**3. Architecture Design**



As was discussed in Deliverable 1, The Force’s high-level architecture follows Model-View-Controller (MVC) architecture. Since the MVC architecture allows for a clear separation between domain logic, controllers, and user interfaces into separate entities/components, it is ideal for this project. Any requests for communication will be made through the appropriate controller, and the components designated to be models would not be exclusive to a single controller. Since the same data might be needed over numerous web pages (for example, when displaying a schedule on one page and when displaying student information on another), the controllers would be able to access said data directly. In addition, the separation of concern allows for development without the need for concern over minor interactions leading to unsatisfactory interference between components.

The complete design of The Force will be presented in various forms and contexts using the 4+1 view. This architectural view will break The Force into Logical, Process, Development, and Physical View, as well as include Scenarios for user interaction.

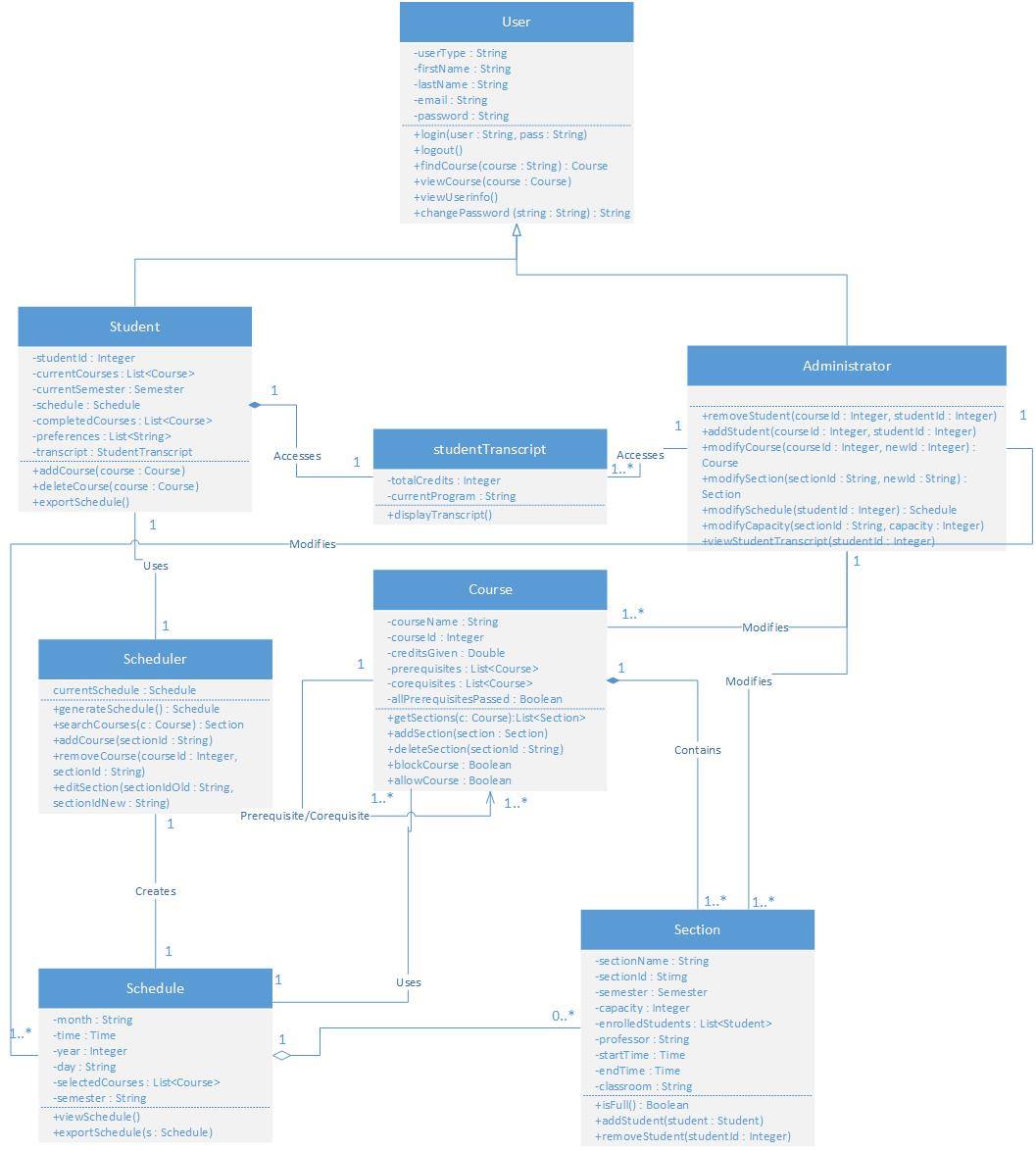
**3.1. Architecture Diagram**

**3.1.1. Logical View**

Logical views are concerned with presenting the functionality provided by the system to the users. As such, the logical view presented here is a presentation of the subsystems (and their components) of The Force’s model component (Refer to 3.1.2 for a full component diagram).

The actors in the system are denoted as Student and Administrator, both inheriting general characteristics of a standard “User”, but with their own unique functions as well. Students are given the ability to use the Scheduler to generate a possible combination of Courses he/she has yet to take, as well as view said Schedule once it’s been made. Each Student also possesses a studentTranscript (which cannot exist without said student), which is used to track his or her academic progress. The Scheduler creates Schedules through the use of Courses and Sections, these being available options that the Student still has remaining in their academic careers. The Student can make modifications to their Schedule, assuming the necessary requirements are met (e.g.: Prerequisites for a particular class), and can also export said Schedule for ease of access. The Administrator, though lacking in the ability to use the Scheduler, can modify the contents of all Courses, Sections, and Schedules, as well as view any studentTranscript he or she might want to see.

Through the use of the following class diagram, it can be seen that concerns are well separated through the use of multiple different classes. Unlike in a system where design dictates that a single class holds a large majority of the functionality, The Force ensures to the best of its abilities that concerns are separated appropriately. As such, changes can more easily be made without affecting a majority of the system, thus allowing for faster progress and implementation.

