Overall Design Approach

Our chat client has a user-centric approach where the focus is on one-on-one and group chats. Modelled somewhat after iMessage, our system does not have chat rooms, but displays a list of online users, and enables you to start a conversation with one or more other online users. Only one conversation is allowed between any given group of users. Therefore when a user logs out, or closes a conversation, that conversation is closed for all other users, and users cannot join preexisting conversations (otherwise this could result in multiple conversations between the same group of users).

User Experience

Login view presented on startup

- Text box: Server IP Address entry
- Text box: Port number entry
- Text box: Username entry
- Dropdown menu: Color selection
- Button: Login
    - Action: Checks for valid IP Address, port number, username (username must only contain letters, be less than 10 characters, not contain spaces, and must not already be in use by another user).
        - If valid: Initialize session and open conversation view
        - Else: Popup message containing error, returns to login screen.

Conversation view shown when user is logged in

- Tabbed view
    - Menu Bar
        - Username is shown, as file menu item. Menu items:
            - Logout (always displayed)
                - Action: Closes all of user's conversations, and end's the user's session. Login view is reopened.
            - Close Conversation (If current tab is a conversation, not the "New" tab)
                - Action: Closes the current conversation.
    - Tabs
        - New
            - List of all online users. Multiple selection is enabled.
            - Start Converastion Button
                - Action: Tries to start a new conversation between you and all of the Users that are selected from the list

            above. If conversation between these users doesn't already exist, a new tab is opened for this conversation. Else, a popup message appears explaining the error.
        - View Chat History Button
            - Action: If the conversation is not currently active, and a chat history exists between the selected users, the conversation history is displayed in a new window. Else, a popup message appears explaining the error.
    - Conversation tabs (for each of user's active conversations)
        - Conversation history is displayed in a scroll pane
            - the history is still displayed if a conversation a set of users is closed and reopened.
        - Text field to enter new message
        - Send button
            - Action: Sends message in the text field to all other users part of that conversation.

Classes

Client: GUI chat client runner

`public class Client`

- `public static void main()`
    - starts a GUI chat client

ChatClient: Wraps together the GUI and User classes. Starts a GUI and logs a user in.

`public class ChatClient`

- `public ChatClient()`
    - starts a UserGUI on the event thread
- `private User user;`
    - the User associated with this client, initially null but is set once login is successful
- `public void setUser(String username, String color, String socket);`
    - sets the user attribute based on the username, color and Socket
- `public void attemptLogin(String IP, String port, String`

`username, String color, UserGUI gui) throws UnknownHostException, IOException`
  - checks validity of the IP, port, and username, then creates a user and sets the user's GUI to be gui
- `public User getUser();`
  - returns the User
- `public User runUser();`
  - logs on the user and runs its main()
- `public static void main(String[] args);`
  - makes a new ChatClient()

ClientThread: Thread run for each new client

`public class ClientThread extends Thread`

- `public ClientThread(Socket socket, User user)`
  - `Socket socket`
    - `the Socket to connect the thread to`
  - `User user`
    - `the User that is running this thread`
- `public void run();`
  - runs handleConnection, closes the socket when it's done

User: Object representing user of a client

`public class User`

- `public User(String username, Color color, Socket socket);`
  - `private String username;`
    - a String representing the User's name
  - `private String color;`
    - the color selected from the choices on the Login GUI
  - `private Socket socket;`
    - the Socket associated with this User
- `private UserGUI gui;`
  - the UserGUI that's associated with and running parallel to the User

- `private LoginView loginView;`
    - the LoginView that the User used to log in
- `private ConversationView convoView;`
    - the ConversationView that the User uses to display its Conversations, initially null but is set once the ConversationView is opened (i.e. when the Client gets confirmation that the username is valid)
- `private ConcurrentHashMap<String, UserInfo> onlineUsers;`
    - maps username strings to UserInfo object containing info about all other online users.
- `private ConcurrentHashMap<String, Conversation> myConvos;`
    - maps convoID strings to Conversation object for all active conversations this User is part of.
- `private ConcurrentHashMap<String, Conversation> inactiveConvos;`
    - maps convoID strings to Conversation object for all inactive conversations this User is part of, so that they can be retrieved if the conversation is reopened.
- `private PrintWriter out;`
    - PrintWriter to communicate with the server
- `public void handleConnection() throws IOException;`
    - handle connection to the server, using its this.socket
    - calls handleRequest()
    - throws IOException if connection has an error or terminates unexpectedly
- `private void handleRequest(String input);`
    - handles server messages. updates conversations, etc based on the message received.
    - param input, the input from the server, from the grammar
    - returns the message (from the grammar) to the client
- `private void updateConvo(String input);`
    - takes the server message string of the format "-c …. -u … -t …" and parses it into a convo ID (the part after the -c), a username (the part after the -u) and the message's text (the part after the -t), then adds the text as a Message object to the User's Conversation with the correct convo ID. also updates the tab color in the GUI to be the sender's color.
- `public void startConvo(Conversation convo);`
    - calls sendMessageToServer("-s" + convo.convo_id)
- `public void closeConvo(Conversation convo);`
    - calls sendMessageToServer("-x" + convo.convo_id)

