Department of Computer Science & Software Engineering

Software Process: SOEN341/4S---2016

System Architecture and Design

The Schedulers	
Emili Vasseva	27526741
Sean Marcoux	27511876
Bruce Edouard Brazier	27419562
Dias Marat	27277911
Le Vinh Dang	26844987
Adriel Fabella	27466005
Gabriele Bavaro	27399103
Ying-Chen Chu	27415710
Alex Eladas	27041462
Salma Aly	27176414
Adil Hssaini	24832396
Nicolas Frazer-McKee	27068956

Table of Contents

Grading Sheet

Section	***Evaluation criteria (see instructions in the template for details)***	***Grading***
all	*10 marks are allocated for excellence, professionalism and quality of work above and beyond the correct meeting of specifications..*	*/10*
1	*Presentation of the document*	*/5*
2	*Introduction of the document*	*/1*
3.1 *.* *3.2*	*Validity and clarity of the architectural diagrams, as well as of the textual rationale.* *Validity, completeness, and clarity of description of each component interface.*	*/5* */4*
4.1 *4.2*	*Validity and clarity of the (UML) class diagrams or equivalent for each subsystem, as well as of the textual rationale.* *Validity and clarity and completeness of the class descriptions for each subsystem.*	*/8* */4*
5	*Validity and clarity of the dynamic design diagrams and contracts for each scenario.* *Compatibility of the scenarios with the components presented in section 2 and 3, as well as of the textual rationale.*	*/8*
6	*Revised cost estimation of each individual artifact, validity of explanation of cost estimation, total cost estimate*	*/2*
7	*Rapid Prototyping and Risk Report*	*/3*
Total		*/50*

DO NOT REMOVE THIS PAGE WHEN SUBMITTING YOUR DOCUMENT

2. Introduction

The present document provides a detailed description of the architecture and design of the scheduling system developed by team schedulers. The architecture of the system is represented through a “4+1” view that comprises logical, development, physical, and process views, in addition to scenarios that explain how actors, processes, and objects interact with each other.

Specification of the interactions between software interfaces and the components is discussed in the Subsystem interfaces specifications section, which describes how exchanges such as parameters passing, or function calls lead to service fulfillment. The design of each subsystem is subsequently detailed with class diagrams used to explain the internal structure of the subsystems, as well as a list of all attributes and functions in each class.

Dynamic design scenarios are depicted in the present document via a dynamic design of four use cases, namely: setpreferences, generateSchedule, addCourseinDB, and resetPasssword. While the first two uses cases are student specific, the third only pertains to the administrator, and the fourth is common between the two aforementioned actors.

A review of the cost estimate, prototyping, and risks comes at the end of the document to justify the validity of the overall software development process.

3. Architectural Design

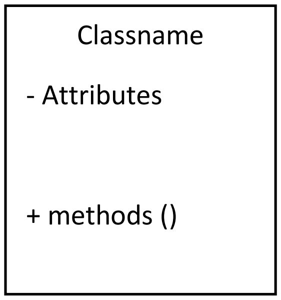
The “4+1 Architecture View” will be used in this section to show a detailed and updated version of the scheduling system, based on what has been developed and designed in deliverable 1. The “4+1 Architecture View” includes Logical, Development, Process, Physical and Scenarios Views. In the Logical View, we are going to present all the classes and their functions in the Class Diagram, and the Component Diagram will be shown in the Development View. Moreover, the Activity Diagram will be used in the Process view. And to implement the overview of the interaction between the system with the client, the Deployment Diagram will be presented in the Physical view. Lastly, the Use Cases Diagram will be shown in the Scenarios to help the clients understand more about the product.

3.1 Architecture Diagram

3.1.1 Logical View: Class Diagram Description

A class diagram is a static structure diagram which describes the system based on displaying its classes, their attributes, their methods, and the relationship between each class. The class diagram provides a good representation on the implementation of the software since it shows the types being modeled. Simply put, the class diagram of the system can be altered or refined.

A class in a class diagram is represented by a block divided into 3 sections.

