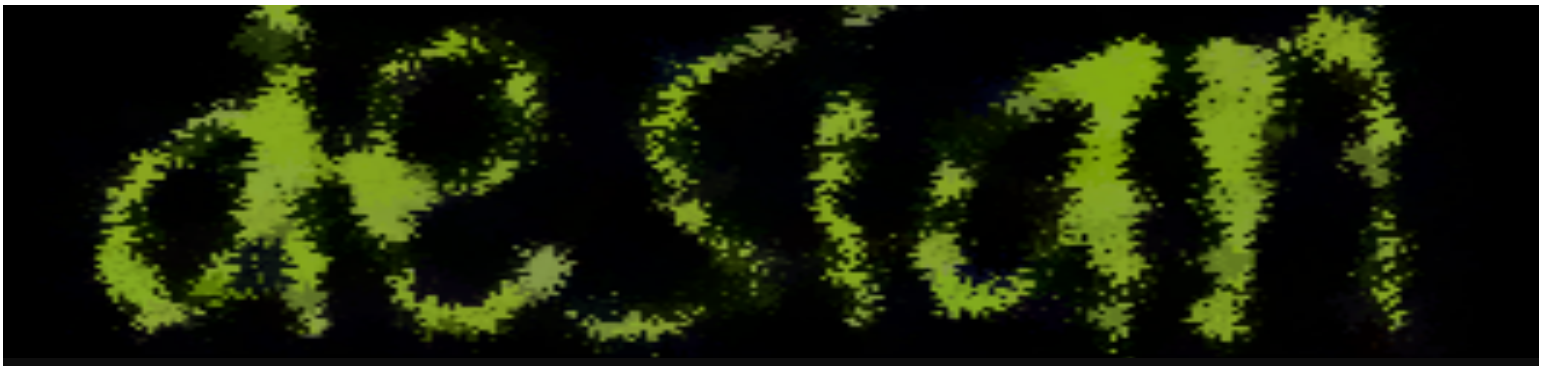
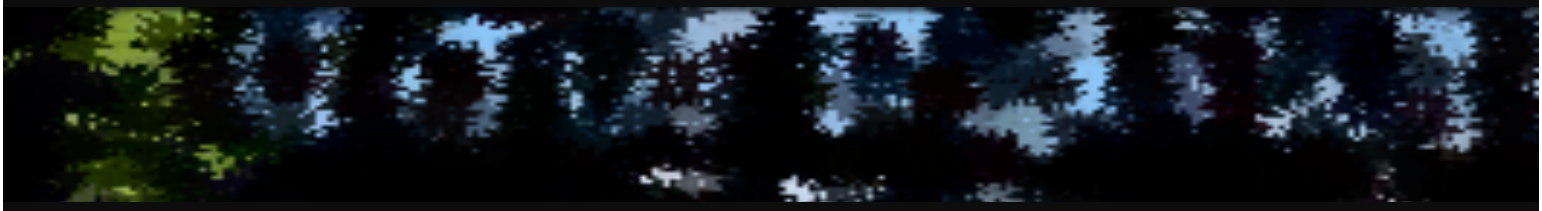
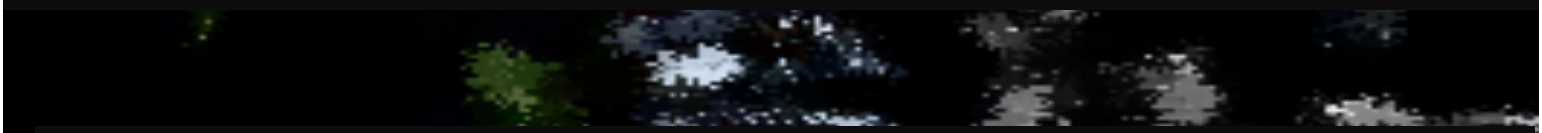