- `public void addMsgToConvo(Conversation convo, String text);`
    - calls sendMessageToServer on the text from the message, the convo_id, and the username, according to the grammar.
- `public void login();`
    - calls sendMessageToServer("-l" + username + " " + color)
- `public void quit();`
    - calls sendMessageToServer("-q" + this.username) and closes its socket's connection.
- `private void sendMessageToServer(String text);`
    - sends text to the server, using this.socket.
- `protected void setOnlineUsers(ConcurrentHashMap<String,UserInfo> userMap);`
    - replaces the onlineUsers ConcurrentHashMap with userMap.
- `private void addOnlineUser(UserInfo user);`
    - adds user to the onlineUsers ConcurrentHashMap.
- `private void removeOnlineUser(UserInfo user);`
    - removes user from the onlineUsers ConcurrentHashMap.
- `public String getUsername();`
    - returns the String representing the User's username
- `public Color getColor();`
    - returns the Color representing the User's color
- `public Socket getSocket();`
    - returns the Socket representing the User's socket
- `public ConcurrentHashMap<String, UserInfo> getOnlineUsers();`
    - returns the ConcurrentHashMap mapping usernames of all online users to their UserInfo objects
- `public ConcurrentHashMap<String, Conversation> getMyConvos();`
    - return the ConcurrentHashMap mapping convoID's to Conversations
- `private void addNewMyConvo(Conversation convo);`
    - adds a newly created conversation convo, using an inactiveConvo if it was already previously created
- `private void removeMyConvo(Conversation convo1);`
    - removes a Conversation that has been closed from the GUI and from the tabs in the GUI
- `private void checkDuplicateConvo(String convoID)`
    - checks a potential new convoID to make sure duplicates aren't created

- `public void setLoginView(LoginView login);`
  - sets the loginView attribute to be login, the LoginView associated with the User's login screen. Useful when it reopens the login window after logging out.
- `public void setConversationView(ConversationView view);`
  - sets the convoView attribute to be view, the ConversationView associated with the User's Conversations
- `public void setGUI(UserGUI gui1);`
  - sets the gui attribute to be gui1, the UserGUI associated with the User
- `public void main() throws IOException;`
  - makes a new ClientThread, and run it on the User's socket.

Conversation: Object representing a Conversation between 2 or more Users. The participants cannot be modified after it is created (i.e. no joining the Conversation after it starts).

`public class Conversation`

- `public Conversation(ConcurrentHashMap<String, UserInfo> participants);`
  - `String convoID;`
    - Identifier for conversation
    - Format: usernames of conversation participants in alphabetical order, separated by spaces.
  - `ConcurrentHashMap<String, UserInfo> participants;`
    - ConcurrentHashMap mapping usernames to UserInfo for all Users who are participating in the conversation
  - `List<Message> history;`
    - ArrayList of all messages in the conversation
- `public void addMessage(Message message);`
  - adds message to the history.
- `public List<Message> getMessages();`
  - returns the history
- `public String getConvoID();`
  - returns the convoID
- `public ConcurrentHashMap<String, UserInfo> getParticipantsMap();`
  - returns the HashMap of participants to their UserInfo.

- `protected String alphabetizeHashMap(ConcurrentHashMap<String, UserInfo> participants);`
    - alphabetizes the participants of a Conversation to get their convoID, using the keys of participants. return the convoID

Message: Object representing an instant message from one User to one or more other Users.

`public class Message`

- `public Message(UserInfo sender, Conversation convo, String text);`
    - `private final UserInfo sender;`
        - the sender of the message
    - `private final Conversation convo;`
        - the conversation the message is part of
    - `private final String text;`
        - the text of the message
- `public UserInfo getSender();`
    - returns sender of message
- `public Conversation getConvo();`
    - returns the Conversation
- `public String getText();`
    - returns the Message's text

UserInfo: Object to hold information about other users, specifically their username, desired color, and the Socket through which to reach them.
First constructor is what the ChatServer class uses for its infoMap, and the second is for Users to keep track of other Users' names and colors.

`public class UserInfo`

- `public class UserInfo(String username, String color, Socket socket);`
- `public class UserInfo(String username, String color);`
    - `private final String username;`

        - the username of the user
    - `private final String color;`
        - the color associated with the user
    - `private final Socket socket;`
        - the socket used by the user
- `public String getUsername();`
    - returns the username
- `public String getColor()`
    - returns the color
- `public Socket getSocket()`
    - returns the Socket

Server: Chat server runner.