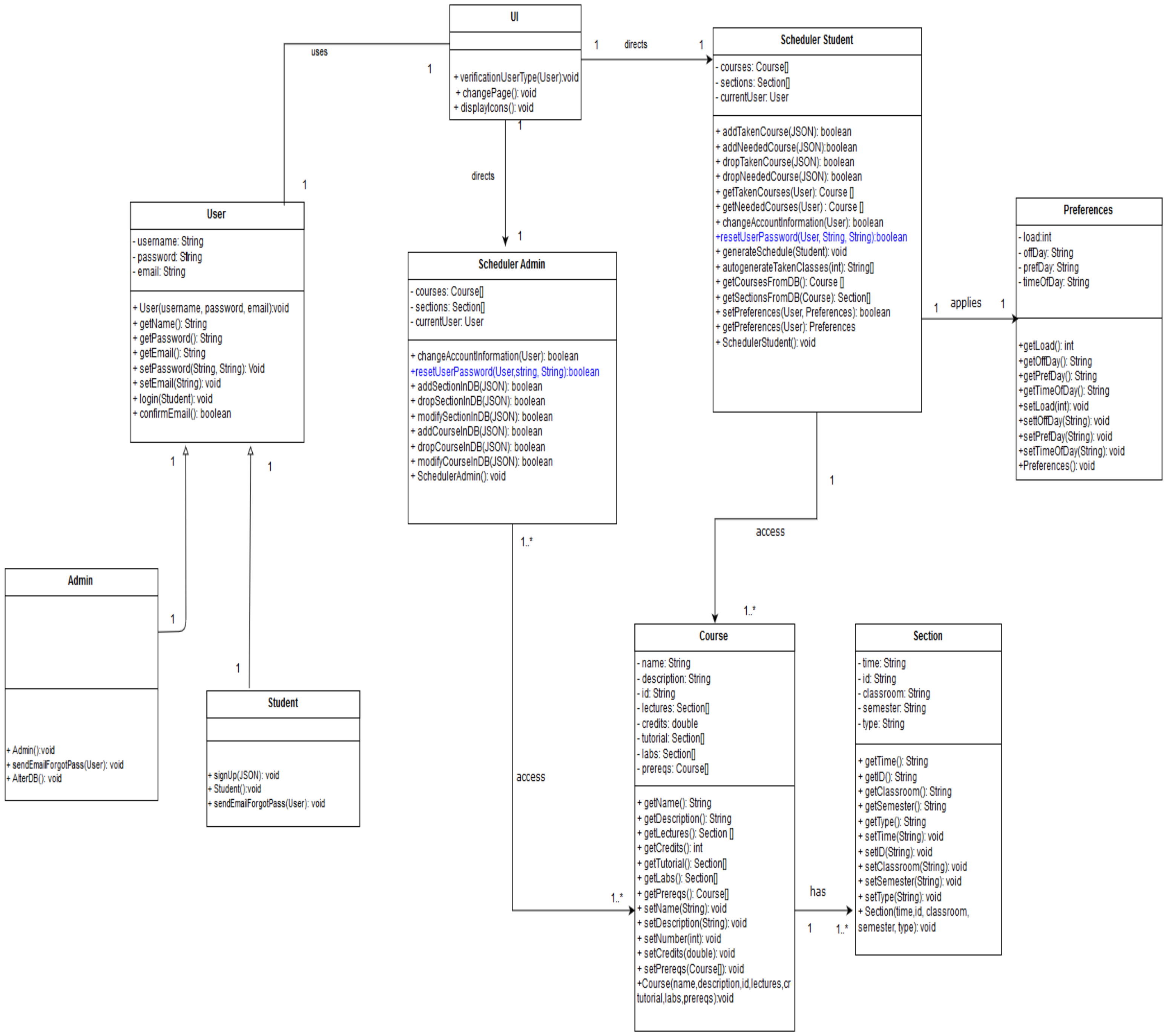


The block contains the **name of the class**, the **attributes** and the **methods**.

The **attributes** of a class describe the properties and characteristics that its object can have, while the **methods** pertain to the main actions and functions that the class can do. The signs, such as **-** and + represent accessibility level of a certain field or method in object oriented-programming. None the less, there also exists other types of accessibility levels such as default. It is conventional to display the attributes as private fields and the methods of public access.

The class diagram of the project portrays the system *Scheduler* as the main bridge to access functionality. The way for a certain user to access the Scheduler is to successfully login. From there, the UI will then redirect the user to the main page. The user interface also manages everything the client needs, meaning changing of pages, accepting requests or displaying information.