COMP 6231



Assignment #1



Hamidreza Heidarzadeh

ID: 40042707

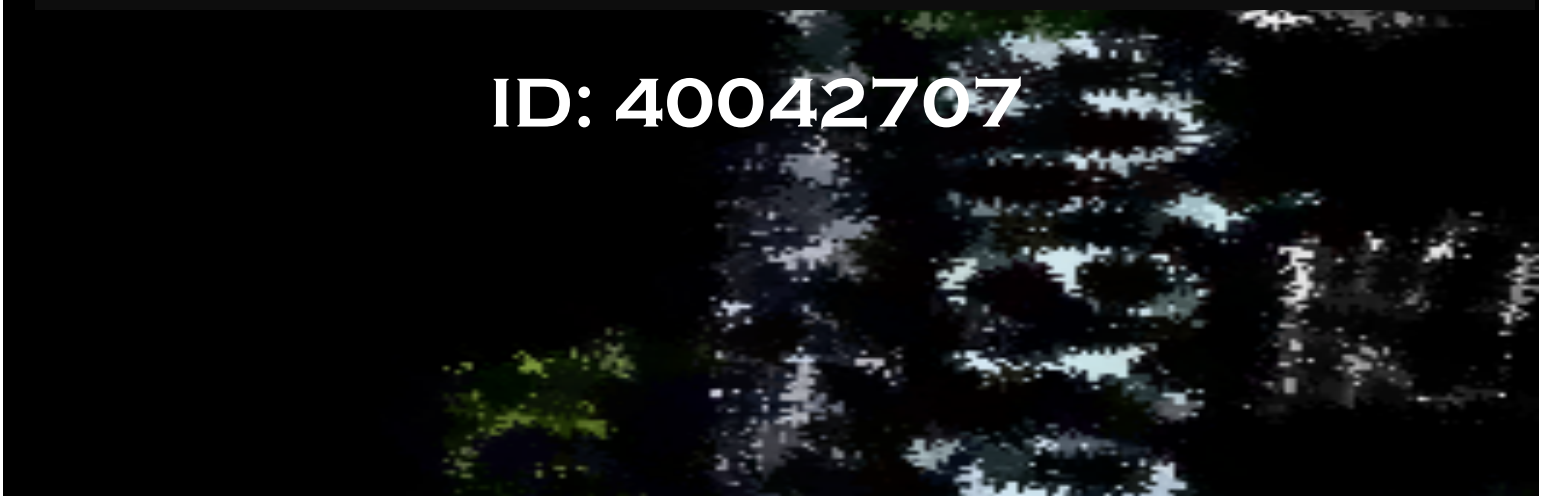


Table of Contents

Project Structure

How to run the application

Application Structure

THE MOST IMPORTANT/DIFFICULT PART IN THIS ASSIGNMENT

TEST SCENARIOS

1. Project Structure

Client Folder

a. **Logs Folder**: containing logs for any user who uses this file to perform some operations. The log file has following naming format "[Client Full ID].log"
b. **Client.java**: is the super class for "Student Client Class" and the "Advisor Client Class". Containing values both have in common.
c. **StudentClient.java**: is the class that handles Student related operations
d. **AdvisorClient.java**: is the class that handles Advisor related operations
e. **ClientTerminal.java**: is the main program that any user may use and run to be able to connect to servers and perform the allowed operations

Server Folder

a. **Courses Folder**: contains courses for each server (Departments). The naming format for files are "[Department][semester].log"
b. **Users Folder**: contains students enrolled courses for each student being saved in their own name. the naming format for these files are "[full id].log"
c. **Logs Folder**: Contains logs for each server. Naming format for these files are "[Server Name].log"
d. **serverRunner.java**: this java file will run all 3 servers (COMP, INSE, SOEN) in a same time through different threads.
e. **server.java**: this is the general server java file which handles all the operations related to each one. By passing the server name trough it's constructor, it will handle the operations related to its own department

Interfaces Folder

a. **clientInterface.java**: all the function related to a client regardless of being a Student or an Advisor is available in this file. Each client will then override any of these functions and will provide the required functionality.
b. **serverInterface.java**: all the server related functions, regardless of the server name is available through this file. Each server will then override these functions providing its own functionality.