`public class Server`

- `public static void main()`
    - starts a chat server

ChatServer: Runs the Server and maintains a ServerSocket to communicate with all Clients.

`public class ChatServer`

- `public ChatServer(int port) throws IOException;`
    - `private ServerSocket serverSocket;`
        - the socket that the server runs on, using port as its port
- `protected ConcurrentHashMap<String, UserInfo> infoMap;`
    - maps String usernames to UserInfo objects which have a record of the user's relevant information
    - uses a threadsafe HashMap
- `protected void serve() throws IOException;`
    - runs the server, listening for client connections and handling them. never returns unless an exception is thrown.
    - uses a ChatServerThread class that extends runnable and calls handleConnection in its run() method to make a new thread each time a new client connects.

    - throws IOException if the main server socket is broken.
- `protected void handleConnection(Socket socket) throws IOException;`
    - handles a single client connection. Returns when client disconnects. Calls handleClientRequest() using socket as its Socket
    - sends messages to clients based on the ServerMessages returned by handleClientRequest()
    - throws IOException if connection has an error or terminates unexpectedly
- `protected ArrayList<ServerMessage> handleClientRequest(String input, Socket socket) throws IOException;`
    - handler for client input. makes requested mutations and returns appropriate ServerMessages (based on client's message, input) which need to be returned to clients.
    - returns a List<ServerMessage> containing all messages and clients which need to receive them
- `protected ArrayList<ServerMessage> addUser(String username, String color, Socket socket)`
    - adds a user to the infoMap unless it is a duplicate username, in which case it sends a INVALID_USER message
    - returns INVALID_USER message to the User trying to connect if username already in use, ONLINE_USERS message to the User and ONLINE message to all other Users if username is valid
- `private ArrayList<ServerMessage> logout(String username)`
    - deletes a user from the infoMap.
    - returns OFFLINE message to all other online users
- `private ArrayList<ServerMessage> updateConvo(String message)`
    - called when the server receives a ADD_MSG message.
    - returns UPDATE message to everyone in convo but me
- `private ArrayList<ServerMessage> startConvo(String message)`
    - returns a START_CONVO message to everyone in the convo_id flag section of message, excluding the sender User itself.
- `private ArrayList<ServerMessage> closeConvo(String message)`
    - return CLOSE_CONVO message to everone in convo_id flag section of message, excluding the sender User itself.
- `protected ArrayList<Socket> justMe(String username);`
    - makes an ArrayList of sockets to send to if just returning to same client (User named username)

    - returns ArrayList containing socket just of that username
- `private ArrayList<Socket> everyoneButMe(String username);`
    - makes an ArrayList of sockets for everyone else but the User named username
    - returns ArrayList with sockets for everyone else
- `private ArrayList<Socket> everyoneInConvoButMe(String convoID, String username)`
    - makes an ArrayList of sockets for everyone else in conversation (founds using convoID) but the User named username
    - returns ArrayList withs sockets of everyone relevant
- `public ConcurrentHashMap<String, UserInfo> getInfoMap()`
    - returns infoMap
- `public ServerSocket getServerSocket()`
    - returns serverSocket
- `public static void runChatServer(int port) throws IOException`
    - makes a server on port and runs serve()
- `public static void main(String[] args)`
    - Calls runChatServer on the port found by parsing args, which should be of the form “-p xxx” where xxx is an integer in the correct port range

ChatServerThread: Creates a new thread to run the server.

```
public class ChatServerThread extends Thread
```

- `public ClientThread(Socket socket, ChatServer server)`
    - `Socket socket`
        - `the Socket to connect the thread to`
    - `ChatServer server`
        - `the server that is running this thread`
- `public void run();`
    - runs handleConnection, closes the socket when it’s done

ServerMessage: Object to handle sending a message to multiple clients.

```
public class ServerMessage
```

- `public class ServerMessage(ArrayList<Socket> recipients, String text);`
    - `private final ArrayList<Socket> recipients;`
        - a list of Sockets to send the message to
    - `private final String text;`
        - the message to be sent
    - `private final ChatServer server;`
        - the server that the message will eventually be sent using
- `public ArrayList<Socket> getRecipients()`
    - returns the recipients of the message
- `public String getText()`
    - returns the text of the message
- `public String toString()`
    - returns a readable String representation of the ServerMessage

UserGUI: Initializes the gui, and manages transitions between the login and conversation views.

