionInfoBox class):

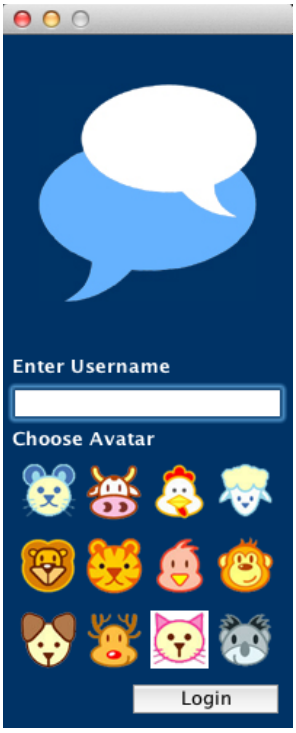


- First type in an invalid IP address with a correct port and try to submit. The input is invalid, so submission does not lead to the main chat client GUI.
- Next type in a valid IP address with an incorrect port and try to submit. The input is again invalid, so submission does not lead to the main chat client GUI.
- Next type in both an invalid IP address and an incorrect port and try to submit. Again, submission does not lead to the main chat client GUI.
- Finally type in a valid IP address and a correct port and try to submit. Since the input is now valid, submission leads to the main chat client GUI.

When testing, we start multiple client programs.

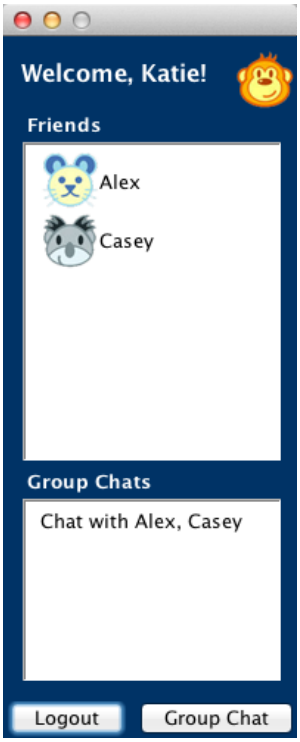
- Tested on multiple computers and with multiple clients on one computer. When testing on multiple computers, we started the server on one computer and then connected to that computer by changing the IP address entered into the ConnectionInfoBox IP address field to that computer's IP address.
- We also made the chat client accessible online (axchen.com/guichat.html) and set up a permanent server that would alway run our server program. Multiple people could then access the chat client through the website and log in. We tested the chat client with as many as six different people across six different computers.

The login window



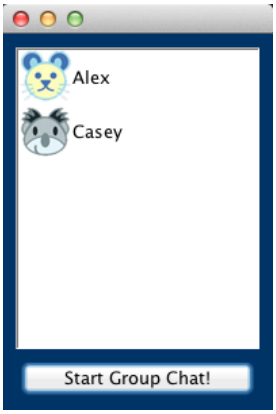
- Regarding the avatar images (and the chat bubble image):
    - Make sure that all of the images load properly, including the avatar images and the chat bubble image. A random avatar should be initially highlighted when the ChatClient GUI first opens.
    - Make sure that when the user mouses over an avatar, a light blue highlight appears, and that when the user is no longer mousing over an avatar, the light blue highlight disappears.
    - Make sure that when the user clicks on an avatar, it then becomes highlighted in white.
- Test all methods of logging in:
    - Test that you can log in by pressing enter.
    - Test that you can log in by pressing the “log in” button.
    - Test that you can log in without choosing a new avatar.
    - Test that you can log in with choosing a new avatar.
    - Make sure that you cannot log in with:
        - A username that is already in use (online)
        - A username with 0 characters (“”)
        - A username with a space
        - A username with a strange character
    - Make sure that you can log in with:
        - A username that is alphanumerically the same as a username already in user, but has letters of different cases (uppercase vs. lowercase)
        - A username that includes numbers