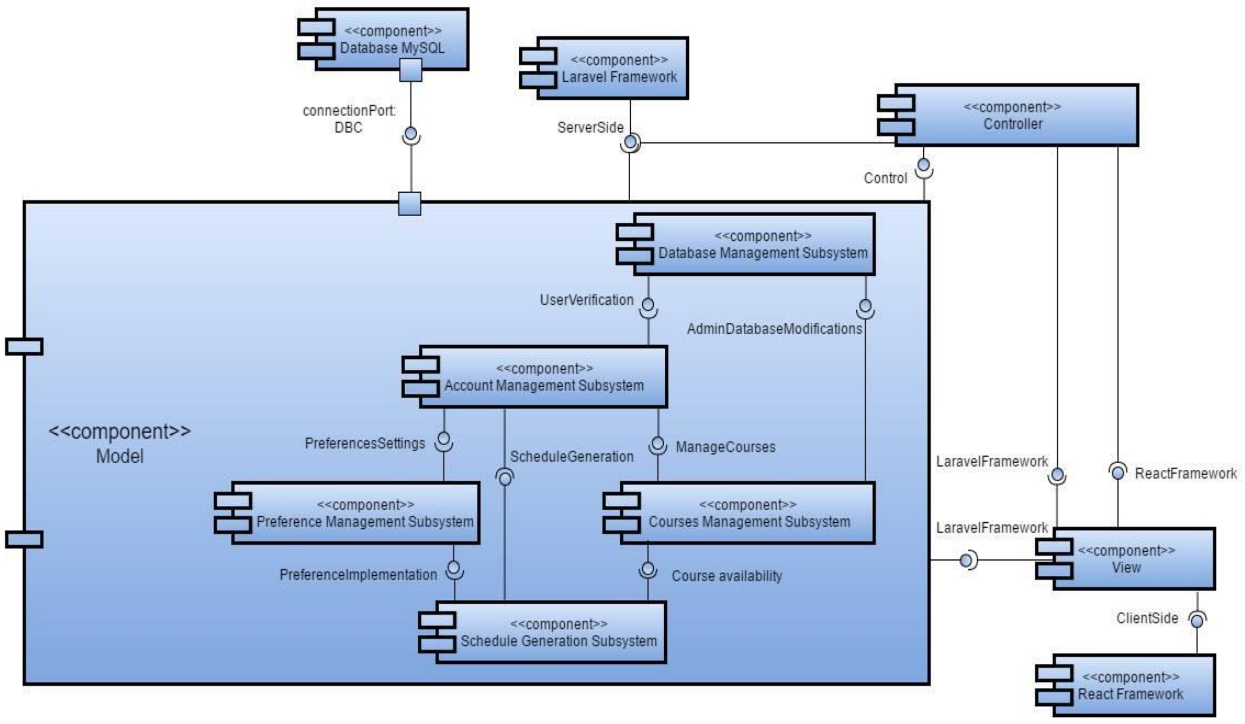
Class Diagram



UI
+ verificationUserType(User):void
+ changePage(): void
+ displayIcons(): void
uses
directs
1
Scheduler Student
- courses: Course[]
- sections: Section[]
- currentUser: User
+ addTakenCourse(JSON): boolean
+ addNeededCourse(JSON):boolean
+ dropTakenCourse(JSON): boolean
+ dropNeededCourse(JSON): boolean
+ getTakenCourses(User): Course []
+ getNeededCourses(User) : Course []
+ changeAccountInformation(User): boolean
+resetUserPassword(User, String, String):boolean
+ generateSchedule(Student): void
+ autogenerateTakenClasses(int): String[]
+ getCoursesFromDB(): Course []
+ getSectionsFromDB(Course): Section[]
+ setPreferences(User, Preferences): boolean
+ getPreferences(User): Preferences
+ SchedulerStudent(): void
applies
Preferences
- load:int
- offDay: String
- prefDay: String
- timeOfDay: String
+getLoad(): int
+getOffDay(): String
+getPrefDay(): String
+getTimeOfDay(): String
+setLoad(int): void
+setOffDay(String): void
+setPrefDay(String): void
+setTimeOfDay(String): void
+Preferences(): void
User
- username: String
- password: String
- email: String
+ User(username, password, email):void
+ getName(): String
+ getPassword(): String
+ getEmail(): String
+ setPassword(String, String): Void
+ setEmail(String): void
+ login(Student): void
+ confirmEmail(): boolean
Scheduler Admin
- courses: Course[]
- sections: Section[]
- currentUser: User
+ changeAccountInformation(User): boolean
+resetUserPassword(User,string, String):boolean
+ addSectionInDB(JSON): boolean
+ dropSectionInDB(JSON): boolean
+ modifySectionInDB(JSON): boolean
+ addCourseInDB(JSON): boolean
+ dropCourseInDB(JSON): boolean
+ modifyCourseInDB(JSON): boolean
+ SchedulerAdmin(): void
access
1..*
Admin
+ Admin():void
+ sendEmailForgotPass(User): void
+ AlterDB(): void
Student
+ signUp(JSON): void
+ Student():void
+ sendEmailForgotPass(User): void
Course
- name: String
- description: String
- id: String
- lectures: Section[]
- credits: double
- tutorial: Section[]
- labs: Section[]
- prereqs: Course[]
+ getName(): String
+ getDescription(): String
+ getLectures(): Section []
+ getCredits(): int
+ getTutorial(): Section[]
+ getLabs(): Section[]
+ getPrereqs(): Course[]
+ setName(String): void
+ setDescription(String): void
+ setNumber(int): void
+ setCredits(double): void
+ setPrereqs(Course[]): void
+Course(name,description,id,lectures,cr
tutorial,labs,prereqs):void
has
Section
- time: String
- id: String
- classroom: String
- semester: String
- type: String
+ getTime(): String
+ getID(): String
+ getClassroom(): String
+ getSemester(): String
+ getType(): String
+ setTime(String): void
+ setID(String): void
+ setClassroom(String): void
+ setSemester(String): void
+ setType(String): void
+ Section(time,id, classroom,
semester, type): void

3.1.2 Development View

The development view describes our scheduling system from our programmer's perspective and the management of our software. It is illustrated by the component diagram which consists of components and links that show the dependencies between them.



<<component>> Database MySQL
<<component>> Laravel Framework
<<component>> Controller
connectionPort: DBC
ServerSide
Control
<<component>> Database Management Subsystem
UserVerification
AdminDatabaseModifications
<<component>> Account Management Subsystem
<<component>> Model
PreferencesSettings
ScheduleGeneration
ManageCourses
<<component>> Preference Management Subsystem
<<component>> Courses Management Subsystem
PreferenceImplementation
Course availability
<<component>> Schedule Generation Subsystem
LaravelFramework
ReactFramework
LaravelFramework
<<component>> View
ClientSide
<<component>> React Framework

Figure 1: Component Diagram

The main components of our system consist of the model, view and controller. Since these 3 components are not taken by the same framework, we have specified on the component diagram that the view is handled using React and that data manipulation and database queries are handled by Laravel. Most importantly, in our model, we have defined our subsystems to be account management, courses management, preferences management, schedule generation and database management. The different subsystems interact with each other through interfaces, with the open end of the link being a required interface and the lollipop being the provided interface. Also, the model is connected to a MySQL database.

In our component diagram the two frameworks, Laravel for server side model and controller, and react for the client side view, are described. The components of the model view controller system interact as following: The model requires connection to the SQL database through Database Connection port and control instructions provided by the controller. The view requires object oriented and control information in order to present the user with the system on a browser. The controller requires input from the user through the view.
Within the model, the database management subsystem can provide the user credentials in order to manage the account subsystem and allow a confirmed admin to modify the database. The account subsystem provides preferences and course management subsystems for students. Each of these subsystems (course management and preferences) each provide their own criteria to a schedule generator that will use the inputs in order to provide a schedule.

3.1.3 Physical View

Deployment diagram will be used to illustrate the physical view of the system. The application server where the Web server and the Database server are located. The Web server and the Database server will interact with each other to get the requested information. MySQL is a tool to get the information stored in the Database server.

The client component can use any devices such as Apple device, Windows Device or Linux device that come with a web browser like Mozilla Firefox, Google Chrome, Safari or Internet Explorer to connect to the server.