`public class UserGUI extends JFrame`

- `public UserGUI(ChatClient client)`
- `private final ChatClient client`
    - the client associated with the GUI
- `private final LoginView login`
    - the view that is displayed for the client to attempt to log in
- `private final ConversationView convo`
    - the view that is displayed for the client after login, where they can create new Conversations. initially null, but set once login is successful.
- `public void openConversationView()`
    - closes the current LoginView and opens a new ConversationView()
- `public void setUserView()`
    - sets the User's login view to be the Client's login view
- `public void setLoginView(LoginView view)`
    - sets login to be view
- `public void setConvoView(ConversationView view)`
    - sets convo to be view

- `public void getClient()`
    - returns client
- `public void getUser()`
    - returns client's User

LoginView: Creates the login view that initiates a session for a user

`public class LoginView extends JPanel;`

- `private final JTextField ipAddress;`
- `private final JTextField portNumber;`
- `private final JTextField username;`
- `private final JComboBox colorDropDown;`
- `private final JButton submitButton;`
- `private final JLabel hermes;`
- `private final JLabel messenger;`
- `public LoginView(final UserGUI gui)`
    - Makes all the formatting for the login screen (background color, title, textfield arrangement, etc)
    - Adds listeners for:
        - Clicking away from a text field - if nothing has been typed, the abandoned text field will display the title text (i.e. "IP Address").
        - Typing in a text field - if it said the title text, it will disappear once a key is typed, otherwise it will let the user add on to whatever the box previously said
        - Clicking enter - calls Client.attemptLogin() and Client.runUser() to attempt to log the user on
- `public void close()`
    - closes the view and sets its visibility to false for good measure!
- `public static void main(final UserGUI gui)`
    - Makes a new LoginView and sets gui's LoginView to be it

ConversationView: Represents the user's conversations as a window with tabs for each conversation and ways to logout, start a new conversation, send messages, etc.

`public class ConversationView extends JPanel;`

- `private static JFrame frame;`
- `private JTabbedPane tabby;`
- `private static JMenuBar menuBar;`
- `private final JMenu file;`
- `private final JMenuItem logout;`
- `private final JMenuItem closeConvo;`
- `private DefaultListModel listModel;`
- `private JList list;`
- `private ConcurrentHashMap<String, TabPanel> tabMap = new ConcurrentHashMap<String, TabPanel>();`
- `public final static ConcurrentHashMap<String, Color> colorMap = new ConcurrentHashMap<String, Color>();`
- `public final UserGUI gui;`
- `public ConversationView(final UserGUI gui)`
  - makes colorMap, makes the conversation window, and makes a tab panel
- `public String parseConvoID(String convoID)`
  - parses convoID to remove own name for display on tab
- `private JComponent newConvoPanel()`
  - creates a panel for starting a new conversation
- `public void updateOnlineUsers()`
  - Updates panel containing all online users, called when a new user logs on
- `public void updateTabs()`
  - adds tabs for new conversations
- `public void updateTab(String convoID)`
  - updates the tab of a conversation when it gets a message
- `public void removeTab(String convoID)`
  - removes the tab of a closed conversation, indicated by convoID
- `public void fillHistory(String convoID)`
  - fills the conversation history in the tab containing the Conversation indicated by convoID
- `private static void createAndShowGUI()`
  - makes the window and ConversationView and displays them
- `public void close()`
  - closes the view and sets its visibility to false for good measure!
- `public static void main(final UserGUI gui)`
  - calls createAndShowGUI on gui

TabPanel: Represents each tab’s layout on the Conversation GUI.

`public class TabPanel extends JPanel;`

- `private Conversation convo;`
- `private DefaultListModel historyModel;`
- `private JScrollPane historyScroll;`
- `private JList history;`
- `private final User user;`
- `public TabPanel(Conversation convo, User user)`
  - `set the Conversation and User associated with the TabPanel`
- `private void makePanel()`
  - `makes a panel for a conversation, including the send button, the text field for the message, and the display of the message history`
- `public Conversation getConvo()`
  - `returns convo`
- `public void showMessage()`
  - `adds a new message on the bottom of the tab (when it is already open)`
- `public void fillHistory()`
  - `fills the historyModel with all the messages from the conversation (for when user has just clicked to the tab)`
- `private class MessageRenderer extends DefaultListCellRenderer`
  - `makes the messages the color of the sender`

DuplicateConvoException: Is thrown when User.startConvo is called with a convoID that already exists.

InvalidUsernameException: Is thrown when the username is invalid, i.e. it is more than 10 characters, contains anything other than letters, or is already taken by someone else

Client/Server Protocol
Client to server:
MESSAGE :== [ADD_MSG START_CONVO CLOSE_CONVO LOGIN QUIT] “\n”
LOGIN :== -l USER_INFO
QUIT :== -q USERNAME
ADD_MSG :== -c CONVO_ID -u USERNAME -t TEXT //-u is the sender
START_CONVO :== -s CONVO_ID -u USERNAME
CLOSE_CONVO :== -x CONVO_ID -u USERNAME
CONVO_ID :== USERNAME+   //alphabetized
USER_INFO :== USERNAME COLOR
USERNAME :== [a-zA-Z]{1,10} //10 character limit, no numbers
COLOR :== [“red” “orange” “yellow” “green” “blue” “pink”]
TEXT :== [^NEWLINE]+
NEWLINE :== “\r?\n”