Models Folder

a. **departmentInfo.java:** keeps the departments key parameter like "Server Name", "RMI Port", "UDP Port", etc. It then provides the values for each department (Server) by ant call on "get Department ()" function
b. **semester.java:** this class handles all the operation related to semesters. Like adding courses to a specific semester, removing course from a semester, loading course lists from the text file, saving any data to the course file, etc.
c. **courses.java:** this class holds the "Course Name" and "Course Capacity" information for each course.
d. **student.java:** this class handles all the student operations happening on the server Like enrolling into a course, dropping a course, reading student courses from the related file on the server in String format, saving any changes to that file, etc.
e. **logger.java:** handles all the log related functionalities. It holds the "log file address" as its own variable and performs all other operations on that specific file.

Document Folder

a. **Document.pdf:** is the document file for the project

2. How to run the application

Running the application:

1. **Run servers: (all 3 of them)**
    a. Javac "server/serverRunner.java"
    b. Java "server.serverRunner"
    NOTE: you cannot run the server file Separately. It needs another file to create an object from it by passing the right parameters.

2. **Run Client:** and login using any of the server's user format. (Student or Advisor) (COMP or INSE or SOEN) and continue on with any operation you like.
    a. Javac "client/ clientTest.java"
    b. Java "client. clientTest"

3. APPLICATION STRUCTURE

CLIENTS:

1. all the operations on the client side is handled by “ClientTerminal.java”. this class is “Runnable” and each call to run this file from command line, starts a new Thread. After Running, It starts with asking for the username. It then evaluates the username to see if this is a right format or not. And then, if it has the right format, then, continues with creating the right client object either from:

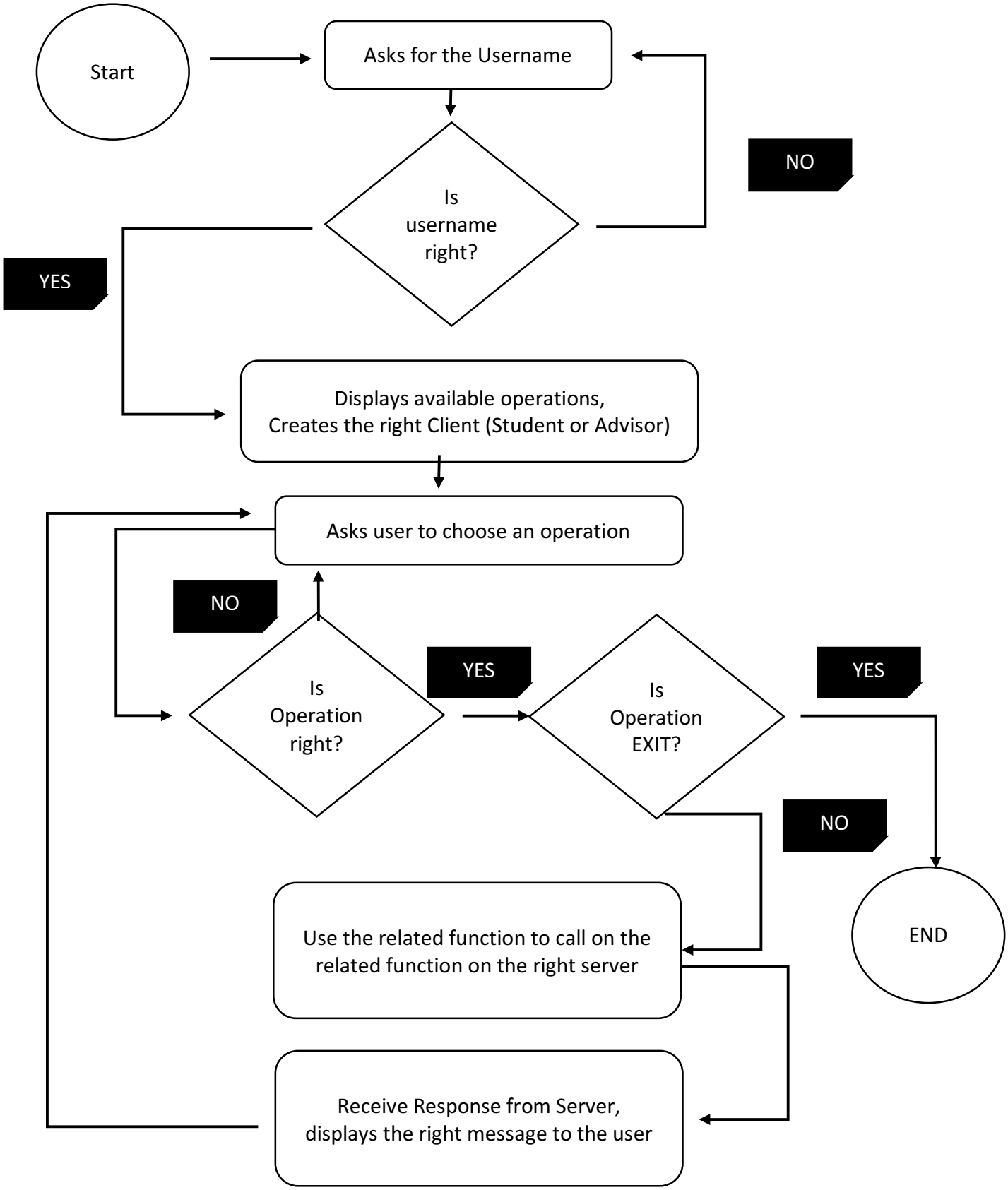
    a. StudentClient.java
    b. Or AdvisorClient.java