****

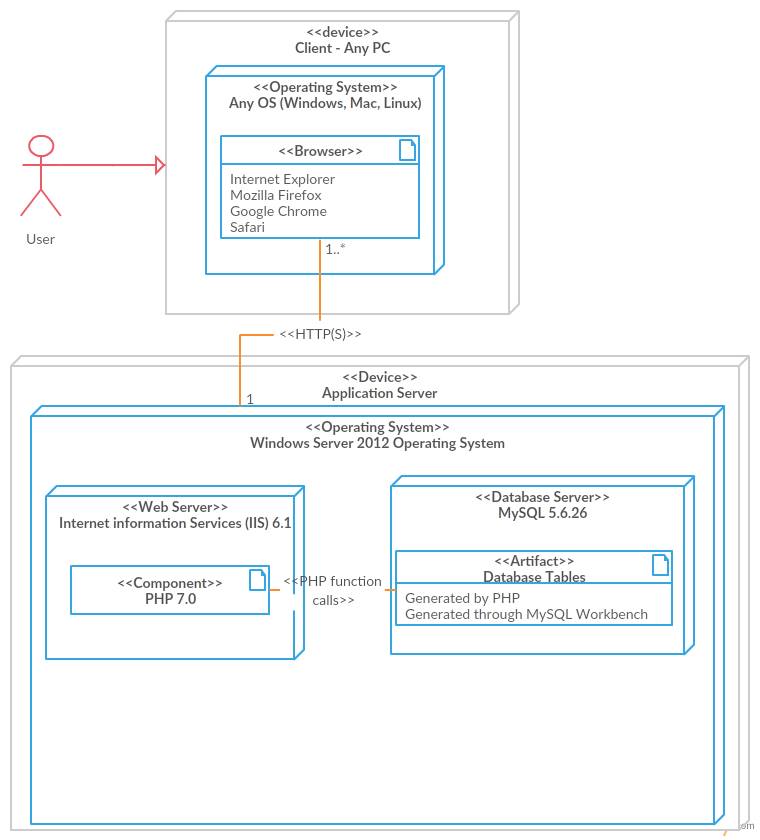
**3.1.2. Development View**

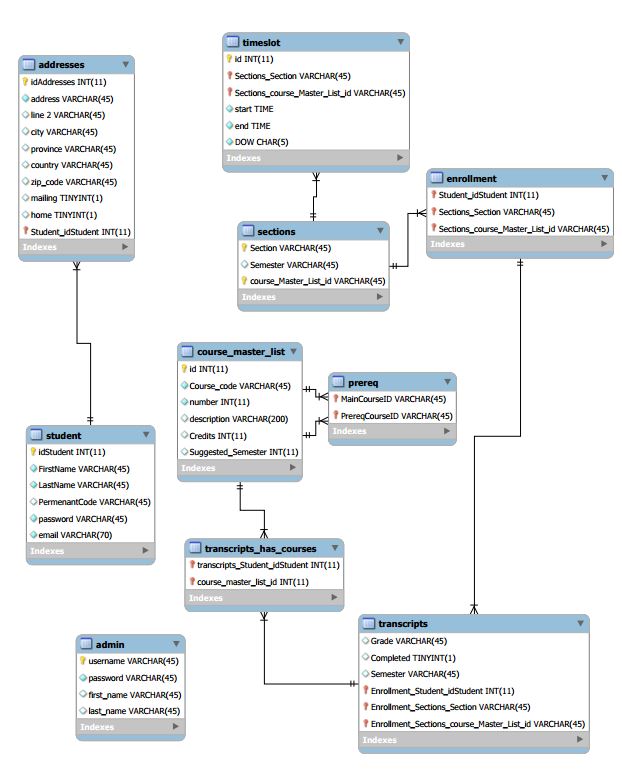
Development views, or implementation views, are concerned with software management. They aim to present a system from a programmer’s perspective.

The Force, as was previously mentioned, is designed using the MVC architecture and as such will be broken down into three components: Model, View, and Controller. All logic, for both the model and controller), is being implemented using PHP and is being handled by a single server. On said server, Windows Server 2012 Operating System is being used as the Operating System, and MySQL 5.6.26 is used for all database needs. The view is being rendered through the user’s browser using HTML, CSS and JavaScript, which the application is generating.

The client side of the application relies strictly on multi-platform browsers such as Google Chrome. As such, The Force will work on all clients but support is mainly given to current versions of most popular browsers currently being used. Through the use of HTTP requests and responses, communication is made possible between the user’s browser and the server.

In addition, a database model is provided in order to present the interactions between tables currently being stored on the server. This database is manipulated through the use of PHP (and MySQLi queries), and MySQL workbench.