Server to client:
MESSAGE :== [START_CONVO UPDATE CLOSE_CONVO ONLINE OFFLINE INVALID_USER ONLINE_USERS] “\n”
ONLINE_USERS :== -f (USER_INFO)* //list of all online users’ names and colors
INVALID_USER :== -i USERNAME //error sent if username is already in use
ONLINE :== -o USER_INFO //lets clients know a new user is online
OFFLINE :== -q USERNAME //lets clients know a user has logged off
START_CONVO :== -s CONVO_ID -u USERNAME//sent to recipient when new convo started
CLOSE_CONVO :== -x CONVO_ID -u USERNAME //sent to other participant when one person exits
UPDATE :== -c CONVO_ID -u USERNAME -t TEXT //-u is the sender
CONVO_ID :== USERNAME+   //alphabetized
USER_INFO :== USERNAME COLOR
USERNAME :== [a-zA-Z]{1,10} //10 character limit, no numbers
COLOR :== [“red” “orange” “yellow” “green” “blue” “pink”]
TEXT :== [^NEWLINE]+
NEWLINE :== “\r?\n”

Concurrency Strategy

Server

- Maintains a ConcurrentHashMap of client usernames to client userInfo objects which contain the client’s username, color, and socket

    - ConcurrentHashMap allows for synchronization in the case of searching for an element which is not there (once the item is not found, the map is synchronized, the item is looked for again (in case it was added before the synchronization) then the item is added)
    - For most other methods, the map is not needlessly locked - only the individual buckets are locked by threads accessing them - this eliminates needless overhead encountered when synchronizing a regular HashMap
    - Iteration through a ConcurrentHashMap ensures any updates or removals of items will be correctly reflected in their behavior
    - Performs better than a HashMap with synchronization under load which ensures our system is scalable while maintaining thread safety
- Every function that reads from or writes to a socket first synchronizes the socket - this eliminates race conditions such as the possibility of sending a message to participants in the conversation as a participant is signing off

Client

- Handling the client's connection happens in two steps to eliminate duplicate username issues:
    - the client connects via a socket and requests to login with a desired username and color
        - if the server sends a message that the username is taken, the socket is closed and can try to login again from the login screen
        - if a message with the current online users is reported, the client adds these usernames to its record of online users (a ConcurrentHashMap of string usernames to userInfo objects - the ConcurrentHashMap eliminates potential concurrency issues from the user's connection handler and the GUI which run on separate threads), then the User is created and a new ConversationView is opened
- The synchronization of Conversation convo of all methods that access (to prevent display inconsistencies) and mutate the convo (by adding messages) ensures the integrity of a user's Conversations
- The client system is free deadlock because no cycles can exist since we are only synchronizing Conversations which do not access other Conversations

Testing Strategy

<u>JUNIT Method Tests:</u>

ChatClient.java

- Method: `ChatClient()`
    - Strategy: cannot be tested effectively with unit test since makes GUI and

    - other stuff, so will test in System test
- Method: `setUser(String username, String color, Socket socket) {`
    - Strategy: will test with mock socket instance and check that the user is set with the correct info
- Method: `attemptLogin(String IP, String port, String username, String color)`
    - Integration testing with Server - run ChatServer and create ChatClient and try to call this method with both a valid and invalid username (partitioning input space)

ClientThread.java

- Cannot be reasonably tested with automatic tests - tested by the concurrent features working correctly for running a ChatClient

Conversation.java

- Method: `Conversation(ConcurrentHashMap<String,UserInfo> participants)`
    - Stategy: create new Conversation with mock participants map, assert the instance variables are set correctly
- Method: `addMessage(Message msg)`
    - Strategy: add a message to a Conversation and access last message to verify it matches
- Method: `getMessages()`
    - Strategy: get messages from a Conversation
- Method: `getConvoID()`
    - Strategy: make sure a convoID is returned as expected
- Method: `alphabetizeHashMap(ConcurrentHashMap<String, UserInfo> participants)`
    - Strategy: partition input space: two participants with same first letter(s) (at least one letter must eventually be different since we use it for chat users who must have unique names), test adding person with later letter to particpants first then adding the alphabetically first person second to participants, test adding participants in alphabetical order then running the method, similarly test with mixture of capital letters bearing in mind all capital letters come before any lowercase letters

ConversationView.java

- GUI is checked visually in the system tests (described below).

GUIThread.java

- GUI is checked visually in the system tests (described below).

LoginView.java

- GUI is checked visually in the system tests (described below).