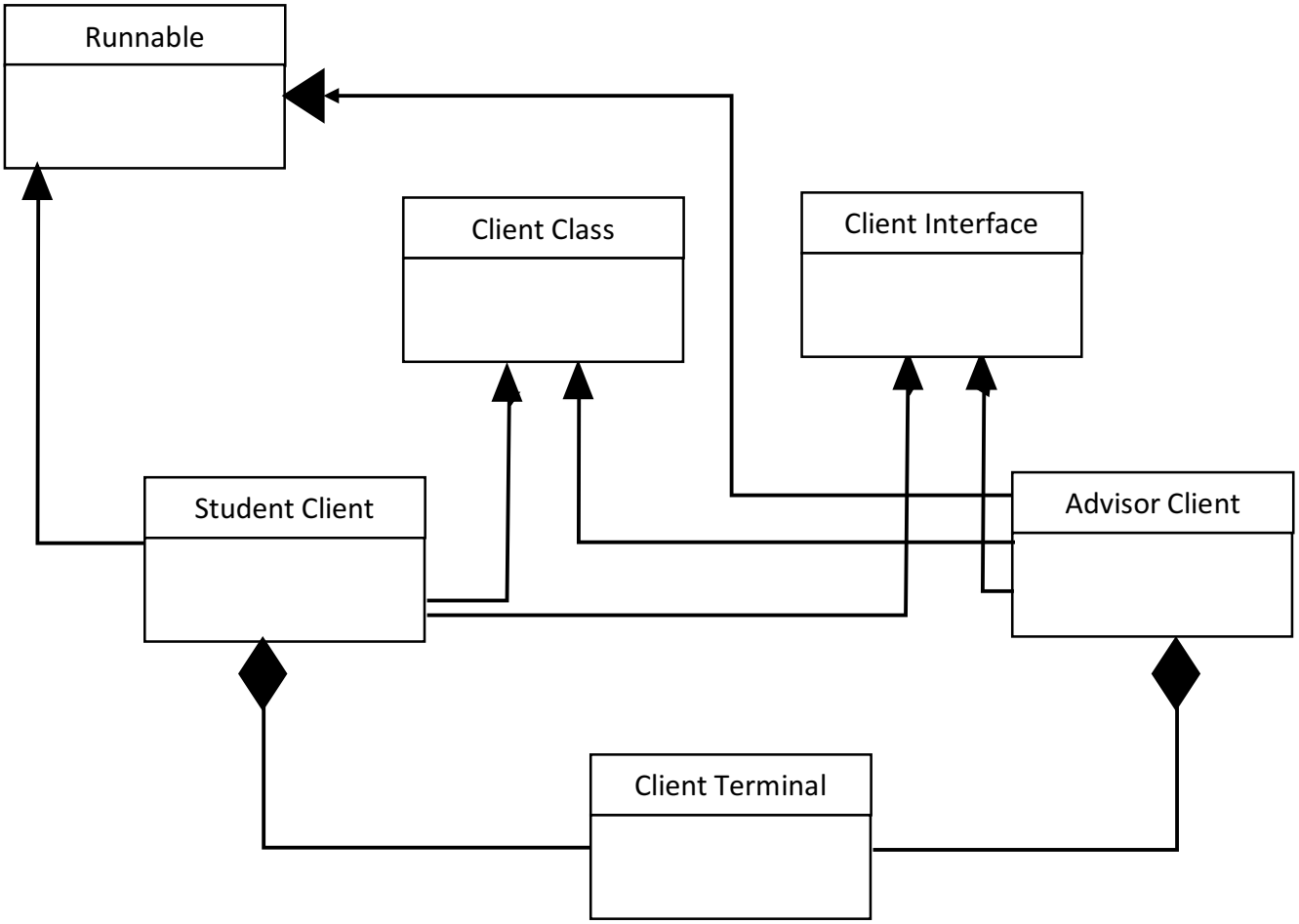
2. After creating the right client, it displays the right menu for the user. So, if that is a student, it will be allowed to ask for student operations. NOTE that, even if they know the commands for the advisors, they cannot run it. Because, the object [that has been created] just contains the related methods.

