**SEG2105A Assignment 5**

1. *The Problem*

University of Ottawa students have no easy way to generate a conflict-free, optimised schedule.

2. *System Description*

The system will be used by students looking to generate an optimized timetable. A student will specify which semester they are planning for, and what courses they want to take. Courses are made up of sections. Each section will contain at least one activity. Activities can be lectures, seminars, discussion groups, tutorials, and laboratories.  More activities are optional. The section will specify the number of lectures, labs, and tutorials required. If a minimum has not been specified of any one kind of activity, then all activities of that type in the section are required for a course selection. The student should be able to specify when he or she does not want to be in a certain section, due to, for example, an unwanted professor or that section being full. Once all course and section selections have been made, system will generate all possible conflict-free schedules.

Students will be able to sort the schedules with options prioritizing number of days off, longest breaks, shortest days, latest start, or earliest finish. The student can also choose a set of courses, and from this set choose a minimum number that will be placed on the schedule. For example, if a student must choose two electives out of four possible elective courses, the generated schedules will contain **every** combination of two out of the four electives, and sorting options will help students choose the best. The student should be able to see a graphical representation of the schedule, including a color-coded week-long calendar and a table that summarizes the course-code information for quick reference when registering. The student should also be able to save the information to a file, along with the option of opening another configuration or starting a new one.

3. *Some user stories:*

**Student:** I want to make a schedule that is conflict-free and doesn’t have many evening classes so that I can sleep more.

**Student:** My friends are taking the same courses as me. I will save my configuration and send them the file to save them some time.

**Student:** I don’t like a certain Professor. I’m going to exclude all the sections with that professor because I want a teacher that has a better reputation.

**Student:** I want to drop a course, so I’m going to see if dropping this course allows me to have a day off. The easiest way to do this is to edit my configuration file and remove this course, then prioritize “day off” in sorting.

**Student:** I have one elective to take, and three choices. I want the one that lets me wake up the latest. Using the scheduler, I’m going to put these courses in a set next to with my mandatory courses and see which one fits my schedule best, using sorting options.

4.      *Architecture*

The system uses a client-server architecture. The coding language will be Java, using OCSF. The server will be controlled via console, including only simple actions such as restart, stop, quit, reload courses, etc. It will also display where connections are coming from at any given time.

Upon startup, the server will load in a comma-separated file that holds information for all course information for the current semesters. The information will be held in a list, possibly an array list, of “Course” objects. Each Course will have a description and a list of Sections. Sections have a name(course code + a letter), a semester, and a collection of Activities. Each Activity has a type (lecture, seminar, discussion group, tutorial, laboratory), a number (e.g.: lecture 2), a day of the week, a start time, an end time, a location, and the name of the professor.

After starting up and loading the database of courses into memory, the server starts listening for clients. The client will automatically connect to the server at startup, having an option to change the default server address using a settings pane. The entire interface for the client is graphical. After connecting to the server, the client may start creating their schedule. To do this, the user uses the graphical interface to select a semester. After that, they use a search interface that responds to course codes and to course names to find their course. The search works by sending what the user is typing to the server, the server returning a list of courses that partially match the user’s input, and then the client displays the list in a list box. After selecting a course in the list box, the client hits an Add button, and the Course object is sent to the client. The course then displays on a list of selected courses, on the client side. From there, the user may remove selected courses, add more courses,change the desired sections for selected courses, as well as use other special course selecting functions. One such planned function is the option to select x of y courses. Each course option will be held on one line, with the exception of an x of y selection. For an x of y selection, y courses will be on a line, and the user will specify that they need x of them. *In real-world usage: I have 2 available electives and 7 choices. What’s the best choice for my schedule?*

