Changes to Original Design:

Mostly the changes from our original design were relatively minor. One major change was the addition of a list of online users in the main window, which appears after logging in. We had originally not planned out where to display this list of users, and how we would interact with the list to initiate conversations. Now, a user can select and then click IM to initiate a 2way conversation with a logged in user, or they can double click on a user on that list.

Also, we made some minor additions to our communications package. In particular, the RealChannelFactory and TestChannelFactory classes were added to make it easier to generate new channel connections on the client side, after a connection session is terminated, and the user wants to login again.

Other minor GUI changes included the addition of a ChatRoomNameDialog, when joining or creating a new chat room. It proved easier to have a separate window for this task, than to update the appearance of a chat room window after typing in the name of a chat room. Overall, the basic structure of our code closely followed our original design. The basic multi threading used on the server side to receive multiple client connections was unchanged, and the model-view-controller structure of the client side was also not changed in any major structural way.

Testing Report:

Our testing strategy was divided into two phases.

Phase 1: Unit Tests

One module tests the the communications package, in particular, whether messages are sent to and from TestChannel objects properly. This unit test creates two instances of TestChannel, connects them to each other, and then attempts to send a message from one channel to the other. After this, one channel is closed, and we check to make sure that exceptions are properly thrown if communication is now attempted between the channels. Only TestChannel objects are tested here, since a real channel requires a network connection, with a socket, and this is somewhat beyond the scope of basic unit testing. The real versions of these were tested extensively by hand, as they are used to send chat messages over the network.

Next, we wrote tests for our ClientController and ClientModel. These tests need to ensure that the proper messages are being sent to the server, and that the ClientModel is being properly adjusted in different cases. These cases include login, logout, receiving a message, starting a conversation, joining a chat room, closing a chat window, and other actions. In the case of a message being received, the Controller passes the message to the model to handle. In the case of a join chat room request, the controller creates and send the appropriate request to the server. In the case of sending a chat message, the Controller generates the appropriate packet, and sends it over the server channel. Other tests were similar in nature, and extensively explore the workings of the ClientController.

Lastly, we wrote tests for the Server. For these tests, we used instances of TestChannel, connected to the server, to determine if the correct messages were being sent from the server to various users. In particular, there are different cases for logging in, logging out, sending a chat message, or performing operations involving a chat room. When logging in, a user expects to receive an Ack packet, which contains a status that informs them whether or not they have chosen a valid username. Other users expect notification that the user has logged in. Logging out should send the appropriate notification to other logged on users. When sending 2 way chat messages, the sender expects to receive an Ack packet, and the other user expects to receive the message that was sent. For chat rooms, notifications of joining, leaving, and sending messages to users in the room are all expected. Writing these tests was effective in showing us possible race conditions due to code that was ordered improperly. An example would be receiving the list of users logged in, before receiving a confirmation message that the username is not taken and you have been logged in.

Our unit tests extensively cover the main modules of our code, for both the client and server side, and for the classes that they use to communicate.

Phase 2: GUI Tests

Lastly, we needed to test our GUI, to determine whether the appropriate events occur when the user interacts with the GUI, and to determine whether the GUI is appropriately updated when events occur (users logging in and out, chats received, etc.)

For the MainLoginWindow, we tested:

1. Whether pressing enter or pressing the login button resulted in the appropriate update to the GUI, which includes a display of "Logging in . . ." and disabling the "login" button and the username field.
2. After receiving the correct login ACK from the model, we checked to see that the login window was hidden, and and that the MainWindow was displayed.
3. We also ensured that the MainLoginWindow reset to the correct state after logging out (from the MainWindow): that the username originally entered was in the username field, and that the username field and login buttons were enabled.
4. As a finishing touch, we ensured that the login button (and corresponding "enter" keystroke to login) was only active if text had been typed in the login field.

Next, we needed to test the MainWindow, which shows the list of logged in users and buttons for different actions on the top. For MainWindow, we tested:

1. Updating of "logged in user list" when other users were logged in and logged out.
2. We tested whether selecting users and pressing the IM button properly displayed a new IM window, for a 2way conversation, and also when double clicking on a user in the list.
3. We also tested that pressing the Join Chat Room button displayed the appropriate chat room dialog box (see below).
4. Finally, we tested that pressing logout properly closed the MainWindow, and displayed the login window again (see above).

Next to test was a TwoWayWindow:

1. We first tested that initiating a conversation with another user made a TwoWayWindow appear on their screen. We chose to allow any user to initiate a conversation with another user at any time, without need for confirmation.
2. Next, we tested whether typing a message and pressing enter properly displayed the message on both the local window, and the other users window.
3. Lastly, we tested that closing a TwoWayWindow did not have any affect, other than continuing to display the MainWindow with the currently logged in users.