****

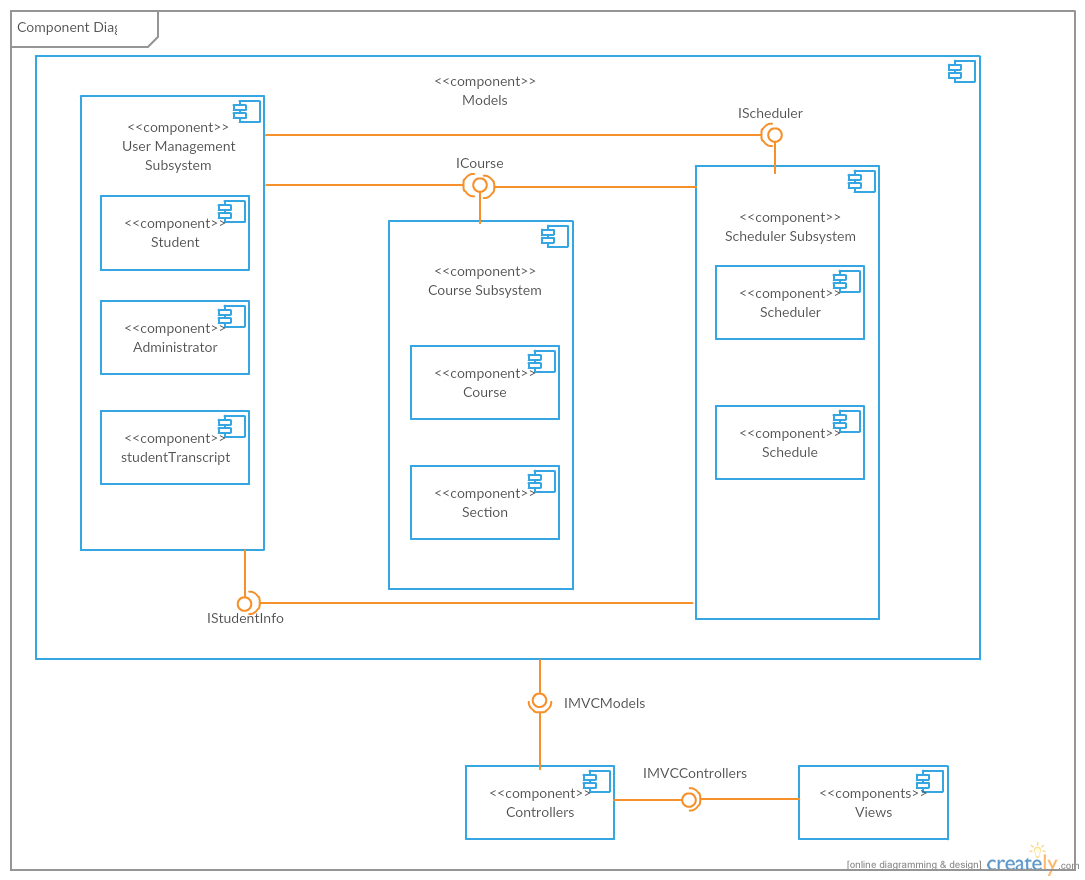
****

**3.1.3. Physical View**

Physical views are concerned with the software components of a given system, as well as how they communicate between each other. As The Force follows MVC architecture, it is broken down into three major components: Model, View, and Controller.

The Views component is composed of all the user interfaces a user would need to interact with while using The Force. It is used to connect information found within the Models to HTML templates, thus allowing a user to understand and interact with them. Each Views component is paired with a controller from the Controllers component, as it is to manipulate the Models based on user input and changes the Views accordingly.

The largest component is the Models component, which is made up of 3 distinct components representing the major subsystems of the application. The User Management subsystem is comprised of classes used to represent the various types of users who might be using The Force. In addition, they deal with all attributes and services unique to just the User classes, such as viewing student transcripts. Classes within the User Management subsystem not only provide information to other subsystems (namely the Scheduler subsystem, allowing its to compose schedules based on the appropriate student information), but it also requires methods from both subsystems in order to manipulate the schedule by modifying the courses being taken by said Student (or by a Student, in the Administrators case). The Scheduler subsystem, in addition to representing all classes relating to one’s schedule and schedule-making, also requires information from the Course subsystem in order to create proper representations of a Student’s academic semester. The schedules it houses are the used by the User Management subsystem, namely when exporting or viewing schedules. Finally, the Course subsystem encompasses all courses and sections provided by the university. It provides services to both other subsystems in order to generate, display, and modify appropriate configurations of courses.

****

**3.1.4. Process View**

Process views deal with how a system’s processes communicate, as well as focusing on the runtime behavior of the system. One activity diagram (Figure 3.1.4.1) is used in order to denote the main purpose of The Force (the generation of a schedule), and two more activity diagrams are used to go into greater detail on how schedules would be auto generated (Figure 3.1.4.2) or manually generated (Figure 3.1.4.3) during the runtime of the application.

Figure 3.1.4.1 describes the general runtime behavior of the complete schedule generation process. Once a student logins in (as an administrator doesn’t possess this feature), they are greeted with a main page consisting of their current semester’s schedule and courses. From here they may either exit the program, access their account page in order to change their personal information (and be redirected to the main page), or remain on the same page. From the main page, the student’s semester could be changed (allowing them to view past or future semesters) thus updating the displayed schedule and course list, or a schedule could be generated. Generation could be done in one of two ways, auto generation (Figure 3.1.4.2) or manual generation (Figure 3.1.4.2), both of which will be explained in more detailed shortly. Once a schedule’s been made, the student can either quit the application, or modify their schedule manually (perhaps they are unhappy with an automatically generated course they were given, or they failed to notice an error while manually creating their schedule). Attempting to modify the current schedule will result in a display of the student’s current courses with modification options now present. Removing a course will simply update the schedule, however requesting to add or change a courses section will result in a list of alternatives (found similarly to what is seen in Figure 3.1.4.3), and once a new option is chosen the complete schedule will be displayed. Once modifications are complete, the student can make different modifications, or simply quit the application.

Figure 3.1.4.2 describes the auto generation process of creating a schedule. Once access to a student’s transcript and preferences are achieved, their current total credits within their program are analyzed. Should it be that they already have the maximum possible amount of credits they can achieve, they would have no courses left to complete and thus would be done, generating an empty schedule. Should their credits be lower than this arbitrary total amount, access is given to the student’s current program and course sequence, and the courseId for the next course in their sequence is acquired. Should the number of courses ever reach the max they are allowed per semester (e.g.: Perhaps 5 or 6), the schedule will stop adding courses and update itself. Otherwise, the next course will be selected and examined. If the student is still taking less than the recommended amount of classes, and the classroom still has rom in it, the course will be added to a list. So long as another course exists in the sequence, this cycle will continue until the maximum amount of courses is being taken, or there are no courses left (credits would equal the maximum amount), thus updating the schedule and ending the generation.

Figure 3.1.4.3 describes the process by which a schedule is generated manually by a student. Upon selecting the option, the student is told to enter their unavailability, or preferences (e.g.: No school on Friday, no class after 4:00pm), and they are then able to enter the name of the course they wish to add. If the course doesn’t exist, an error message appears and they can try again, otherwise the course information is displayed. Before searching for possible sections based on the students unavailability, room availability (whether there’s space in the class), and whether the necessary prerequisites are met, the student could decide against the current choice and change their mind. Once all checks have been performed, the student could cancel their selections (exiting the generation), or should a proper timeslot have been found, they may select their preferred section within the course. This will lead to their schedule being updated, and they may either stop choosing courses, or repeat the process again. It should be noted that the process used to check for a course availability is the same process used to determine the list of alternative sections proposed in Figure 3.1.4.1, thus it was omitted there to reduce redundancy.