Message.java

- Method: `Message(UserInfo sender, Conversation convo, String text)`
  - Strategy: make sure things are set correctly
- Method: `getSender()`
  - Strategy: make sure sender is returned
- Method: `getConvo()`
  - Strategy: make sure the Conversation being returned is the same instance as the one the Message was made with
- Method: `getText()`
  - Strategy: make sure correct text returned

TabPanel.java

Since this is a custom JComponent, it is difficult to test in isolation and will be tested in Integration testing with ConversationView. We will make sure the appropriate conversations show up in the panels and are updated properly.

User.java

- Method: `User(String username1, String color1, Socket socket1)`
  - Strategy: test creation of a User with mock socket and ensure things are set correctly
- Method: `main()`
  - Strategy: since main creates a new thread for the client connection, it is difficult to test in isolation; therefore, this will be tested via the System test and with Integration tests with the Server
- Method: `handleConnection(Socket socket)`
  - Strategy: Can't be tested with unit tests because of required socket connection. Has been visually verified to work because the connection between Client and Server is working, and all of the incoming messages from the server are being processed and making the appropriate changes in the GUI.
- Method: `handleRequest(String input)`
  - Strategy: Also can't be unit tested because of the required connection to a server. Visually tested by manually stepping through actions to induce all possible messages from the server (listed in the Client/Server protocol documentation). These incoming messages were successfully received (verified through print statements), and were processed correctly, making the desired changes in the GUI.

- Method: `sendMessageToServer(String text)`
  - Strategy: Can't be unit tested without a server. Visually tested by printing incoming messages on the server. Actions on the client are properly composed into outgoing messages and are successfully received by the server.
- Method: `startConvo(Object[] usernames)`
  - Strategy: Can't be unit tested without a server. Visually tested by successfully initializing conversations, and printing MyConvos, which keeps track of all of the users active conversations.
- Method: `closeConvo(Conversation convo)`
  - Strategy: Can't be unit tested without a server. Visually tested by successfully closing conversations, and printing MyConvos, which keeps track of all of the users active conversations.
- Method: `addMsgToConvo(Conversation convo, String text)`
  - Strategy: Can't be tested without a server. Visually tested by sending a message to another user. Visually tested by GUI updating with new message, and the correct message being received by the server.
- Method: `updateConvo(String input)`
  - Strategy: Can't be tested without a server. Visually tested. The incoming message from the server is correctly parsed, added to the Conversation, and displayed in the GUI.
- Method: `login()`
  - Strategy: Can't be tested without a server. Testing using print statements, the login message is properly created, sent, and received by the server.
- Method: `quit()`
  - Strategy: Can't be tested without a server. Visually tested. The login view is opened, all the user's conversations are closed, and the quit message is correctly generated, sent, and received by the server.
- Method: `setOnlineUsers(ConcurrentHashMap<String,UserInfo> userMap)`
  - Strategy: make sure correct users are set
- Method: `addOnlineUser(UserInfo user)`
  - Strategy: ensure users added correctly
- Method: `removeOnlineUser(String user)`
  - Strategy: test that user can be removed from online user list properly
- Method: `getUsername()`
  - Strategy: assert correct username returned

- Method: `getColor()`
    - Strategy: assert correct color returned
- Method: `getSocket()`
    - Strategy: assert correct socket instance returned
- Method: `getOnlineUsers()`
    - Strategy: assert that online users returned
- Method: `getMyConvos()`
    - Strategy: assert correct conversations returned
- Method: `addNewMyConvo(Conversation convo)`
    - Strategy: Can't be tested without GUI. Tested visually. Conversations are properly created and added to MyConvos. The GUI is properly updated with the new tab, and the content is filled in the tab.
- Method: `removeMyConvo(Conversation convo)`
    - Strategy: Can't be tested without GUI. Tested visually. Conversations are properly removed from MyConvos, and added to the inactiveConvos. The GUI responds correctly, removing the conversation tab, and saving the history in inactiveConvos so it can be viewed later.
- Method: `checkDuplicateConvo(Conversation convo)`
    - Strategy: Can't be tested without GUI. Tested visually. Attempting to create a convo that already exists, and the appropriate exception in thrown.

ChatServer.java

- Method: `addUser(String, String, Socket)`
    - Strategy: Make a mock socket, input a username string and color, and see if the correct ServerMessage arrayList is returned
- Method: `closeConvo(String)`
    - Strategy: input a valid CLOSE_CONVO message and see if it returns CLOSE_CONVO messages for all other users involved in the convo (tested with real sockets in ChatServerNoDiditTest)
- Method: `everyoneButMe(String)`
    - Strategy: input a username String and see if all other “logged on” users have a ServerMessage returned - should be null. Also will need to be tested in an integration / system test.
        - tested in conjunction with login test in ChatServerNoDiditTest
- Method: `everyoneInConvoButMe(String convoId, String username)`
    - Strategy: test in conjunction with send message test by making sockets and connecting them and making sure other sockets get the right message and the sender doesn’t - in ChatServerNoDiditTest

