6.005 Project 3: GUI Chat Second Deliverable

Group 17: David Wen, Olivia Bishop, Wesley Graybill

May 8, 2009

Code Design

Server Design:

One of the key ideas behind our server design is the use of a centralized queue that stores all commands that need to be executed by the server. All classes that need to execute a command on the server are passed this queue and may add to it as needed. We then have one CommandHandler class that executes all commands added to this queue. Alternatively, we could allow each thread to execute its own commands. However, without the use of locks, this could lead to an inconsistent state on the server. The use of this single CommandHandler class ensures that only one thread will be attempting to change the server state at a time.

Additionally, in order to eliminate concurrency issues of adding to the queue, we will use a BlockingQueue from the java.util.concurrent package. This queue is thread-safe and will make it so that no two threads will be adding to the queue at a time. The combination of the CommandHandler and use of a BlockingQueue will effectively remove all concurrency problems in the server.

Below is a description of all primary classes of the server:

Server – The entrance point into the chat server. The Server creates and starts a thread running the CommandHandler and then listens on the socket for clients to connect. When clients connect, the Server sends a NewUser Command to the CommanHandler. The NewUser Command creates both an InputHandler and an OutputHandler for the client.

ServerState – The ServerState stores all states of the server. It manages all users and conversations, and has methods for adding and removing users and conversations.

Command – An interface that represents a command that the CommandHandler may execute. Each implementing class includes an execute method that handles executing the command. Command implementations include NewUser, NewConv, SetUsername, JoinConv, LeaveConv, Message, and RemoveUser.

CommandHandler – The CommandHandler pulls Commands off of the queue and calls the execute method of each command.

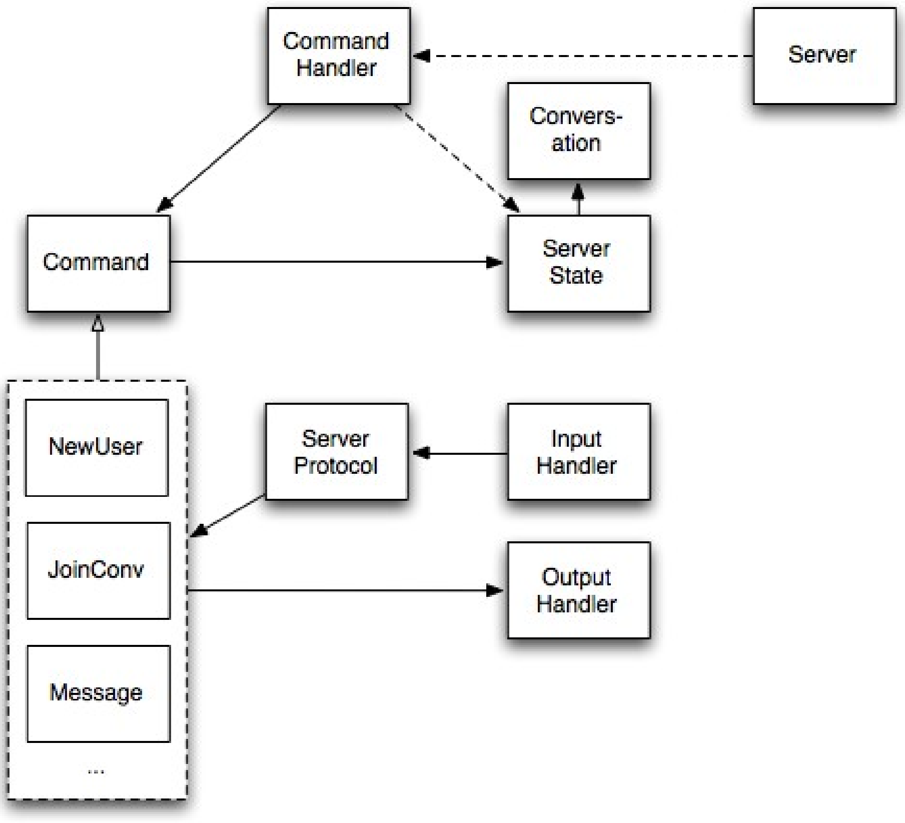
Conversation – A representation of a conversation. A Conversation contains a list of the users involved and methods for adding and removing users from the conversation.

ServerProtocol – The ServerProtocol is used by the InputHandler to add Commands to the queue. The protocol has a parse method that accepts a string as input and outputs the Command that is represented by the string.

InputHandler – The InputHandler is connected to the socket of a client. It listens to the InputStream of the socket and uses the parser of the ServerProtocol to add Commands to the central command queue.

OutputHandler – The OutputHandler is connected to the socket of a client and controls output to the client’s OutputStream. It has methods for each type of message in the protocol that construct the message and send it on the OutputStream.

Server Dependency Diagram:



Client Design:

Our client design is very similar to that of our server. The main differences are these:

- Instead of a ServerState, there is a ClientState, which carries information relevant to the client, such as the list of users logged in and the list of conversations available.

- The Commands are different, they are those that the client receives from the server.

- The InputHandler handles the stream from the server, and the OutputHandler writes to the server.