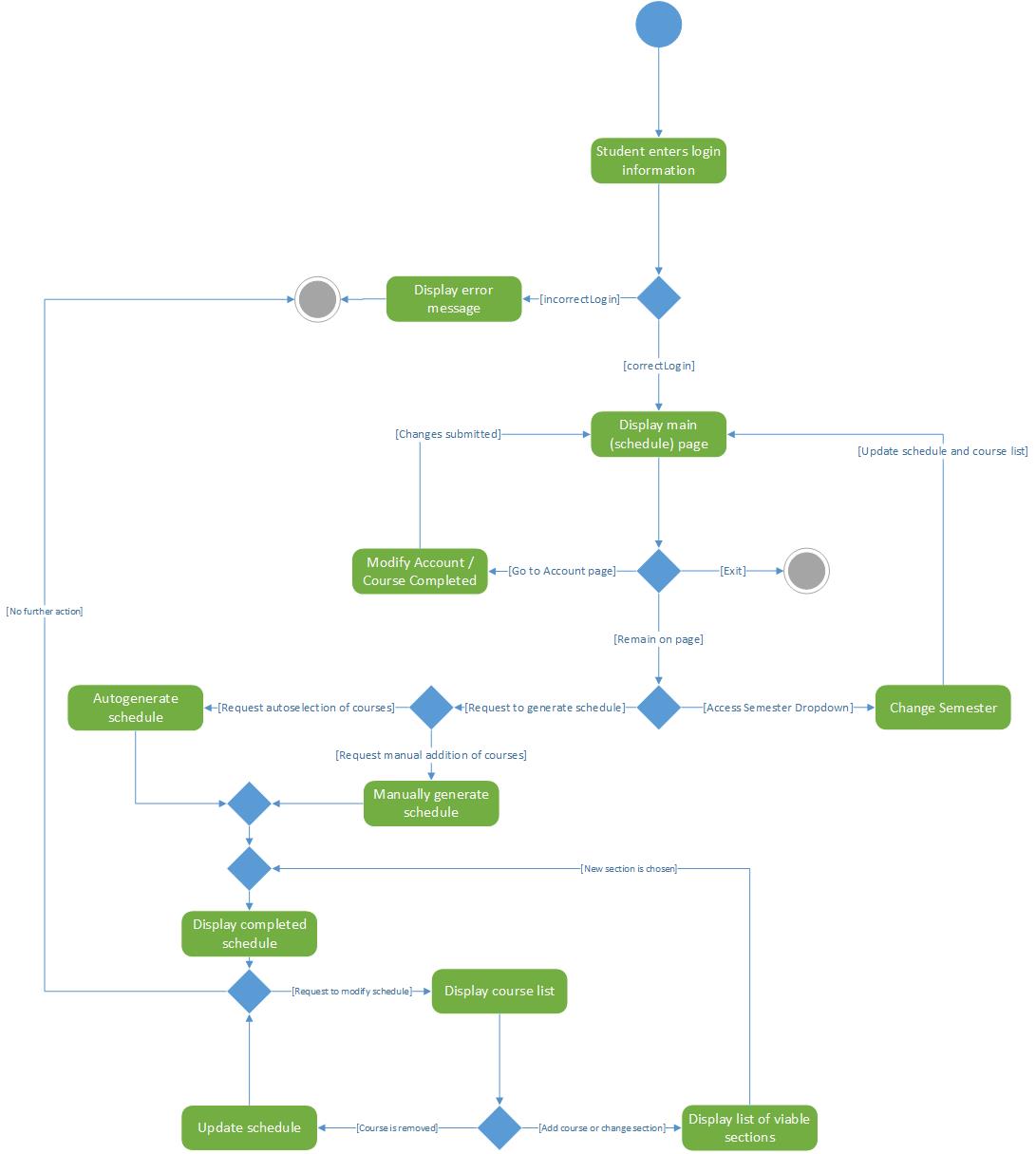
****

Figure 3.1.4.1: Main Schedule Generation

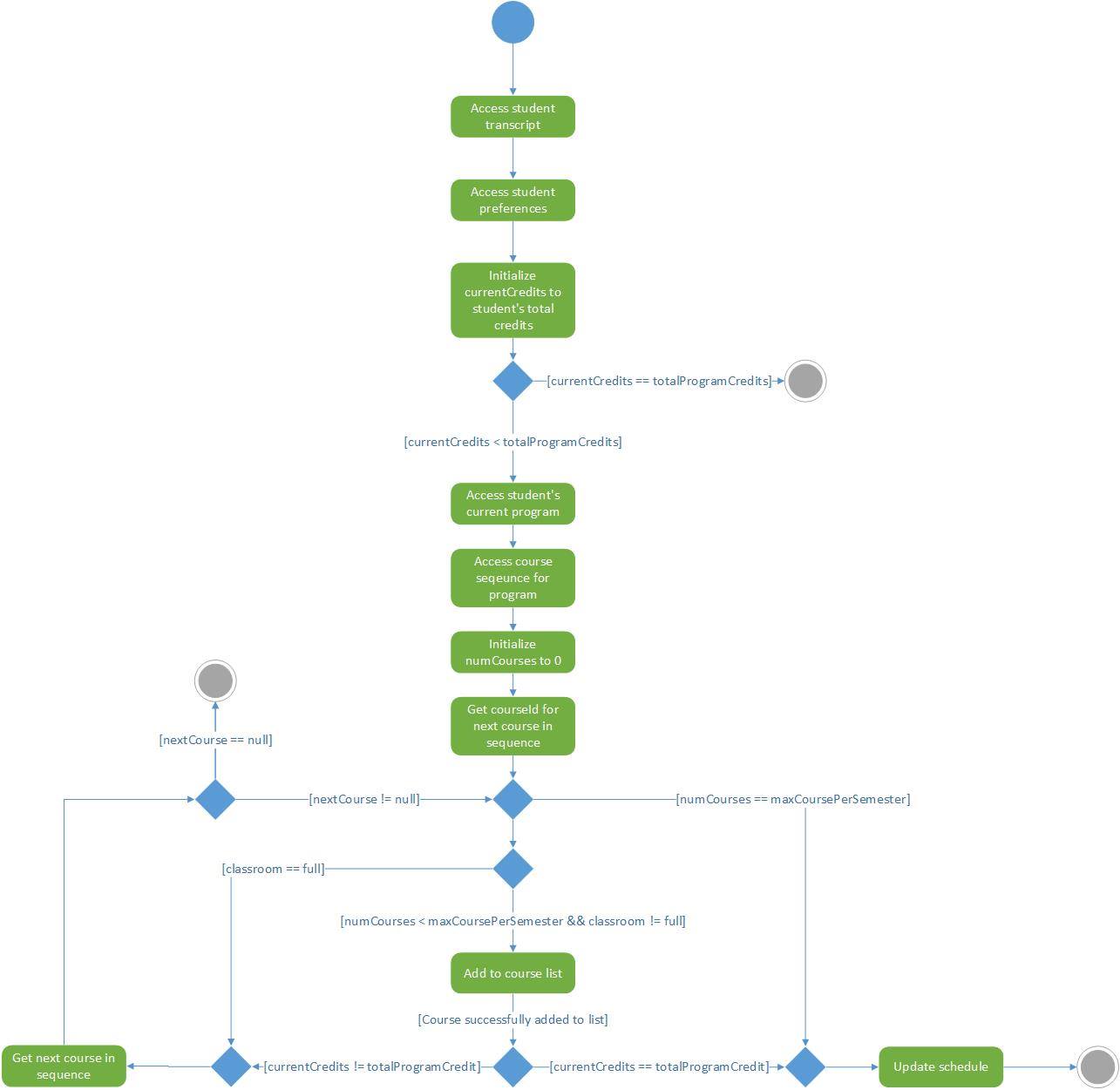


Figure 3.1.4.2: Auto Generated Schedule

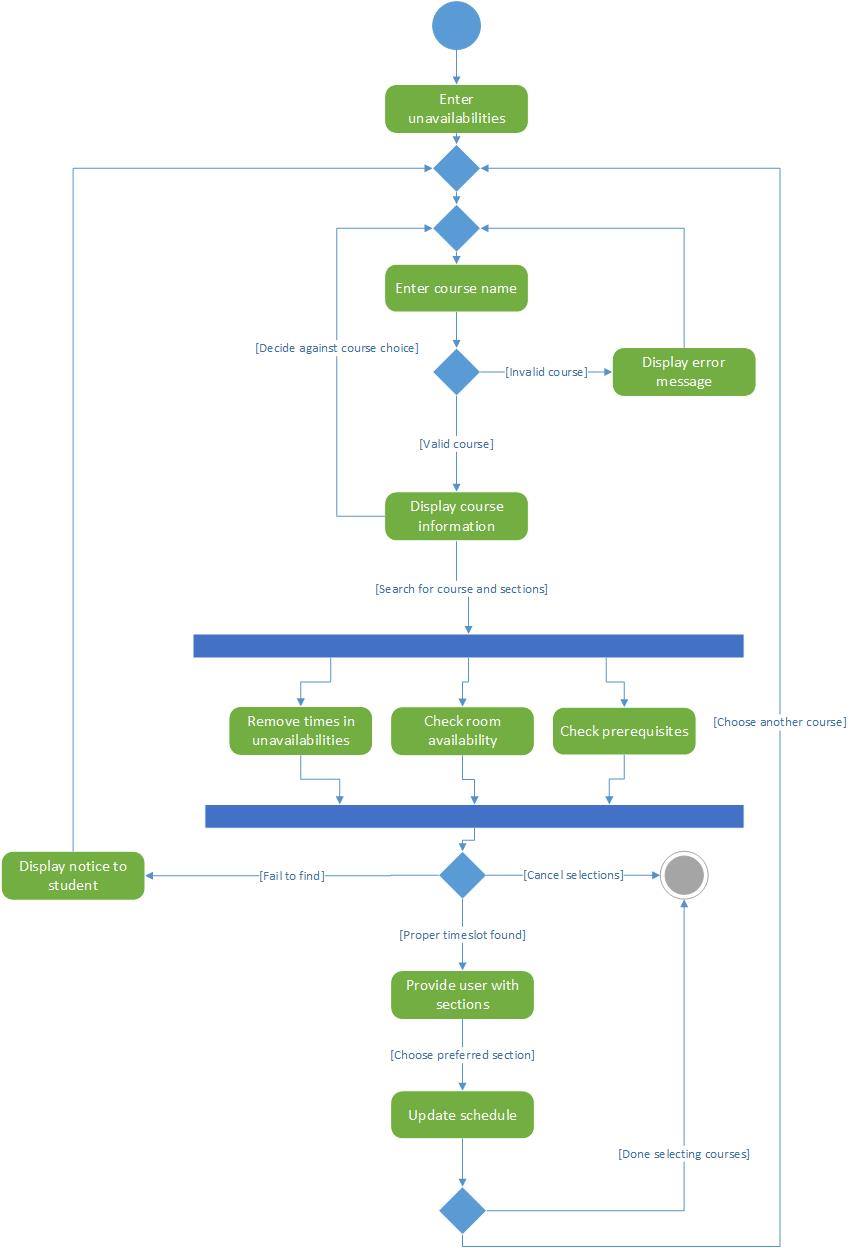


Figure .1.4.3: User Generated Schedule

**3.1.5. Scenarios**

Scenarios are used to interactions between objects and between processes, as such Use Case Diagrams will be used to denote the way the various types of users can interact with The Force.

It should be noted that, by comparison to the Use Case Diagram made in Deliverable 1 (Figure 3.1.5.1), quite a bit has changed. First and foremost, the UCD from Deliverable 1 was strictly for one situation, most notably assuming that the only type of user would be a student. Since then, the addition of the administrator user type not only required a separate diagram (Figure 3.1.5.4), but also allowed for the creation of a general User diagram (Figure 3.1.5.2). This use case diagram denotes interactions which both Students and Administrators have access too, and helped not only reduce redundancy within each unique diagram, but also allowed a clear distinction to be made between the roles each type of user is allowed to play, and the access each type of user is given. As before, a Use Case Diagram is still present for Students (Figure 3.1.5.3), but it now includes new interactions which were added to The Force between Deliverable 1 and 2, and allows for a clearer view of its unique interactions in comparison to the Administrators.