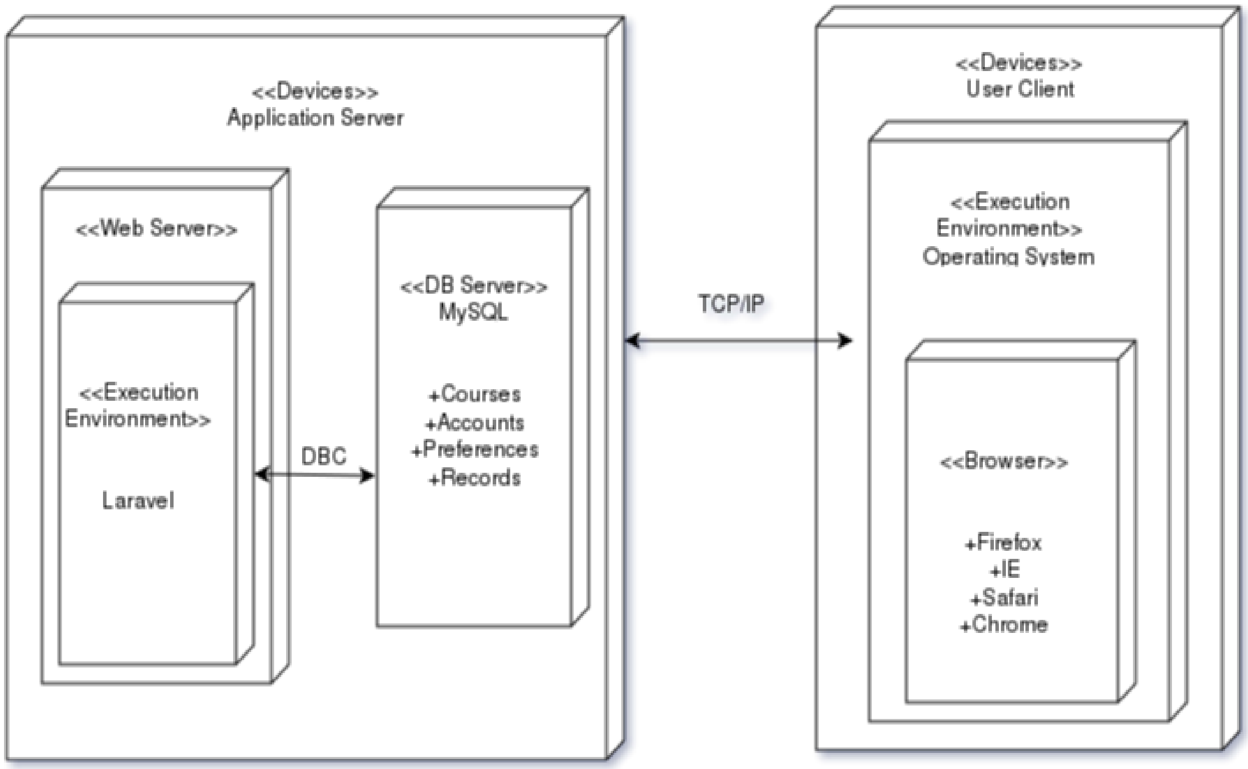


Figure 2: Physical View: Deployment Diagram.

3.1.4 Process View

The purpose of these diagrams are to show the path of system when actions and decisions are made by Users or Admins order to complete a specific task. In this case, showing the steps to generate a schedule and update account information are shown for the Student diagram. As for the admin diagram, representing the activities required in order to modify the database or to change a student's information can be observed.

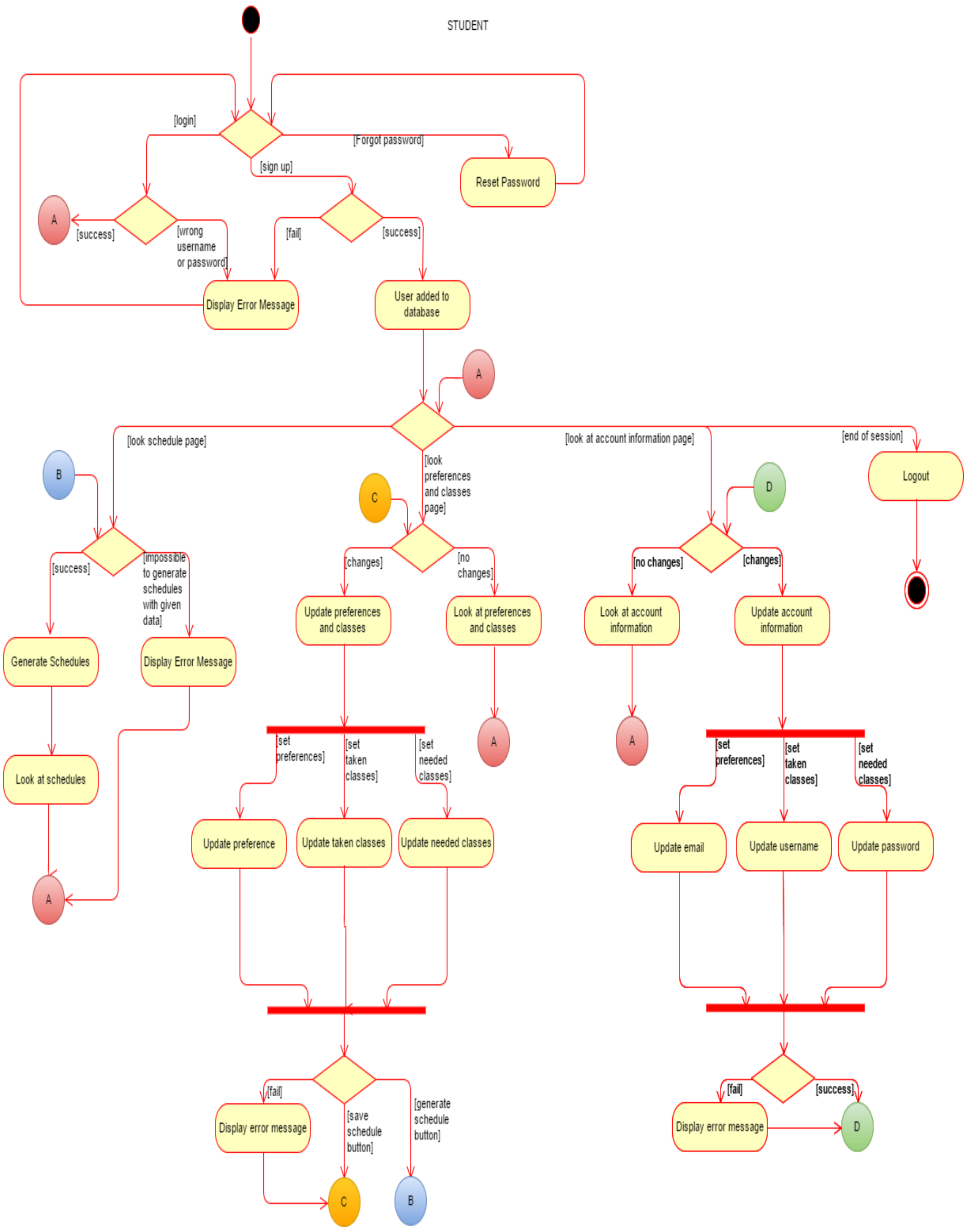


Figure 3: Student Activity diagram

The first activity diagram demonstrates the generation of a schedule by a User using The Scheduler system. Upon entering the website, the student is prompted with the decision to login using a username and password or create a new account. If the users are on their first visit, they will choose to sign up, otherwise, they can simply login. If the users input wrong data, an error message will be displayed. The option to reset the password can then be used. Once all the required information are given, the Scheduler opens the menu page, giving the choice to, have a look at the schedule, access the preferences and classes, open the account information page or logout.

If the users opt to look at their preferences and classes, they can either select to update them or to simply display them. If the former is chosen, the user will be prompted to set their preferences, classes and the needed classes. An error message will be displayed if anything went wrong and the system will bring back the users to update the preferences again, otherwise the choices will be saved or the schedule can be generated, which brings the user to the schedule display page.

In the event that the student decides to look at the schedule, it will either provide the user with an option to generate a schedule or it will display an error due to conflicting data, such as time conflicts for courses. After generating the schedule, the user can take a look at it and then choose to go back to the menu page.

The last option available on the menu page is the account information. If the users decide to choose this option, they can either update or simply display their account information. The information that are to be saved for the account are the email, username and password. A display error will be seen if there were any problems. Once the users are done using the Scheduler, they can logout.