- Method: `getInfoMap()`
    - Strategy: make mock infoMap, set, and make sure it is returned
- Method: `closeConvo(String)`
    - Strategy: make sure appropriate message sent to socket outputstream of appropriate sockets
- Method: `handleClientRequest(String input, Socket socket)`
    - Strategy: Partition the input space and check that appropriate response returned for all testable input
- Method: `handleConnection(Socket socket)`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test
- Method: `justMe(String)`
    - Strategy: ensure socket returned corresponds to username given as input
- Method: `logout(String)`
    - Strategy: ensure that username is properly removed from the infoMap, and that the ServerMessage returned contains the recipients, flag, and username
- Method: `main(String[] args)`
    - Strategy: difficult to test with unit test, but tested in System test
- Method: `runChatServer(int)`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test
- Method: `startConvo(String)`
    - Strategy: Ensure the correct ServerMessage is generated including correct recipients and message
- Method: `updateConvo(String message)`
    - Strategy: unit test to make sure correct messages returned to be sent to right sockets
- Method: `serve()`
    - Strategy: Cannot be tested with unit tests effectively but tested in System test

ServerMessage.java

- Method: `getRecipients()`
    - Strategy: Make a ServerMessage object and make sure it returns the same Sockets you put in
- Method: `ServerMessage(ArrayList<Socket>, String)`
    - Strategy: Make various ServerMessages with mock Sockets and text Strings. Mostly to check for no errors.
- Method: `getText()`
    - Strategy: Make a ServerMessage object and make sure it returns the correct text

- Method: `toString()`
  - Strategy: Make a ServerMessage object and call toString on it, make sure it prints reasonably and in a user-friendly way

System (Integration) Tests:

JUNIT

- Tests running the server and one or more clients
- We will test all possible interactions between the server and client, which are detailed in the “Client/Server Protocol” section below, and ensure that the server and client respond with the appropriate messages dictated by our protocol.
- In the case where these messages correspond to updates in the GUI, we will verify the appropriate changes visually, as described below.

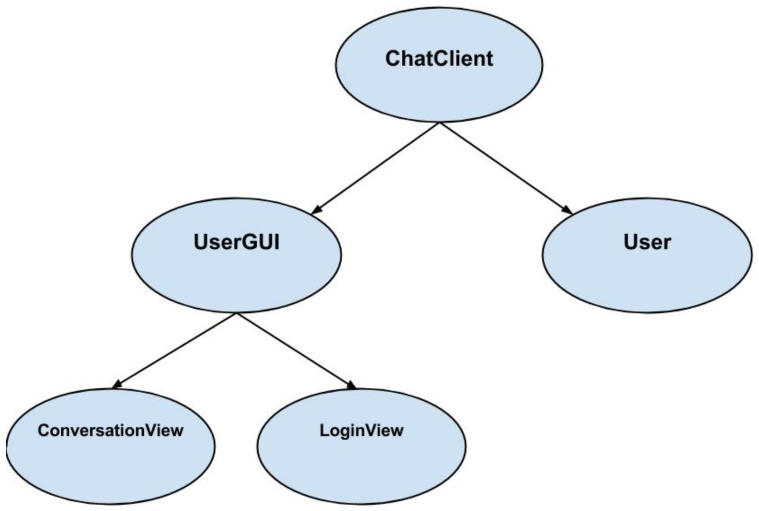
Visual Testing Report

- In addition to the JUNIT tests, we will visually test the GUI to ensure the user experience matches our specifications. Above in the User Experience section we have outlined the behaviour of our GUI in response to any inputs from the User. We will perform User testing on our GUI by stepping through all viable combinations of user inputs to ensure the behavior matches what we expect.
- **ConversationView**
  - The conversation view is where the majority of user interaction goes on. The obvious check was the GUI was properly laid out, containing all the correct components. The actions for each possible user interaction with on-screen components are documented above in the “User Experience” section. We verified that all of these actions are correct by printing out the messages being passed between the server and client and checking if they were correct. We did this for all messages in our Client/Server protocol which includes logging in, logging out, sending a message, starting a conversation, and closing a conversation. We tested this for all normal uses and edge cases, such as trying to start a conversation when no users are selected (or no one else is online), and various combinations of closing/opening/checking history of conversations. We checked concurrency by running clients on different computers and simultaneously logging on with the same username, logging off, starting conversations simultaneously, and sending messages simultaneously. All of the behaviors remained normal in all these cases.
- **LoginView**
  - The login view creates the window for user’s to enter credentials to log on onto the chat server. The constructor is easily verified to be correct because