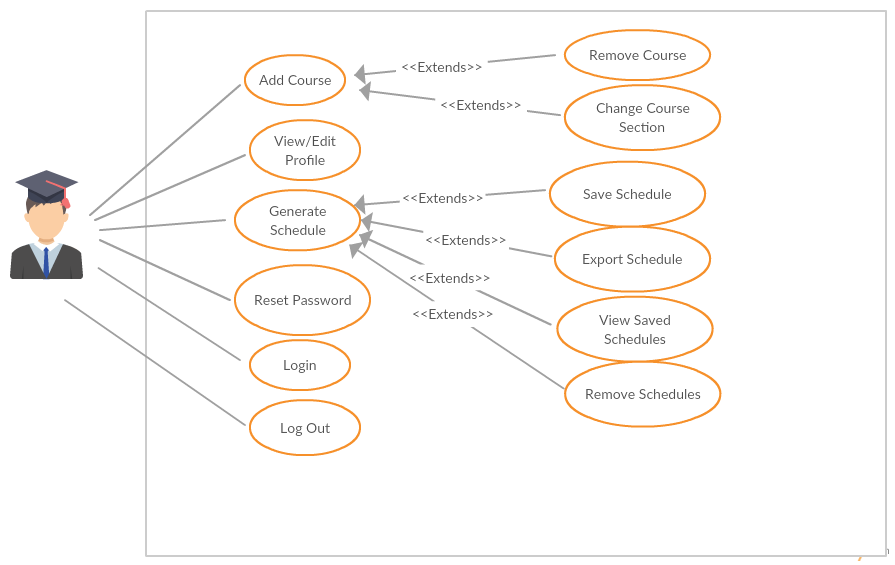
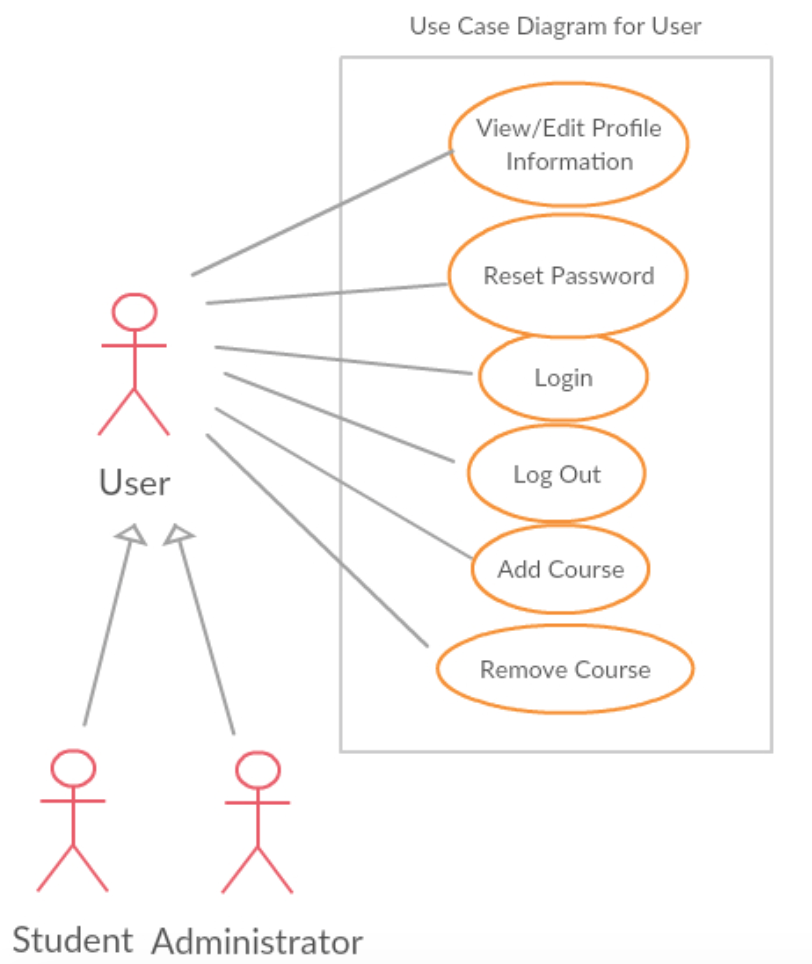