3. After receiving the command from the user, it passes that client object along with to the right function, which then will call on the correspondent function in server.

4. After receiving the response from the server, it displays the message to the user and wait for the next command from the user.

5. The user can terminate the program using “Exit” command on the console.


Start
Asks for the Username
Is username right?
NO
YES
Displays available operations, Creates the right Client (Student or Advisor)
Asks user to choose an operation
NO
Is Operation right?
YES
Is Operation EXIT?
YES
NO
END
Use the related function to call on the related function on the right server
Receive Response from Server, displays the right message to the user


Runnable
Client Class
Client Interface
Student Client
Advisor Client
Client Terminal

SERVER

Initiation

For server, there is just one class “server.java” which handles all the servers. When we create a new object from the class, by passing the server information through constructor method, It will servers the related methods. For example “Server(departmentInfo.COMP)” will just serve the “COMP” students and advisors, and it keeps the information related to “COMP” courses.

1. For the functionality described above, it receives all the required server information through constructor and saves it in the object. It also creates a “LOGGER” object from logger and provides a unique log file address.
2. Next, because the class is Runnable, through “run” method, it:
    a. Binds itself with the registry
    b. Runs UDP server
3. And then waits for any incoming request (RMI or UDP)

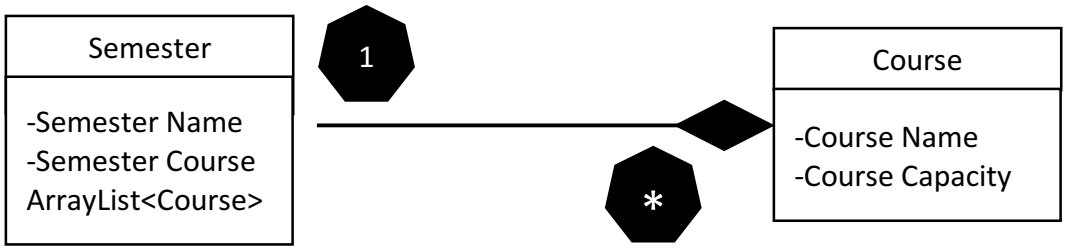
Advisor Operation

Advisor operation is all semester related. So, for that reason if any of those functions gets called, that function:

1. will create the related semester object from a related file
2. handles the requested operation (add, remove, listAvailableCourses) and then return the response to the server.

