COMP-3415 Software Engineering
Java Project

Winter, 2020

Most software projects are evolutionary project which are developed by modifying an existing system. The purpose of this assignment is to practice such a development. This project also used a simple framework, which will give students some experiences for using a framework. This project will last a semester. However, a student should finish each phase on time frame according to the schedule of lectures.

Each student is required to do this assignment **individually** and to hand in the following on due date:

- Electronic files which provide complete solutions of Problems, showing all your work and including computer source files, input and output, if any.
- Optionally, you can also hand in some paper work related to your assignment.

Each set of assignment solutions should put the following information as a cover page:

Name, Student Number, Course Number (CS 3415), and your email address.

Assignments which are not met the above requirements will not be marked. The score of the assignment will depend on:

1. Specification and documentation: 15 %
2. Format and readability: 15 %
3. Correctness: 70 %

Submit your solutions by D2L.

Optionally, you can summit paper work at class.

OCSF (Object client-server framework) can be used to exchange Java objects between a server and clients. The textbook already developed a simple chat system using OCSF. Now you need to improve and extend this application according to the following steps.

Phase 1.

- Download source files of the frame "`OCSF`" and project "`SimpleChat`" from: `http://www.site.uottawa.ca/school/research/lloseng/supportMaterial/source/`
- Compile and test `EchoServer` and `ClientConsole` in your computer using eclipse or other software.

- Complete the modification of client side of the SimpleChat:

    1. In Simple Chat, 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 that it responds to the shutdown of server by printing a message saying the server has shut down, and quitting. (look at the methods called `connectionClosed` and `connectionException`).
    2. The client currently always uses a default port. Modify the client so that it obtains the port number from the command line. (look at the way it obtains the host name from the command line).
    3. 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 the following commands:
        (a) `#quit` cause the client to terminate gracefully. Make sure the connection to the server is terminated before exiting the program.
        (b) `#logoff` causes the client to disconnect from the server, but not quit.
        (c) `#sethost <host>` calls the `setHost` method in the client. Only allowed if the client is logged off; displays an error message otherwise.
        (d) `#setport <prot>` calls the `setPort` method in the client, with the same constraints as `#sethost`.
        (e) `#login` causes the client to connect to the server. Only allowed if the client is not already connected; display an error message otherwise.
        (f) `#gethost` displays the current host name.
        (g) `#getport` displays the current port number.

- Complete the modification of server side:

    1. 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. (write code in EchoServer that overrides certain methods found in AbstractServer).
    2. Currently, the server does not allow any user input. Study the way user input in obtained from the client, using the `ClientConsole` class, which implements the `ChatIF` interface. Create an analogous mechanism on the server side. (add a new class ServerConsole that also implements the `ChatIF` interface. Anything typed on the server's console should be echoed to the server's console and to all the clients. The message is prefixed by the string `SERVER msg>`).

- 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.

    1. `#quit` cause the server to terminate gracefully.
    2. `#stop` causes the server to stop listening for new clients.
    3. `#close` causes the server not only to stop listening for new clients, but also to disconnect all existing clients.
    4. `#setport <port>` calls the `setPort` method in the server. Only allowed if the server is closed.
    5. `#start` causes the server starts to listening for new clients. Only valid if the server is stopped.
    6. `#getport` displays the current port number.

The due date for Phase 1 is February 20, 2020.

Phase 2.

- Modify the client side:

    1. Add a new 'login id' command line argument to the client. 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. (login id should be stored in an instance variable in `ChatClient`).
    2. Whenever a client connects to a server, it should automatically send the message '`#login <loginid>`' to the server.

- Modify the server side: Arrange for the server to receive the `#login <loginid>` command from the client. It should behave according to the following rules:

    1. The `#login` commend should be recognized by the server.
    (modify `handleMessageFromClient`).
    2. The login id should be saved, so that the server can always identify the client. (use the `setInfo` method to set the login id and the `getInfo` method to retrieve it again later).
    3. Each message echoed by the server should be prefixed by the login id of the client that sent the message.
    4. The `#login` command should only be allowed as the first command received after a client connect. If `#login` is received at any other time, the server should send an error message back to the client.

5. If the `#login` command is not received as the first command, then the server should send an error message back to the client and terminate the client's connection. (use the method `close` in `ConnectionToClient`).

- Test your system and record the test results.

The due date for Phase 2 is March 12, 2020.

Phase 3.

- Modify the SimpleChat system so that it uses the Observable layer of the OCSF. Note that when you use Observable pattern in this application, you also need to use the Adaptable pattern.
- You need to modify both server and client referring Figure 6.15 page 248 of the text book.
- `ObservableClient, ObservableServer, AdaptableClient, AdaptableServr` are already included in the OCFS package. So you don't need to develop these classes.

Optional Phase

- You may finish this phase for extra credits (bonus).
- Design and implement a GUI for client of SimpleChat.
- Design and implement a GUI for server of SimpleChat.

The due date for Phase 3 is April 2, 2020.

Note

Plan your work for the project. You may need 1- 2 weeks for each phase. Don't wait till the due date, or you will not be able to finish the project. You may discuss with the TA in the lab if you have questions.