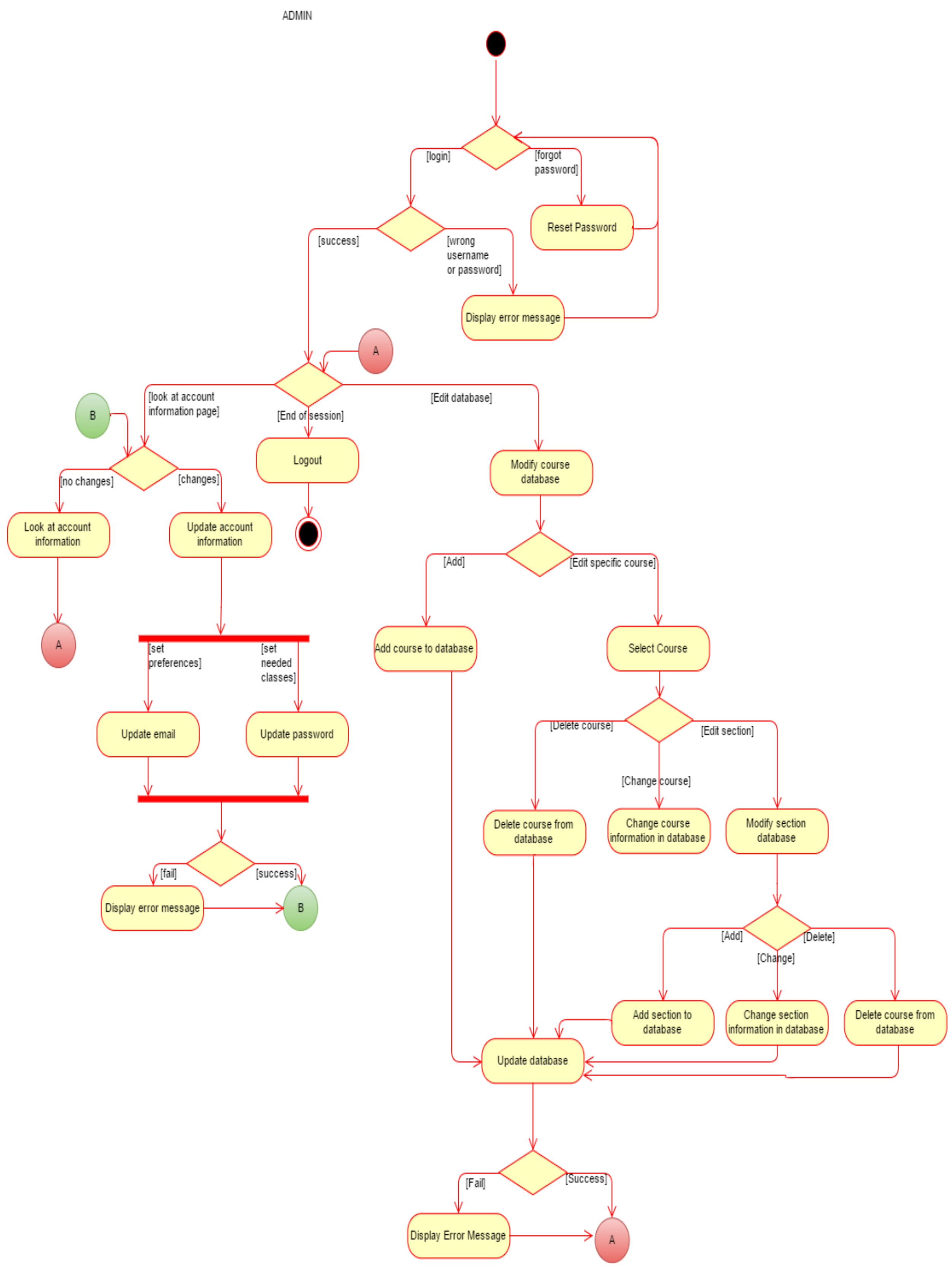


Figure 4: Admin Activity diagram

The second activity diagram shows the Admin’s rights over the Scheduler system’s database. When accessing the system, the admins also have to login. Successfully signing in to the website, brings up the menu page. The choices here are to open the account information page, which contains the students account information, to modify the course database and to logout.

In order to modify the database, the scheduler either requests to add a course or to edit a specific course. Choosing to select a course requires the admin to either delete it from the database, alter its description and information or modify the sections. The last option consists of adding deleting and changing information of the course section. After finishing working on the database, all the changes are saved.

For the account information page, this is used for any student that is unable to access the Scheduler for any reason and requires the assistance of an Admin. The Admin can look at the student’s information and update their email or password. Upon modifying their login information, an error will be displayed if anything went wrong. If the changes were successful, the admin will be brought back to the account information page. Once the admins are done using the system, they can simply logout.

3.1.5 Scenarios

In the “4+1” architecture, the "1" stands for the *scenarios*. This section describes and illustrates the interactions between objects and process, or better said, between actors and use cases. The use case diagrams represent the functionalities and requirements of the system, such as generating a schedule, managing the account information, etc.

The system three types of users: public users, students, and admin. The *public users* can sign up into the system, and become a *student*, therefore interact with the system. As said before, the system goals is to generate a schedule out of the *student* preferences, course choices for the current semester and his own personalized list of taken classes. On the other side, the *admin* does not have the same functionalities as the *student* and a *public user*. First, an admin cannot sign in, as his/her account is already in the database. Second, the *admin*'s system goal is to mainly manipulate the course and section database. This includes adding, deleting, and modifying data. However, the *admin* and the *student* still shares some functionalities; both can login, edit their account (except the *admin* cannot change his username), reset their password if forgotten, login and logout.