NOTE:

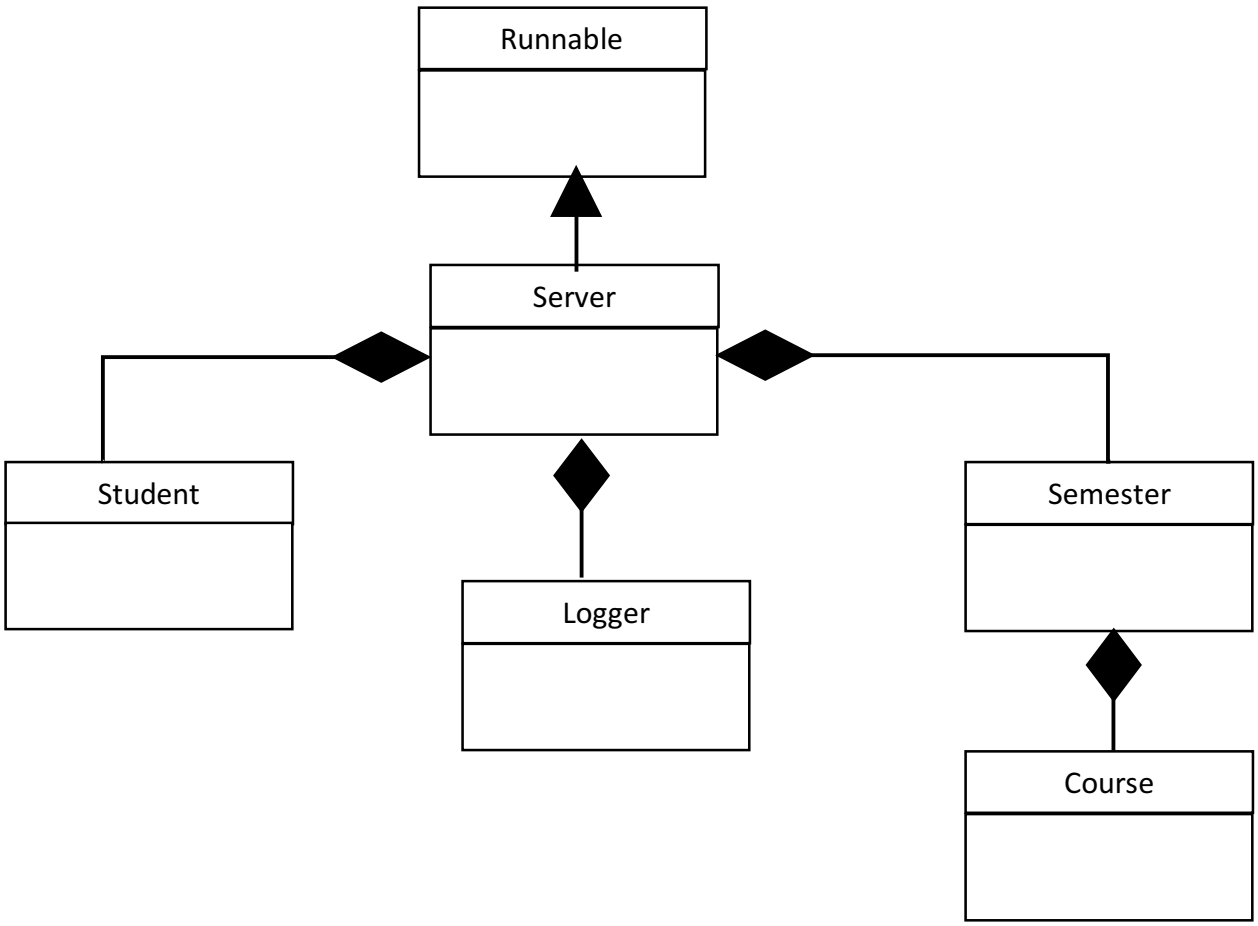
1. for “List Available course”:
    a. server checks its status and finds what server it is.
    b. And then sends 2 UDP requests to 2 other servers. It obtains the servers information using DepartmentInfo model.
    c. It then receives all the information from those servers.
    d. It creates a string, adding all the courser on after the other one and then sends it back to the client
2. Through each semester, there is an “ArrayList<Course>” which handles operation on the courses in that semester.
    a. It loads all the required information from, the correspondent file
    b. It performs all the requested operation
    c. It updates the file on the server