The main ChatClient GUI



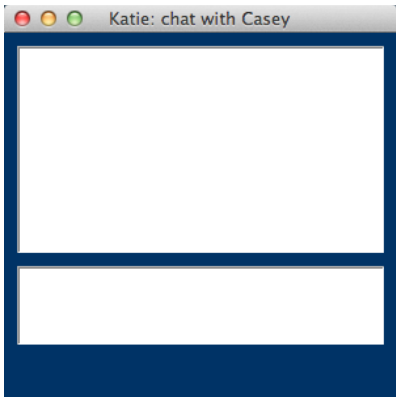
- Test that the layout is correct
- Test that online users appear in lexicographic order, and that the users' avatars appear to the left of their names.
- Test that mousing over users' names in the list of online users causes the text of those labels to turn light blue, and that when a name is no longer moused-over, it returns to its original black color.
- Test that clicking on a user's name causes a private chat window to pop up for only the user that clicked. (The chat window pops up for the other person only after the person who initiated the chat says sometime. This was modeled after gchat, in which the same thing occurs.)
- Test that clicking on past chat histories in the "past group chats" box brings up a ChatHistoryBox that contains the correct chat history.
- Test that the logout button logs the user out and quits the client window.
- Test that clicking the "Group Chat" button brings up the group chat selection window.

The Group Chat selection window



- Make sure that all of the components are in the group chat window.
- Make sure that mousing over a name highlights it by changing the text to light blue, and that no longer mousing over a name changes the text back to black.
- Make sure that clicking a name highlights it, and that clicking it again unhighlights it. Make sure that mousing over a selected name does not change the label's color.
- Make sure that clicking the "Start Group Chat!" button starts a group chat with the selected people.

The Chat Box window



- Make sure that chats appear correctly. They should include the user who sent the chat, the timestamp, and the text of the message.
- Emoticons should also appear correctly. Try all emoticons to see that they all work.
- Pressing enter in the lower text window after typing a chat should send a chat to the chatbox.
- Make sure that other users can send chats.
- Make sure that the alerts that certain other users are typing or have typed text show up correctly.
- Make sure that group chat works--i.e. all users can chat into the group chat.
- Make sure that, in group chat, the users are alerted when someone leaves the group chat.
- Make sure that the chat box title bar is captioned appropriately.

History Box window

- Make sure that the appropriate chat history shows up in the HistoryBox
- Make sure that the history label in the main ChatClient GUI corresponding to a particular history is labeled with all of the users who were ever in the chat.

Overall, we originally planned to have more automatic tests of our GUIChat program. However, we found that we could more easily and more thoroughly test these cases manually. As such, we performed extensive manual tests with many different individuals chatting at once.