Figure 3.1.5.1. : General UCD from Deliverable 1

****

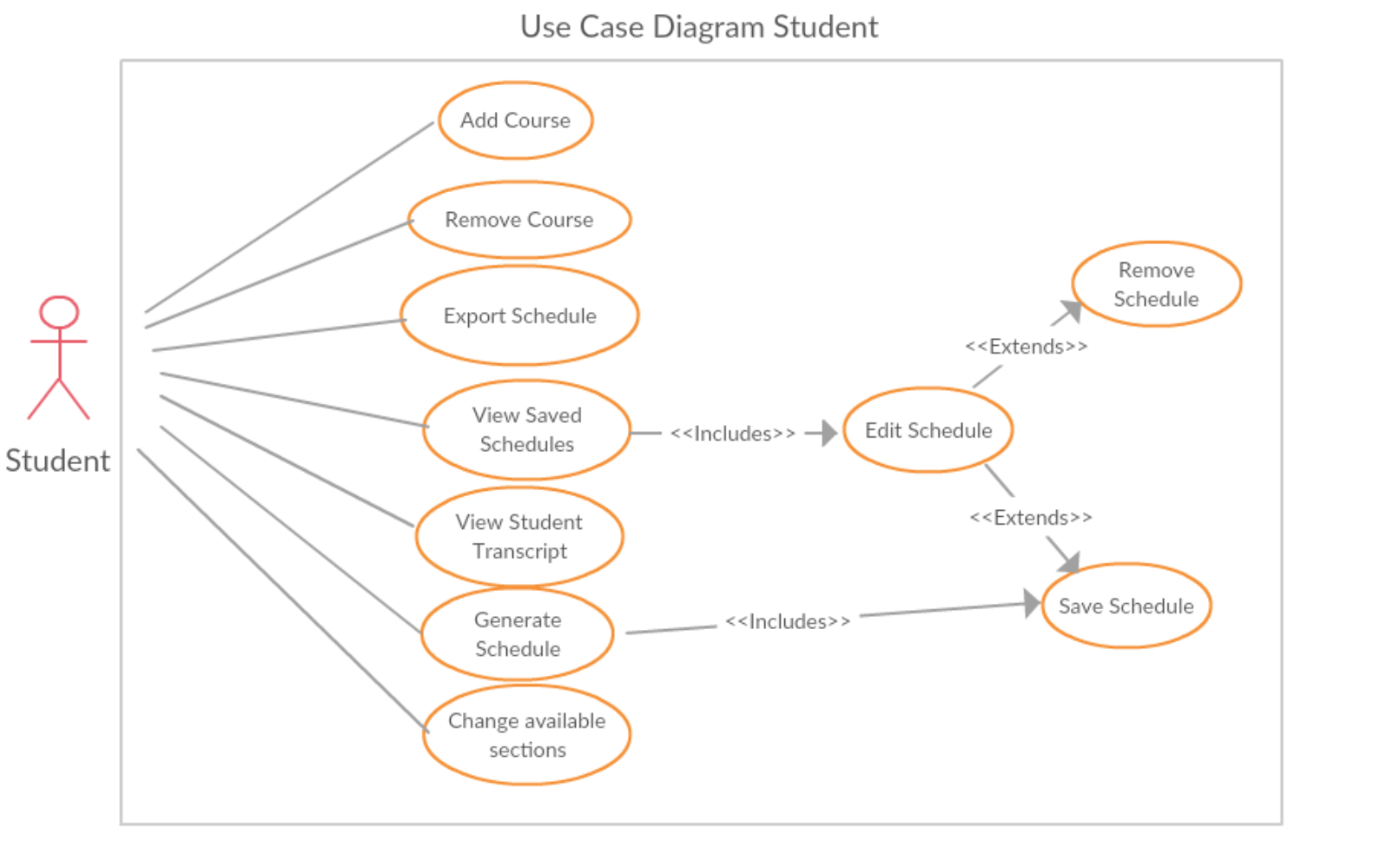
Figure 3.1.5.2. : General User UCD

Figure 3.1.5.3. : Student UCD

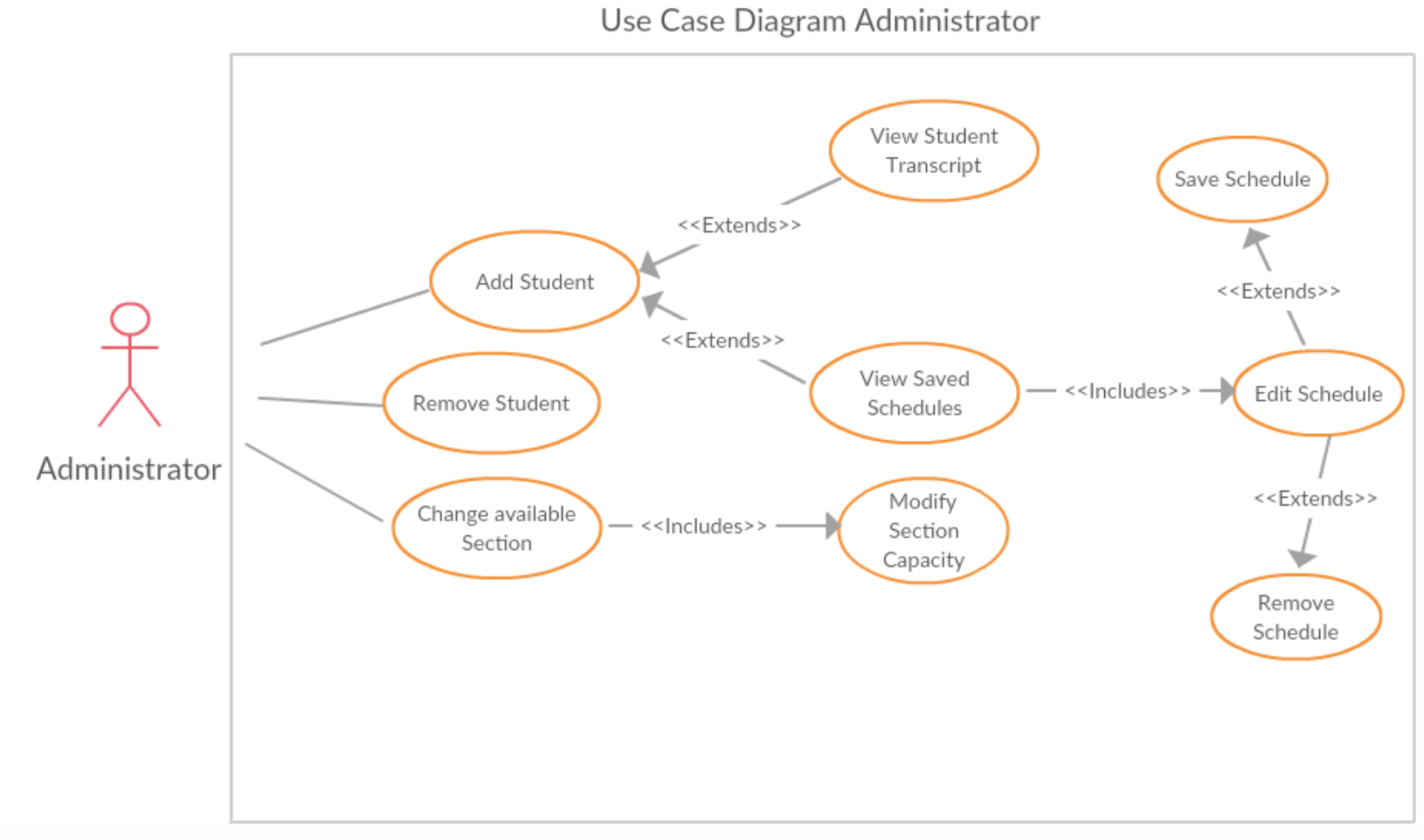
****

Figure 3.1.5.4. : Administrator UCD

**3.2. Subsystem Interfaces Specifications**

The system will be broken down into three separate subsystems in order to not only facilitate separation of concerns, but also to facilitate public interfacing. All methods referenced below will be those found in the class diagram used to represent Logical View (refer to section 3.1.1.). All constructors, as well as all getter and setter methods, are assumed whenever necessary (as all variables as private), but won’t be included in order to reduce redundancy. Another important note is that some methods, such as the inherited login and logout, are not particularly relevant to one interface or another, nor are they used by subsystems they’re not a part of. These methods will also be removed from the specifications to reduce clutter and increase cohesion.