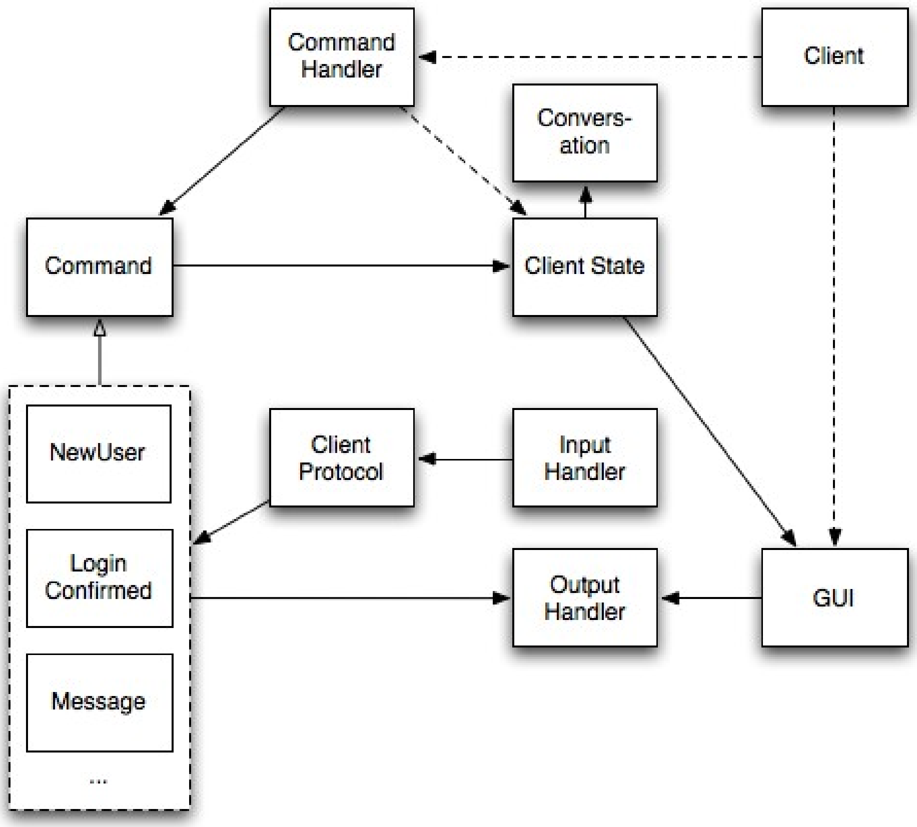
- There is a GUI, which the ClientState changes, and which passes data to the OutputHandler, which writes to the server. It will contain public methods for changing itself, such as those to add a conversation or user.

For the code that is reusable, we will recreate the classes in the appropriate packages for the client (copy and paste). This is because we want to be able to package the software separately, i.e. the client does not need the server's code to run and vise versa.

There were many different code designs we considered for integration of the GUI into the client model. Some consisted of intermediate classes for control, in order to make sure the data in the ClientState and that in the GUI matched at all times. We decided, though, that for what we are doing, this is not necessary. Others included a UI interface for the best possible abstraction. This will be difficult, as we need to take precautions to keep everything decoupled, but we think this is necessary for best results. It will make the code flexible (another GUI can be switched out if necessary), and testable (see testing section with the mock GUI). We also considered making Commands directly dependent upon the GUI as well as the Client State, but decided against this because we wanted to keep the GUI and ClientState updated and not allow otherwise. We ended with a design where the Commands act on the model (ClientState), and the ClientState acts upon the GUI by calling methods in it. This has the disadvantage that much of the data will be stored twice (in the model and the view), but is much less dependent and complicated than a system where the GUI listens to the ClientState for changes.

Note: on both the server and client diagrams, dashed lines represent weak dependencies, where the class only depends on the existence of the class it is linked to. For instance, the Server creates and begins running the CommandHandler, but does not depend on any of its functionality.

Client Dependency Diagram:



Testing Strategy

Our instant messaging has a server and a client component. We have designed both components so that they can be tested independently. The client and server communicate only through our text-based protocol. We just require that given an input, the server's state changes correctly and it outputs the right response. For the client, we require that given an input, the client state changes correctly and the GUI reflects this change. Also, we want to make sure that a client action on the GUI generates the proper output message.

We can automate testing on the server. We do this by initiating a server and feeding in specified inputs. Testing would then assert that the server's state changes appropriately and also assert that the server response is correct.

We can also automate testing on the client to a degree. By creating a client and feeding in specified inputs, we can check on its state to assert that it is changing correctly. However, we can't automate assertion that the GUI looks correct. We can, though, create a mock GUI that doesn't actually have any visual components, but gives indication whenever its methods are called. By having our client use this mock GUI in our testing phase, we can check on the state of the mock GUI to ensure that the correct GUI methods are at least being called.

We must create manual tests for the client as well. These manual tests will deal with ensuring that a user action on the GUI, such as adding a conversation or sending a message in a conversation, results in the correct output message to server. We will have to manually perform the action on the GUI. Also, we will need to create manual tests to ensure that the GUI is stable. For example, adjusting the size of windows should not break the GUI or make it unusable. Also, dialogs should work appropriately and never leave the user stuck in a state he or she can't get out of.

Demo

We have implemented much of the server side of the code, and we have a mock GUI that displays hard-coded information but demonstrates how our client program will look. See code.