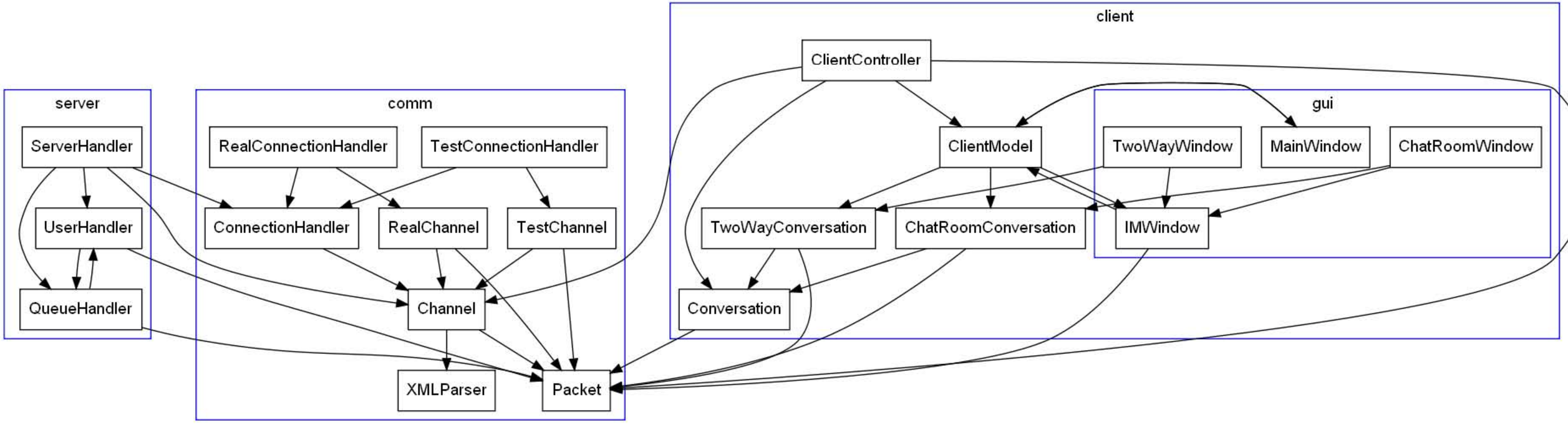
Code Design

Dependency diagram:
(see next page)


client
ClientController
gui
ClientModel
TwoWayWindow
MainWindow
ChatRoomWindow
TwoWayConversation
ChatRoomConversation
IMWindow
Conversation
server
ServerHandler
UserHandler
QueueHandler
comm
RealConnectionHandler
TestConnectionHandler
ConnectionHandler
RealChannel
TestChannel
Channel
XMLParser
Packet

As shown in the diagram above, code is broken up into four main packages:

Main:
The actual server and client applications are started up from the **Client** and **Server** classes. These classes contain some initialization code, such as specifying whether to use real or simulated communications channels (see the **comm** package below).

Client:
The client consists of main client driver classes and a subgroup of GUI classes. This package follows the MVC design pattern for separating graphical presentation, flow of control, and state management. The GUI classes (**MainWindow** and **IMWindow**) manage the view, the **ClientModel** stores the state and triggers GUI updates when appropriate, and the **ClientController** manages the interaction between them: whenever a relevant GUI event occurs, the GUI notifies the controller, which manages any server requests and updates the model. **ClientController** also processes incoming message events. The model will alert the GUI to changes following a modified version of the Publish-Subscribe design pattern: one window will be considered the "subscriber" to each conversation's events, and the main window will be considered the subscriber to global events. (Some global events, such as user signoff, may be relevant to individual Conversations as well as the main window.) Whenever an event occurs that is relevant to a subscriber, the model will call a handler method on the

The **ClientModel** class will contain two maps: one from name to **TwoWayConversation** instances and one map from name to **ChatRoomConversation** instances. **TwoWayConversation** and **ChatRoomConversation** will store chat histories as collections of **Packet**s, which are described in the *comm* package below. (Both are subclasses of the abstract **Conversation** class.) **ChatRoomConversation** will also maintain a room name and a collection of usernames that represents the users currently in the corresponding chat room, while **TwoWayConversation** will contain a username that represents the other user involved in the conversation. **ClientModel** will also maintain a list of currently logged in users, to be updated by regular notifications from the server.

Upon instruction from the controller, **ClientModel** also updates our GUI window classes, which are as follows: **MainWindow**, which displays buttons for loggin in, starting a new IM conversation, and logging out; **IMWindow**, which is an abstract class of an **IMWindow**, and is extended by **TwoWayWindow**, which displays a button to send a message and to close the window and has a field for typing messages and viewing message history, and **ChatRoomWindow** which contains the same features as **TwoWayWindow** and additionally displays a list of currently online users.

The following events and responses will be handled:

- Login status updates will change the **ClientModel** state and cause the **MainWindow** to change from the login field and button to the logged in view.
- **IMWindow** opening and closing requests will call the **ClientController** to add a new conversation to **ClientModel**'s collection, causing the GUI to display new windows
- Conversation message receiving and message sending acknowledgements will add packets containing message content to **Conversation** history, causing the GUI to display these in the relevant **IMWindow**
- Conversation message sending (before being acknowledged) will call the **ClientController** to change the state of a **Conversation**, such that the **IMWindow** will show the message as being sent but not received (not placed in history)
- Online user notifications will change **ClientModel**'s user list, updating changing the **MainWindow**'s displayed online users
- Disconnection from the server causes the GUI to disable all send buttons, and the main window to revert to a login window. It also causes the **ClientModel** to note that it is now offline.
- Reconnection from a disconnected state causes the **ClientModel** to note that it is now online. It replaces the login window with the main IM window, and re-enables the Send buttons on any open chat windows.