Server-side testing report

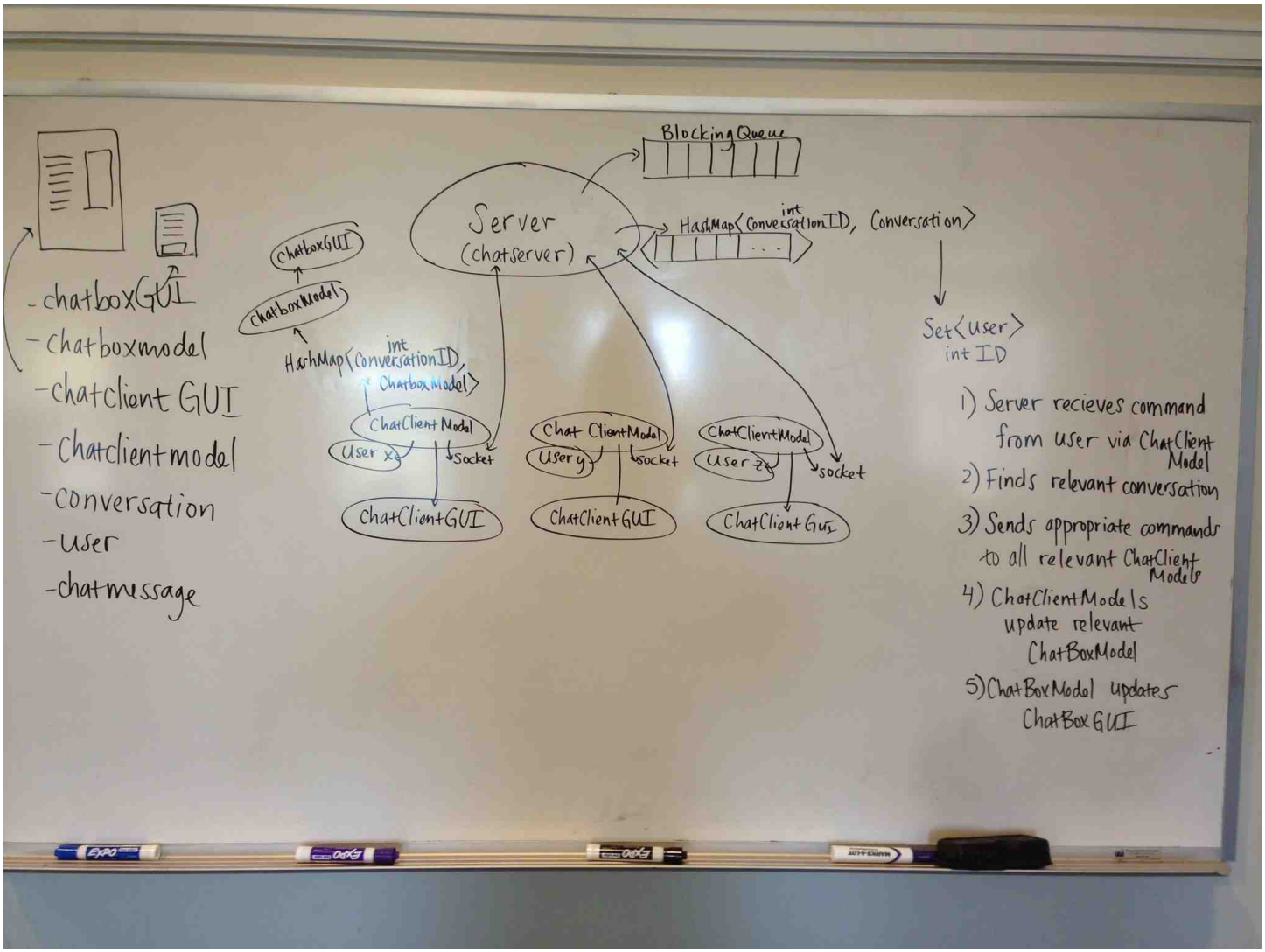
To test the server, we do two types of tests:

1) First, we tested the ChatServerClientThread by creating a dummy ChatServer to make sure the ChatServerClientThread passed on the messages connected clients to the server. The DummyServer merely checks to make sure it receives the right messages and does not process the messages.

2) Next, we tested the entire ChatServer by creating a DummyClient class that sent messages from a “client” and then read the messages and verify that they are the messages we expect. In general, there is a test for each possible command in the protocol that the server can possible send or receive, and as the tests go, they become more and more inclusive (because to test the “say” command, we must test the login and conversation start commands). Essentially, the later tests serve as integration tests. Finally, there is a concurrency test to make sure the server can handle many synchronous connections.

Results: the server passes all the required tests. We did not make any major changes due to any of the tests. The main bug that was caught and fixed was how to deal with a client disconnecting without sending a “logout” command. We fixed this by catching socket IOExceptions by sending a “logout” command from that socket to force the user to logout if the user has not already logged out.

Finally, the server is also essentially tested through our manual tests because those also require the server to work.


- chatboxGUI
- chatboxmodel
- chatclient GUI
- chatclientmodel
- conversation
- user
- chatmessage
ChatboxGUI
ChatboxModel
HashMap<int ConversationID, ChatboxModel>
ChatClient Model
User x
socket
ChatClientGUI
Server
(chatserver)
BlockingQueue
HashMap<int ConversationID, Conversation>
Chat ClientModel
User y
socket
ChatClientGUI
ChatClientModel
User z
socket
ChatClient Gui
Set<User>
int ID
1) Server recieves command from user via ChatClient Model
2) Finds relevant conversation
3) Sends appropriate commands to all relevant ChatClient Models
4) ChatClientModels update relevant ChatBoxModel
5) ChatBoxModel updates ChatBox GUI