Student Operations

For student operation, the StudentID is always available. So, for each student related operation:

1. Server creates and object from the “Student” model.
2. Passes the StudentID to that object through Constructor.
3. And the object handles the rest by different methods in there.



4. THE MOST IMPORTANT/DIFFICULT PART IN THIS ASSIGNMENT

For me the most difficult part to run the "UDP" server on the server using a different port. Have it always running and listening on a port and also receiving and handling the requests and response to those requests through multi-threading.

1. So, I made the server Runnable [because I wanted to run all 3 servers using a separate java program.]
2. Then through "run" function for that Runnable server I run:
    a. RegisterWithRegistry(): which will act as a RMI server and
    b. RunUDPServer(): which will run the UDP server from the same server handling just the related servers UDP requests, on a different port.
3. I save and read the server parameters through the "Department Info" model in Models folder.
4. So, when a server wants to send a request to another one could obtain that information by just passing the server name to the "Get Department" function on that model.

5. Test Scenarios

I created a specific client test class "ClientTest.java" for testing following scenarios. Because through each one of these scenarios I should pass pre-defined variable values and the "ClientTerminal.java" needs to interact with the user through "console" which is not possible through automated testing.

Running the Tests:

1. You need to run the "serverRunner.java" from "Server" folder first.
    a. Javac "server/serverRunner.java"
    b. Java "server.serverRunner"
2. Then you need to run "ClientTest.java" from "Client" folder.
    a. Javac "client/ tests.java"
    b. Java "client. tests"

3. The result will be printed on the console.
4. If needed, you can refer to the log files on each server for further details.

For each testing scenarios, I will mention the function which will be called and the values I pass to those functions, followed by a screen shot from the final result which will be printed on the console.

1. @Test => checking with wrong username Format;
2. @Test => checking with the Advisor username format;
3. @Test=>Checking with the Student username format;
4. @Test => advisor => listCourseAvailability (fall)
5. @Test => advisor => addCourse(String courseID, String semester)
    a. advisor tries to add course for fall which already Exists
    b. advisor tries to add a new course for fall
6. @Test => advisor => listCourseAvailability (fall)
7. @Test => advisor => removeCourse (String courseID, String semester)
    a. advisor tries to remove course in fall which doesn't exist
    b. advisor tries to remove course in fall which exist
8. @Test => advisor => listCourseAvailability (fall)

9. @Test => student => getClassSchedule (String studentID)
10. @Test =>student => enrolCourse (String studentID, String courseID, String semester)
    a. student tries to enroll a course that doesn't exist
    b. student tries to enroll in a COMP course which has been added by the advisor
    c. student tries to enroll in a SOEN course which has been added by the advisor
11. @Test => student => getClassSchedule (String studentID)
12. @Test => student => drop Course (String studentID, String courseID)
    a. student tries to drop a course which doesn't exist
    b. student tries to drop a COMP course which has been enrolled into before.
    c. student tries to drop a SOEN course which has been enrolled into before.
13. @Test => student => getClassSchedule (String studentID)