Comm:
This package contains various classes for managing server-client communications. The central class in this package is **Packet**, which represents a single atomic XML message sent from client to server or

server to client. Since it is used to represent all sorts of messages (the types of which are defined in the protocol from the abstract design), every **Packet** object includes a flag indicating what type of XML message it represents. The class defines fields for all possible elements and attributes of an XML message; any given instance will only use those fields relating to its message type.

Packets are constructed programmatically by the server and client. They are then sent over a **Channel** to the other party. **Channel** is an abstract class that handles the details of actually packaging and sending messages. It internally converts Package objects into XML and sends them to another **Channel**, which parses the XML and constructs a **Packet** from it. Much like a Java Socket, every **Channel** can be read from or written to.

There are two types of **Channel**s: A **RealChannel** represents an actual server-client network connection, handled by Java Sockets. A **TestChannel** is a fake channel that simply connects up two objects running in the same JVM, allowing for automated tests to construct objects and pass messages between them as if they were being passed over the network.

The server uses **ConnectionHandler** objects to establish connections with clients. The server is initialized with a **ConnectionHandler** object, which it uses to repeatedly wait for client connections. Whenever the **ConnectionHandler** detects a new connection, it returns a **Channel** object to the server, which can then spawn a new thread with which to communicate with the client on the other end of the returned **Channel**. As with **Channel**s, there are two kinds of **ConnectionHandler**: a **RealConnectionHander** uses a ServerSocket to await network connections on a specific port, and returns a **RealChannel**. A **TestConnectionHandler** awaits programmatically generated client connections, returning a **TestChannel**.

The **comm** package also includes the **XMLParser** class, which does the grunt work of parsing an XML file (using the standard Java API) and processing the resulting DOM object to create a **Packet** object out of it.

Server:
The server consists of three main thread classes: **ServerHandler**, which is the main server thread class, **UserHandler**, an instance of which is created to process messages to and from each user, and **QueueHandler** which performs the routing tasks between **UserHandler**s.

ServerHandler handles the other two server thread objects; it creates a new **UserHandler** thread each time a new user connects. It also creates one **QueueHandler** thread.

UserHandler processes messages from a single client by using the **Packet** class of **comm.**, and passes messages to the **QueueHandler**. **QueueHandler** passes messages back to a **UserHandler** for sending, and **UserHandler** then uses the **Packet** class again to send out a message to its associated client. **UserHandler** associates with **ConnectionHandler** to deal with connections.
QueueHandler has a map from username to **UserHandler** object; it uses this to route a message passed to it from one **UserHandler** to the correct destination **UserHandler**(s) (more than one are used in the case of chat rooms). **QueueHandler** is responsible for keeping track of which **UserHandler** every username is associated with, in addition to keeping track of which users are in each chat room. When receiving any request packets, the **QueueHandler** must make all necessary changes to its internal model, and send out appropriate notifications and acknowledgement packets.

QueueHandler also implements a GUI to display statistics about the current connections and users of the server (since **QueueHandler** is aware of all users that are connected, and about the state of all chat rooms). The GUI displays the server status, the number of users connected, and the people in each chat room.

Testing Strategy

Our testing strategy maximizes the number of automatic tests that can be performed; the two main ways of doing this are

1. a mock communication link in order to test our server and client programmatically without opening network connections, described in the **comm.** package above, and
2. A **ClientController** API that allows us to simulate GUI user input events that change the **ClientModel**

When testing a particular feature, errors will be explicitly created and expected (for instance, a user logging off without sending a notification to the server) in addition to testing correct performance.

Automated Unit Tests:

1. Client tests
    a. Using **ClientController** API, test that every possible action of the controller produces the correct change in **ClientModel** and that correct messages are sent to the server
        i. Login/logout
        ii. IM Window opening
        iii. IM Window receiving
        iv. Conversation message sending
    b. Using **comm** and programmatically inserting messages to simulate the server; checking that correct messages are sent to server and that **ClientModel** correctly updates
        i. Login - correct handling by **ClientModel** when login ACK received
        ii. Conversation message receiving
        iii. Conversation ACK message receiving
        iv. Online user update message (sent when user logs on or off)
2. Server tests
    a. Using **comm** and programmatically inserting messages to simulate clients – testing via monitoring outgoing connection via **comm**
        i. Messages are delivered to correct clients in 2 way and chat conversations
        ii. Correct user login/logout behavior (correct messages sent)
        iii. Online users list correctly updates on server given user login messages
        iv. Test many users communicating at once to test for concurrency issues (speed effects will be tested manually as described below)

GUI Tests:

These tests will require manual execution; buttons on the GUI have to be pressed to ascertain whether the GUI contains the **ClientController**, and programmatically triggered **ClientModel** events that update the GUI will have to be visually verified to ensure that the GUI is behaving correctly.

1. GUI correctly calls methods on controller
    a. Login
    b. Send
    c. IMWindow open/close
    d. Join Chat
    e. Select user in logged in users
    f. Select user in chat room participants
    g. Close window
2. ClientModel correctly updates GUI
    a. Login/logout
    b. Send
    c. IMWindow open/close
    d. Join Chat
    e. Leave chat
    f. Online user list updates