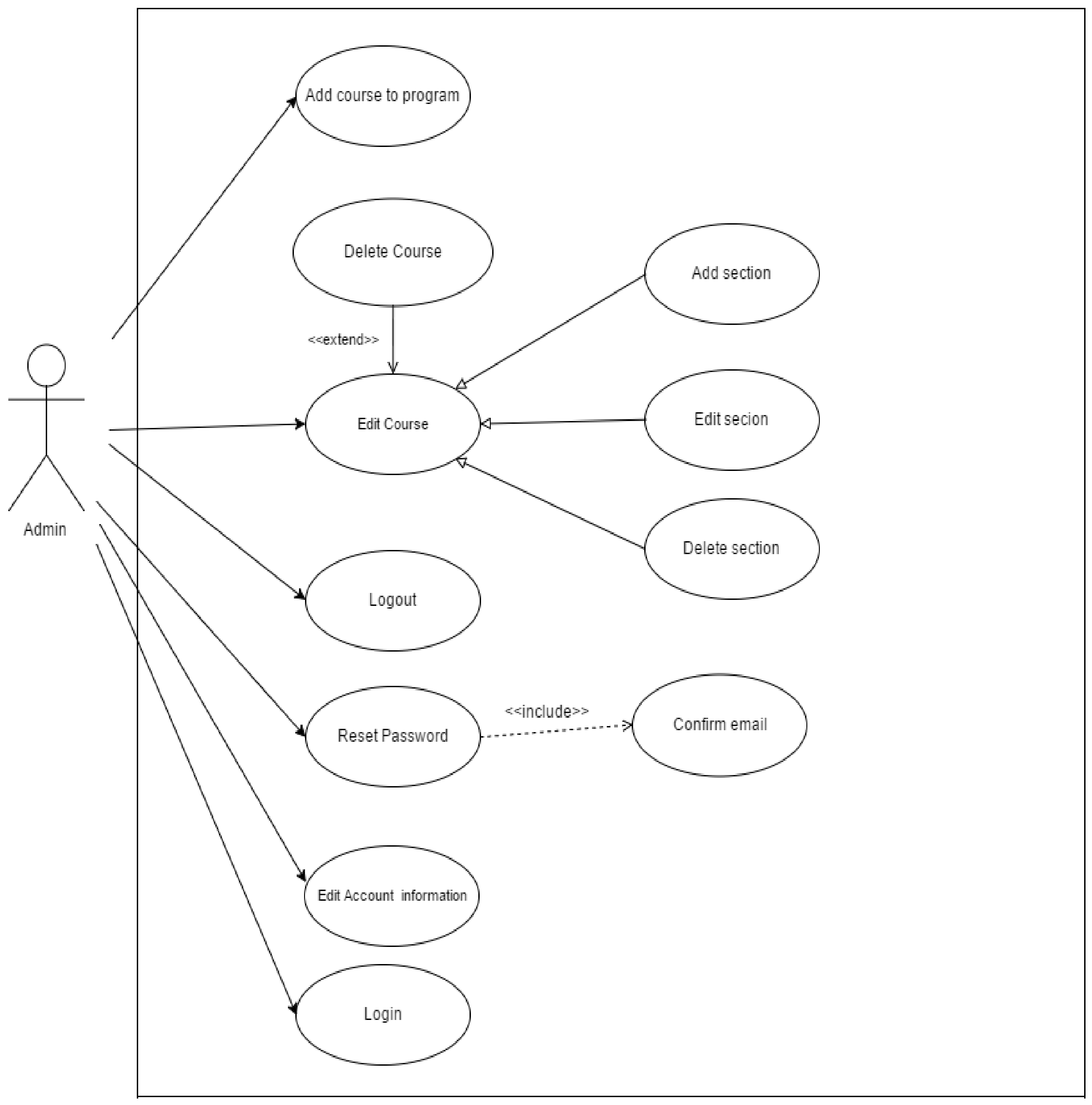


Figure 5: Admin Use Case Diagram

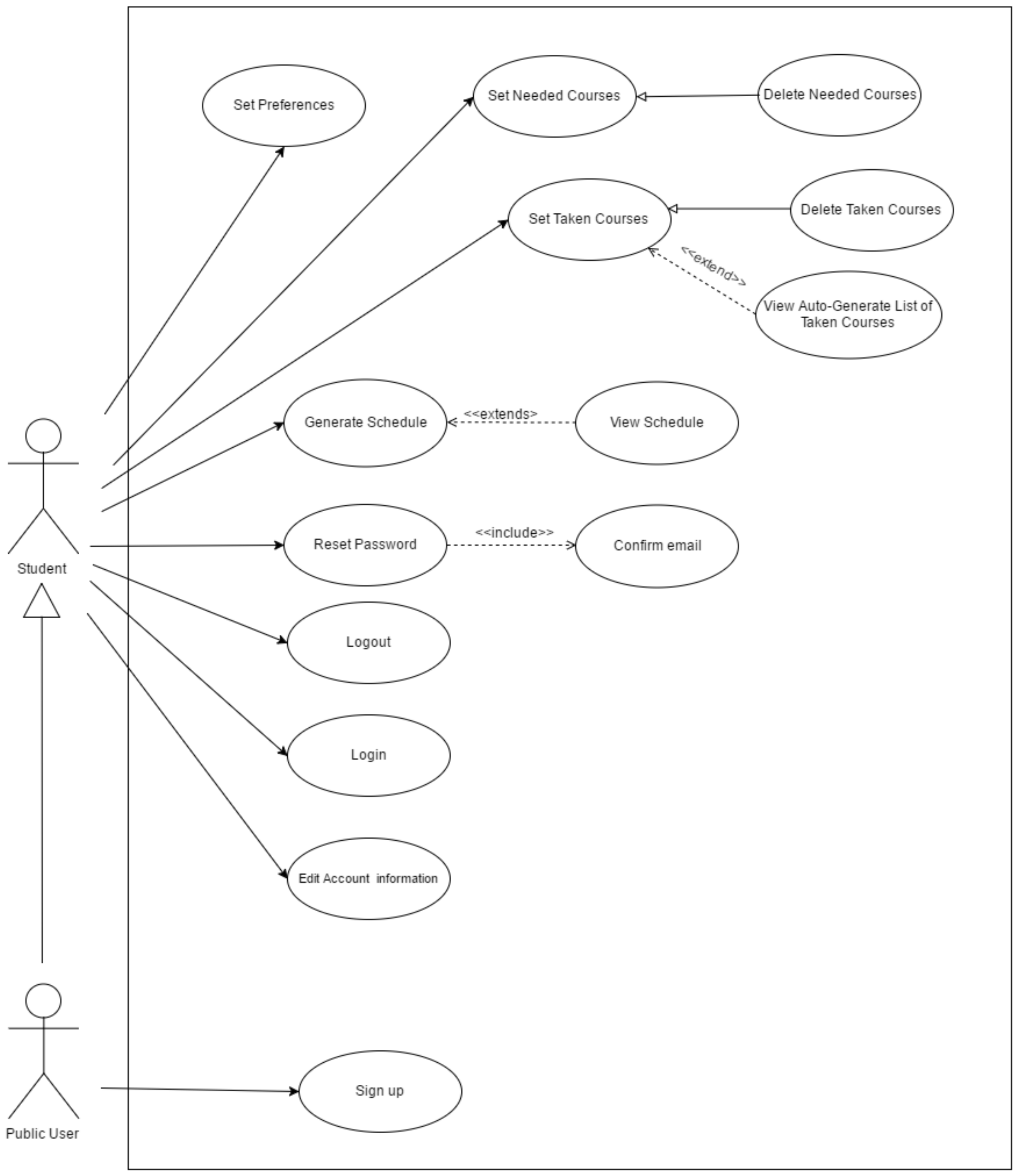


Figure 6: Student Use Case Diagram

3.2 Subsystem Interfaces Specifications

The subsystem interfaces consist of: ManageCourses, PreferenceSettings, UserVerification, CourseAvailabilty, PreferenceImplementation, ScheduleGeneration. Each of the interface connect two subsystems together. In this section, each of these interfaces will be described and the function calls exchanged between its subsystems will be given along with the specific description of the parameters passed.

3.2.1 ManageCourses

The manage courses interface joins the provided interface of account management with the required interface of the subsystem courses management.

Classes involved	SchedulerStudent, Course
List of Methods	1. addTakenCourses(JSON): boolean **Implemented in Class**: Scheduler Student **Description**: Method to add courses to the list of taken courses. **Input Parameter(s)**: serialized Course -course to be added **Return Type**: boolean 2. getTakenCourses(User): Course[] **Implemented in Class**: Scheduler Student **Description**: Method to get the list of courses taken. **Input Parameter(s)**: User of type Student -user making inquiry **Return Type**: an array of type Course 3. addNeededCourses(JSON): boolean **Implemented in Class**: Scheduler Student **Description**: Method to add courses to the list of needed courses. **Input Parameter(s)**: serialized Course -courses to be aded **Return Type**: boolean 4. getNeededCourses(User): Course[] **Implemented in Class**: Scheduler Student **Description**: Method to get the list of courses needed. **Input Parameter(s)**: User type Student -user making inquiry **Return Type**: an array of type Course 5. autogenerateTakenClasses(int): String[]

	Implemented in Class: Scheduler Student **Description**: Method to generate a list of taken courses. **Input Parameter(s)**: int -number of classes t **Return Type**: an array of Strings -names of courses 6. getCoursesFromDB(int): void **Implemented in Class**: Scheduler Student **Description**: Method to access courses in database. **Input Parameter(s)**: none **Return Type**: N/A

3.2.2 PreferencesSettings

The preference settings interface joins the provided interface of account management with the required interface of the preferences management.

Classes involved	User Student, Preferences, Scheduler Student
List of Methods	1. setPreferences(User, Preferences): boolean **Implemented in Class**: Scheduler Student **Description**: Method to set the preferences for a user of type Student **Input Parameter(s)**: object of type Preferences and User of type Student -current user and their new preferences **Return Type**: boolean 2. getPreferences(User): Preferences **Implemented in Class**: Scheduler Student **Description**: Method to get the preferences **Input Parameter(s)**: User of type Student -current user and their preferences **Return Type**: An object of type Preferences

3.2.3 UserVerification

The UserVerification interface is provided by the user account in order to manage the database. The credentials entered in the UI are compared to the User's in order to establish a valid login type.

Classes involved	User