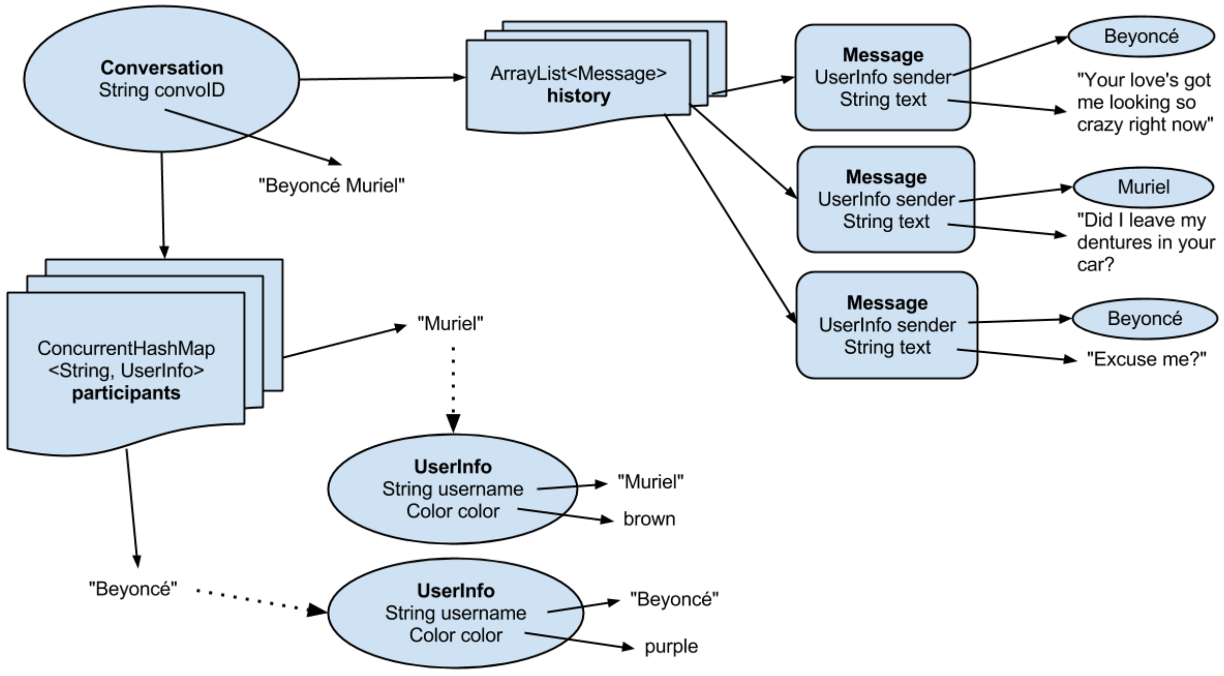
the components are correctly displayed in the window and displayed correctly. This includes the title text, the text fields to enter IP address, port number, username, and color. The only user action in this view is to enter information and submit. Entering valid information will correctly login the user (previously verified by printing the messages passed between server and client), and call to UserGUI that opens the conversation view is successful. If any of the credentials is incorrect, a dialog pop up appears explaining the error and returns the user to the login screen to try again.

- **TabPanel**
  - Our visual tests for TabPanel ensure that the panel is correctly generated and that the methods that fill in the conversation history do so accurately. Multiple open tabs, multiple user conversations, and repeatedly opening and closing conversations were all tested.
  - Method: `makePanel()`
    - Strategy: Tested visually. The panel contains the correct components that are described in the user experience section of our design documentation. The panel appears formatted correctly and the components all react with the appropriate actions when interacted with by the user.
  - Method: `showMessage()`
    - Tested visually. This method is called each time a new message is sent or received and must display in the conversation history panel. We ensured that the message is always correctly added including when multiple tabs are open.
  - Method: `fillHistory()`
    - Tested visually. Fill history is called when a conversation is re-opened and the past chat history must be displayed. The visual testing involved closing and reopening conversations between two users and more users, and verifying that the past chat history was accurately reproduced.
- **UserGUI**
  - UserGUI is in charge of opening and closing the appropriate windows. This was visually tested by running the client. This opens the login window. Entering credentials to log in will close the login view and open the conversation view. Hitting the logout button in the conversation view will close the conversation view and open the login view. These transitions all worked properly.

Dependency Diagram



Snapshot Diagram


Conversation
String convoID
"Beyoncé Muriel"
ArrayList<Message>
history
Message
UserInfo sender
String text
Beyoncé
"Your love's got me looking so crazy right now"
Message
UserInfo sender
String text
Muriel
"Did I leave my dentures in your car?
Message
UserInfo sender
String text
Beyoncé
"Excuse me?"
ConcurrentHashMap
<String, UserInfo>
participants
"Muriel"
UserInfo
String username
Color color
"Muriel"
brown
"Beyoncé"
UserInfo
String username
Color color
"Beyoncé"
purple