**3.2.1. IStudentInfo**

|  |  |
| --- | --- |
| Class | Student |
| Methods: | * addCourse(Course course) * Add a given course to the current student’s course list. * Parameters: * course – The course to be added to the current semester’s course list. * Invalid values: NULL * Returns: void |
|  | * deleteCourse(Course course) * Remove a given course from the current student’s course list. * Parameters: * course – The course to be removed from the course list. * Invalid values: NULL * Returns: void |
|  | * exportSchedule() * Export the current student’s schedule. * Parameters: * None * Returns: void |

|  |  |
| --- | --- |
| Class | studentTranscript |
| Methods: | * displayTranscript() * Display current student’s transcript to be seen by the user. * Parameters: * None * Returns: void |

|  |  |
| --- | --- |
| Class | Administrator |
| Methods: | * removeStudent(Integer courseId, Integer studentId) * Remove the given student from the given course. Student’s course list is updated as well. * Parameters: * courseId – The course the student will be removed from * Invalid values: NULL. Student must be enrolled to be removed. * StudentId – The student to be removed * Invalid values: NULL * Returns: void |
|  | * addStudent(Integer courseId, Integer studentId) * Add the given student to the given course. Student’s course list is updated as well. * Parameters: * courseId – The course the student will be added to * Invalid values: NULL. Student must not be enrolled already. * StudentId – The student to be removed * Invalid values: NULL * Returns: void |
|  | * modifyCourse(Integer courseId, Integer newId) * Modify a given course’s courseId to a new value. * Parameters: * courseId – The course to be modified * Invalid values: NULL * newId – The new courseId * Invalid values: NULL * Returns: Course. Returns the updated Course |
|  | * modifySection(String sectionId, String newId) * Modify a given section’s sectionId to a new value. * Parameters: * sectionId – The section to be modified * Invalid values: NULL * newId – The new sectionId * Invalid values: NULL * Returns: Section. Returns the updated Section |
|  | * modifySchedule(Integer studentId) * Gain access to a given student’s schedule for future modifications. * Parameters: * studentId – The student whose schedule will be accessed. * Invalid values: NULL * Returns: Schedule. Returns the Schedule to be modified at a later time. |
|  | * modifyCapacity(String sectionId, Integer capacity) * Modify a given section’s maximum capacity. * Parameters: * sectionId – The section to be modified * Invalid values: NULL * capacity – The new capacity of the section * Invalid values: NULL, capacity < 0. Capacity must be defined, and can’t be a negative number. * Returns: void |
|  | * viewStudentTranscript(Integer studentId) * View a given student’s official transcript * Parameters: * studentId – The student whose transcript will be accessed. * Invalid values: NULL * Returns: void |

**3.2.2. IScheduler**

|  |  |
| --- | --- |
| Class | Scheduler |
| Methods: | * generateSchedule() * Generates a schedule for a student. * Parameters: * None * Returns: Schedule |
|  | * searchCourses(Course c) * Retrieves a section within Course c. * Parameters: * c – The course whose section is being identified * Invalid values: NULL * Returns: Section |
|  | * addCourse(String sectionId) * Add course with the given section, and sectionId, to the schedule. Eliminates the need to pass a course as well as a section. * Parameters: * sectionId – The id of the section found within the course to be added. * Invalid values: NULL * Returns: void |
|  | * RemoveCourse(Integer courseId, String sectionId) * Removes the course and section whose ids were passed, from the schedule. * Parameters: * courseId – The id of the course to be removed. * Invalid value: NULL * sectionId – The id of the section to be removed. * Invalid value: NULL * Returns: void |
|  | * editSection(String sectionOld, String sectionNew) * Find the section whose id is sectionOld, replace the id by sectionNew. * Parameters: * sectionOld – The section to be found. * Invalid value: NULL * sectionNew – The new id of the section. * Invalid value: NULL * Returns: void |

|  |  |
| --- | --- |
| Class | Schedule |
| Methods: | * viewSchedule() * Display current schedule for user to see. * Parameters: * None * Returns: void |
|  | * exportSchedule(Schedule s) * Export a given schedule to PDF format. * Parameters: * s – The schedule to be exported * Invalid value: NULL * Returns: void. |

**3.2.3. ICourse**

|  |  |
| --- | --- |
| Class | Course |
| Methods: | * getSections(Course c) * Retrieves all sections associated with a given course c. * Parameters: * c – The course which contains the sections required. * Returns: List<Schedule> |
|  | * addSection(Section section) * Add a section to the course calling the method. * Parameters: * section – The section to be added. * Invalid values: NULL * Returns: void |
|  | * deleteSection(String sectionId) * Deletes the section associated with the provided id. * Parameters: * sectionId – The id of the section to be removed. * Invalid values: NULL * Returns: void |
|  | * blockCourse() * Blocks the course from being registered to. * Parameters: * N/A * Returns: Boolean. Returns true if course is now blocked, false otherwise. |
|  | * allowCourse() * Allows the course to be registered to. * Parameters: * N/A * Returns: Boolean. Returns true if course is now allowed, false otherwise. |

|  |  |
| --- | --- |
| Class | Section |
| Methods: | * isFull() * Checks if the calling section is full. * Parameters: * N/A * Returns: Boolean. Returns true if the section is full, false otherwise. |
|  | * addStudent(Student student) * Add a student to the calling section * Parameters: * student – The student to be added * Invalid values: NULL * Returns: void |
|  | * removeStudent(Integer studentId) * Remove the student with the id being passed from the calling section. * Parameters: * studentId – The id of the student to be removed. * Invalid values: NULL * Returns: void |