List of Methods	1. getName() : String **Implemented in Class**: User **Description**: Method to access the name of a user. **Input Parameter(s)**: N/A **Return Type**: String 2. getPassword() : String **Implemented in Class**: User **Description**: Method to access the password of a user. **Input Parameter(s)**: N/A **Return Type**: String 3. getEmail() : String **Implemented in Class**: User **Description**: Method to access the email of a user. **Input Parameter(s)**: N/A **Return Type**: String 4. login(Student) : void **Implemented in Class**: User **Description**: Method to access the student object associated with a user account. **Input Parameter(s)**: object of type Student **Return Type**: void

Verification of user login type is used to ensure proper access to the database.

Classes involved	UI
List of Methods	1. verificationUserType(User) : void **Implemented in Class**: UI **Description**: Method to access the user's dynamic type Student/Admin. **Input Parameter(s)**: object of type User -user to be verified **Return Type**: void

Enables an Admin to modify any User's information, allow Student to modify its own information.

Classes involved	Scheduler Student, Scheduler Student

List of Methods	1. changeAccountInformation(User) : boolean **Implemented in Class**: Scheduler Admin/Student **Description**: Method to modify a user's account information. **Input Parameter(s)**: object of type User -user who is verified as either Admin or Student. **Return Type**: boolean

3.2.4 AdminDataBaseModifications

This interface allows an administrator to edit the courses and sections list in the database. This is not accessible for a student because of the UserValidation interface.

Classes involved	Scheduler Admin, User Admin
List of Methods	1. getCoursesFromDB(): void **Implemented in Class**: Scheduler Admin **Description**: Method to access all Courses in database. **Input Parameter(s)**: N/A **Return Type**: void 2. addCourseInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to add a Course in database. **Input Parameter(s)**: serialized Course -to be added **Return Type**: boolean 3. dropCourseInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to remove a Course from database. **Input Parameter(s)**: serialized Course -to be removed **Return Type**: boolean 4. modifyCourseInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to modify an existing Course in database. **Input Parameter(s)**: serialized Course -to be modified **Return Type**: boolean 5. modifySectionInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to modify an existing section of a course in database. **Input Parameter(s)**: serialized Section -to be modified **Return Type**: boolean

	6. addSectionInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to add a Section for a Course in database. **Input Parameter(s)**: serialized Section -to be added **Return Type**: Boolean 7. dropSectionInDB(JSON) : boolean **Implemented in Class**: Scheduler Admin **Description**: Method to remove a section for a course. **Input Parameter(s)**: serialized Section -to be removed **Return Type**: boolean

3.2.5 CourseAvailability

The Course availability Interface generates, from a list of Courses that a student may take, a list of course Sections that could match together in a schedule. This interface provides viewing of courses list, not modification.