4. We implemented the "send" button through the same mechanism as the "login" button so that sending was only enabled when text had been pressed in the input text pane.
5. We tested that the correct error messages were displayed when we attempt to send a message to them ("[Username] is offline or unavailable" for logged out users and "Could not send message to [Username]" for a message timing out (8 seconds)).
6. Also, when another user logs out, we tested if the correct buddy logout notification was displayed in the chat history window ("[Username]" has logged out).
7. We checked if the correct formatting was applied and that received messages had the correct username (we used HTML to format usernames in the history box to be bold red and blue, similar to popular chat programs)

Finally, we tested a ChatRoomWindow.

1. First, we tested that a new ChatRoomNameDialog appeared asking us for the chat room name.
2. We tested in the ChatRoomNameDialog window that a name must have been entered to attempt to create or make a chat room with a particular name, and that a chat would be joined if it already existed and that a chat would be created if it did not exit.
3. We tested that the users logged in list was properly updated everytime someone entered or left the current room, by entering and leaving chat rooms with multiple users.
4. As in a TwoWayWindow, we tested whether typing messages and pressing enter or send properly displayed the messages to all users in the chat room, and implemented the same "send" button mechanism that is only active when text has been typed in the box.
5. We tested whether correct user logout and login notifications appeared ("[Username] has joined the room") or ("[Username] has left the room").
6. As in TwoWayWindow, we checked to see if formatting was correctly applied to usernames in the chat history window.

This concluded our GUI tests. All possible actions were tested in our GUI, in the login screen, main window, and two way/chatroom windows, and the appropriate results were found in these cases.

Reflection

This project presented a great challenge, with no code given to us to start with, and little guidance beyond some basic guidelines. We confronted a number of challenges in developing our chat system (KID Chat 1.0), from our GUI to threads in our server.

First, the easy parts: At the start of the project, we settled quickly on using some basic XML text strings as our communication protocol. By choosing to do this, we were able to use an XML parser included with java to take care of most of the parsing work for received XML messages. When messages are parsed, they are returned as a Packet. This was a construct we created to represent one communication, sent between the server and the client. Working with Packet objects was much easier than dealing directly with the XML transmission, and we created a method within Packet to convert the contents of that packet back to an XML string, for easy text based transmission.

Another piece that was not too difficult was setting up classes to deal with communication.

In particular, we created interfaces called Channel and ConnectionHandler. In the "Real" implementations of these, we deal directly with sockets and server sockets, and in the "Test" implementations, we avoid using the network completely. Setting up this abstraction layer, through which we could send packets without dealing with the underlying sockets, was not too difficult and proved very useful, both in testing and in implementation.

After this, we started to confront the harder parts of the project. On the server side, it took a bit of time to properly set up the interaction between the different user threads, and the main queue handling thread, to which all messages were passed. After some experimentation, we settled on the use of a BlockingQueue, which is set up to properly handle access from multiple threads, and to wait properly for something to appear in the queue when attempting to read from it. Another challenging part of the server was properly using a synchronized map to store the map of logged in users, since multiple user threads could be attempting to access this at the same time.

There were many difficult parts to the client side of this project. In general, getting the GUI setup, and working properly with the controller and model was very difficult. We tried to follow the model-view-controller design closely, and it was generally effective. Various listeners in the GUI called methods in the controller to take appropriate actions, which could include updating the model. In turn, changes to the model are updated to the GUI. It was very difficult to keep track of the large number of fields needed to represent the windows in this relatively simple GUI. The basic layout code (generated by a GUI designer) became very lengthy. There was also some difficulties in determining how the model code should process incoming messages, and make the appropriate adjustments to the GUI. In general, the model, view and controller were all tightly coupled, and when coding we needed to be in close proximity to each other, so that we would know which methods to use from other parts of the code.

There were a number of unexpected difficulties when working on this project. On the server side, it was important to correctly handle unexpected losses of connection. In particular, whether or not a user logged off properly, it was necessary to send the appropriate updates to other users logged in, and to remove the user from all chat rooms they were currently in. In the client, the high level of complexity for a seemingly simple GUI was unexpected. It's easy to see how quickly the code for a more complex GUI program, like a word processor, would grow in size and level of confusion.

If we were to design a new chat system, perhaps for a larger scale commercial application, we would likely need to design it to be scalable from the ground up. Right now, our system would not work reasonably for more than 10 or 15 users. A larger system might need to make user of other approaches, like the use of a database to store and lookup user information, or the use of multiple queues on the server side to take care of the very large number of requests. In addition to making a commercial applications scalable, we would need to code it with security in mind, ensuring that users could log in only as their registered username, using a password. The likelihood of some kind of attack on usernames would grow as the system grows, so some kind of protection like this would be necessary. Many additional features could be added to our chat system, ranging from video chat to file transfers, and more, and if we were to do this again, we'd probably pick a feature like this, and plan out how to include it from the beginning, instead of trying to stick it in somewhere after finishing some basic coding.

The work was distributed well in our group across the different parts of the project. It was ultimately very satisfying to see the different parts of our code come together, and to watch clients communicating properly over the server, all updating in a well formatted GUI.