Assignment 2 – Weight: 6.5%

Due Date: October 28th, 2020 at 9:00AM

This is an individual assignment.

We have seen Phase 1 of the simpleChat application in class. In this assignment, you will be adding features to this application.

The following series of exercises should ideally be followed in sequence. After completion of these exercises you will have built Phase 2 of the SimpleChat application. A complete implementation of Phase 1 is available at https://github.com/husseinalosman/simpleChat.

Setting up the SimpleChat Application

Setup: Login to your GitHub account and fork this repository: https://github.com/husseinalosman/simpleChat. Clone the forked repository to your own machine, so that you can make changes.

Compilation: Follow the instructions in the "Instructions for Installing the SimpleChat Application" document to compile the SimpleChat application.

Making Changes: You should make all changes as commits to your GitHub repository

Exercises

Exercise 1 – 30 Points

This exercise will help you to become familiar with the internals of OCSF and an instant messaging application we call SimpleChat (that is the application we have seen in class). Modify the application to provide the following features (Remember: do not modify the OCSF framework):

Client side:

a) Currently, if the server shuts down while a client is connected, the client does not respond, and continues to wait for messages. Modify the client so it responds to the shutdown of the server by printing a message saying the server has shut down, and quitting. Design hint: look at the methods called connectionClosed and connectionException.

b) *The client currently always uses a default port*. Modify the client so that it obtains the port number from the command line. Design hint: Look at the way it obtains the host name from the command line. Test that this works by connecting a client to a server using a different port from the default. If the port is omitted from the command line, then the default value should still be used.

Server side:

c) *Currently the server ignores situations where clients connect or disconnect*. Modify the server so that it prints out a nice message whenever a client connects or disconnects. Hint: you will simply have to write a code in EchoServer that overrides certain methods found in AbstractServer – study the AbstractServer description to determine which methods you must override.

Exercise 2 – 30 Points

Make further modifications to the SimpleChat application, as follows:

Client side:

a) Currently, the client simply sends to the server everything the end-user types. When the server receives these messages, it simply echoes them to all clients. Add a mechanism so that the user of the client can type commands that perform special functions. Each command should start with the '#' symbol – in fact, anything that starts with that symbol should be considered a command.

   You should implement commands specified as follows:
   i) #quit Causes the client to terminate gracefully. Make sure the connection to the server is terminated before exiting the program.
   ii) #logoff Causes the client to disconnect from the server, but not quit.
   iii) #sethost <host> Calls the setHost method in the client. Only allowed if the client is logged off; displays an error message otherwise.
   iv) #setport <port> Calls the setPort method in the client, with the same constraints as #sethost.
   v) #login Causes the client to connect to the server. Only allowed if the client is not already connected; displays an error message otherwise.
   vi) #gethost Displays the current host name.
   vii) #getport Displays the current port number.

Server side:

b) *Currently, the server does not allow any user input*. Study the way user input is obtained from the client, using the ClientConsole class, which implements the ChatIF interface. Create an analogous mechanism on the server side. Design hint: you will have to add a new class you can call ServerConsole that also implements the ChatIF interface. Following your modifications, the following should be true:
   i) Anything typed on the server's console by an end-user of the server should be echoed to the server's console and to all the clients.
   ii) Any message originating from the end-user of the server should be prefixed by the string "SERVER MSG>".

c) In a similar manner to the way you implemented commands on the client side, add a mechanism so that the user of the server can type commands that perform special functions. You should implement commands specified as follows:
   i) #quit Causes the server to quit gracefully.
   ii) #stop Causes the server to stop listening for new clients.
   iii) #close Causes the server not only to stop listening for new clients, but also to disconnect all existing clients.
   iv) #setport <port> Calls the setPort method in the server. Only allowed if the server is closed.
   v) #start Causes the server to start listening for new clients. Only valid if the server is stopped.
   vi) #getport Displays the current port number.

Exercise 3 – 30 Points

Make further modifications to the SimpleChat application, as follows:

In phase 1, clients are always anonymous. When a message is sent from a client, it is echoed to all the other clients, but nobody knows who sent it. In this exercise, you will implement a basic mechanism by which clients have a 'login id' that is known both to the client and the server.

Client side:

a) Add a new 'login id' java argument to the client program. This should be the first argument, before the host name and port, because the host name and port are optional in the sense that if they are omitted, defaults are used. The login id should be mandatory; the client should immediately quit if it is not provided.
   Design hint: the login id should be stored in an instance variable in ChatClient. You might ask the question: Why not put the instance variable in ClientConsole? The reason is to separate the user interface (how information is displayed and input) from the other aspects of the system.

b) Whenever a client connects to a server, it should automatically send the message '#login <loginid>' (i.e. the string #login with the login id appended to it) to the server. Note that this use of the '#' is different from what we have seen so far: The #login is sent to the server; it is not handled by the client as was the case with #quit, #logoff etc.

Server side:

c) Arrange for the server to receive the #login <loginid> command from the client. It should behave according to the following rules:
   i) The #login command should be recognized by the server. Design hint: Modify handleMessageFromClient so it does more than just echo messages.
   ii) The login id should be saved, so the server can always identify the client. Design hint: use the setInfo method to set the login id and the getInfo method to retrieve it again later.
   iii) Each message echoed by the server should be prefixed by the login id of the client that sent the message.
   iv) The #login command should only be allowed as the first command received after a client connects. If #login is received at any other time, the server should send an error message back to the client and terminate the connection. Hint: use the method called close found in ConnnectionToClient.

Exercise 4 - 10 Points

Run all the test cases described in the attached "Test Cases" document and make sure that all of them pass. This exercise will allow you to catch any problems in your implementation. Fill the ''Test Cases" document to specify whether a test case has passed or failed.

Submission of results:

If your Github repository is public, simply submit a file to Brightspace with a link to your repository. Otherwise, submit all classes that you modified. And submit that to Brightspace. **In addition to your code, make sure to include the "Test Cases" document in your repository or Brightspace submission.**