After making a selection of courses, the user can now generate schedules. The user initiates this process by selecting a sort order (“maximum number of days off”) and then clicking a *generate courses* button. Then the course objects, along with their section selections, are sent to the server. The server will generate a list of every single possible combination of *CourseSelection* for all given courses. A CourseSelection is an object that contain a valid set of activities for a course. A course will have a method to return an ArrayList of all possible CourseSelections for a desired semester. The CourseSelections will be made up of every section, and every combination of optional activities possible. The server will generate all the CourseSelections for every Course that the client sent. Afterwards, it will systematically compare every single possible combination of CourseSelections to see what combinations do not have conflicts. Every set of CourseSelections that has no conflicts and includes every course that the client specified will be called a *Schedule*. The schedule will have statistics associated with it, like average length of days, average of starting times, average of ending time, and number of days off. After all possible schedules have been generated, the next step is to sort them the way that the client specified. After sorting based on statistics, the ordered set of schedulesis to be stored with the ConnectionToClient on the server, so long as his course selection stays the same. This way, if the client chooses another sort order but doesn’t change his course selection, then the server will simply reorder the set of schedules, instead of generating all of them again.

The next step is for the server to send back the ordered set of schedules generated, where the client will display the generated schedule, the schedule order, and give the option to transition through to the end of the list. The client may save their instance, which includes the course and section selection, the list of sorted schedules, and the sort order. All these objects are serializable in order to be saved to a file. The user may also load another instance, which will send the necessary files to the server after loading. The program’s settings (at this point, only the server address) are held locally in a configuration file, separately.

This should cover the essential architecture. All details are subject to change, depending on design decisions made later on. As is apparent, this is a thin-client, thick-server operation. The server stores the global schedule database, does the searching, the brute-force computations, and all the sorting. The client only sends simple lines of input and receives and displays outputs.

5.       *UML*

Editable Umple model:

<http://cruise.eecs.uottawa.ca/umpleonline/umple.php?model=131103600516>

Umple Textual Code:

class Course

{

description;

1 -- 1..\* Section;

}

class Section

{

name;

semester;

Integer requiredLEC = 0;

Integer requiredSEM = 0;

Integer requiredDGD = 0;

Integer requiredTUT = 0;

Integer requiredLAB = 0;

1 -- 1..\* Activity;

}

class Activity

{

type;

Integer number;

day;

Time startTime;

Time endTime;

place;

professor;

}

classCourseSelection

{

\* -> 1..\* Activity;

}

class Schedule

{

\* -> 1..\*CourseSelection;

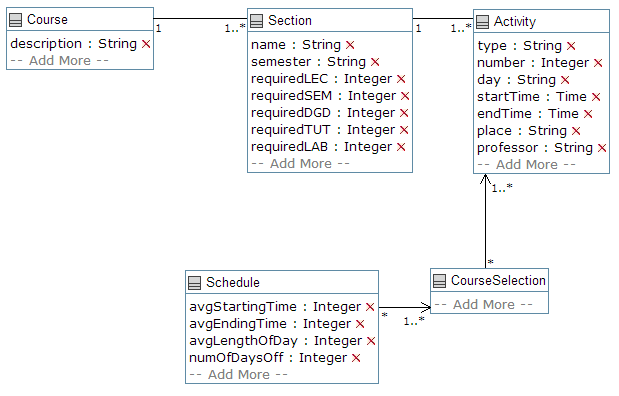
Integer avgStartingTime;

Integer avgEndingTime;

Integer avgLengthOfDay;

Integer numOfDaysOff;

}



6.       *Messages*

**From Client to Server:**  
-"Request for Timetable from Client xxxxxxxx for Semester "2014 Fall". Classes: SEG2105 CEG2136 CSI2110"  
 This message is an example of what it would look like on the server console side when a client (xxxxxxxx) submits a request to generate a timetable for the given semester with the given three classes (SEG2105 CEG2136 CSI2110). This message would look different and be added upon if the client desired to specify what type of days they would like (long break days, early classes, late classes, etc.), or if they have to take a subset of a given set of electives.   
**From Server to Client:**  
-This area of the program will be mainly visual, displaying timetables that have been generated by the server to the client. The server will be returning requested timetable generations, however these can't really be considered as messages, they are more like data, as they will be parsed by the clients application . Ex:

- "Class: SEG2105 Group: A Lecture 1: Tues 1200-1330 Lecture2: Thurs 1200-1330   
 Lab: Tues 1900-2200 DGD: Wed 0800-0930"

-Messages like this can be parsed by the clients application and turned into visual representations of the server generated timetable.