The Scheduler Student provides a list of courses needed and their sections. This list is solely based on which course could be taken for the schedule; semester and prerequisites are the only constraints.

Classes involved	Scheduler Student, User Student, Course
List of Methods	1. getTakenCourses(User) : Course[] **Implemented in Class**: Scheduler Student **Description**: Method to access the list of courses taken. **Input Parameter(s)**: User of type Student -who has taken the courses **Return Type**: array of objects of type Course 2. getNeededCourses(User) : Course[] **Implemented in Class**: Scheduler Student **Description**: Method to access the list of needed courses. **Input Parameter(s)**: User of type Student -who needs the courses **Return Type**: array of objects of type Course 3. getCoursesFromBD() : void **Implemented in Class**: Scheduler Student **Description**: Method to access the list of all courses. **Input Parameter(s)**: none **Return Type**: N/A

	4. getSectionsForCourse(String) : Section[] **Implemented in Class**: Scheduler Student **Description**: Method to access the list of sections for a course. **Input Parameter(s)**: String -name of course **Return Type**: array of objects type Section

The Course attributes are accessed in order to create the schedule, more importantly they provide a list of Sections.

Classes involved	Course, Section
List of Methods	1. getName(): String **Implemented in Class**: Course **Description**: Method to access the name of a course. **Input Parameter(s)**: N/A **Return Type**: string 2. getNumber(): int **Implemented in Class**: Course **Description**: Method to access the number of a course. **Input Parameter(s)**: N/A **Return Type**: int 3. getPrereqs(): Courses[] **Implemented in Class**: Course **Description**: Method to access the list prerequisite courses of a course. **Input Parameter(s)**: N/A **Return Type**: array of objects of type Course -sections not initialized 4. getLectures(): Section[] **Implemented in Class**: Course **Description**: Method to access the list of lectures for a course. **Input Parameter(s)**: N/A **Return Type**: array of objects of type Course Section 5. getTutorial(): Section[] **Implemented in Class**: Course **Description**: Method to access the list of tutorials for a course. **Input Parameter(s)**: N/A

	Return Type: array of Course objects of type Section 6. getLabs(): Section[] **Implemented in Class**: Course **Description**: Method to access the list labs of a course. **Input Parameter(s)**: N/A **Return Type**: array of Course objects of type Section

Sections are contained within Courses, stored in the arrays lecture, tutorial and Lab -from which they get their section type. Sections provide availability and more precise scheduling information.

Classes involved	Section
List of Methods	1. getSemester(): String **Implemented in Class**: Section **Description**: Method to access the semester in which the section is taught. **Input Parameter(s)**: N/A **Return Type**: string

3.2.6 PreferenceImplementation

The preference implementation interface provides the student preferences in order to generate a schedule with the current student preferences.

Classes involved	Scheduler Student, User
List of Methods	1. getPreferences(User) : Preferences **Implemented in Class**: Scheduler **Description**: Method to access the preferences of a User of type Student. **Input Parameter(s)**: User type Student -who will access their preferences **Return Type**: object of type Preferences

3.2.7 ScheduleGeneration

The schedule generation interface is a combination of the course availability and the preferences of a student. A schedule is generated from a list of sections for the courses to be taken that are offered during the semester and have times corresponding with preferences. The Schedulegeneration then chooses sections from the lists that fulfil the requirements to be added to a schedule until it is filled. This schedule can then be displayed with section information - times and classrooms. In order to modify the schedule, preferences or courses to be taken can be modified to generate a new schedule.

Classes involved	Scheduler Student, Student
List of Methods	1. generateSchedule(Student) : void **Implemented in Class**: Scheduler **Description**: Method to produce a schedule for a semester according to a list of courses and preferences. **Input Parameter(s)**: object of type Student **Return Type**: void.

More Information is required in order to draw a schedule: the course names, number and section.

Classes involved	Course, Section
List of Methods	1. getName(): String **Implemented in Class**: Course **Description**: Method to access the name of a course. **Input Parameter(s)**: N/A **Return Type**: string 2. getNumber(): int **Implemented in Class**: Course **Description**: Method to access the number of a course. **Input Parameter(s)**: N/A **Return Type**: int 3. getCredits(): double **Implemented in Class**: Course **Description**: Method to access the number of credits of a course. **Input Parameter(s)**: N/A **Return Type**: double 4. getPrereqs(): Courses[] **Implemented in Class**: Course **Description**: Method to access the list prerequisite courses of a course. **Input Parameter(s)**: N/A **Return Type**: array of objects of type Course -sections not initialized

	5. getLectures(): Section[] **Implemented in Class**: Course **Description**: Method to access the list of lectures for a course. **Input Parameter(s)**: N/A **Return Type**: array of objects of type Course Section 6. getTutorial(): Section[] **Implemented in Class**: Course **Description**: Method to access the list of tutorials for a course. **Input Parameter(s)**: N/A **Return Type**: array of Course objects of type Section 7. getLabs(): Section[] **Implemented in Class**: Course **Description**: Method to access the list labs of a course. **Input Parameter(s)**: N/A **Return Type**: array of Course objects of type Section

Still more Information is required in order to draw a schedule: the section IDs, Times, Types, Classrooms.

Classes involved	Section
List of Methods	1. getID(): String **Implemented in Class**: Section **Description**: Method to access the id of a section. **Input Parameter(s)**: N/A **Return Type**: string 2. getTime(): String **Implemented in Class**: Section **Description**: Method to access the time of a section. **Input Parameter(s)**: N/A **Return Type**: string 3. getClassroom(): String **Implemented in Class**: Section **Description**: Method to access the location of a section. **Input Parameter(s)**: N/A **Return Type**: stringA **Return Type**: string

4. Detailed Design

4.1 Detailed Design Diagram

The following section describes a detailed description of the concerning the class diagram of the system. The scheduler contains a total of 8 classes. The classes *Student, Admin* and *User* represent essentially the user. *Student* and *Admin* are subclasses of the *User* class. This corresponds to the possibility of a student and an admin to log in to the system. The *User* class interacts with the *UI* class where this class simply manages the information to be displayed. The classes *Scheduler - Admin* and *Scheduler-Student*. Depending on the type of user, the UI will redirect that user the appropriate *Scheduler* system. These classes are the core of the system since they provide functionality with the help of the other classes *Preferences, Classes & Sections.*

4.1.1 Student Generate Schedule

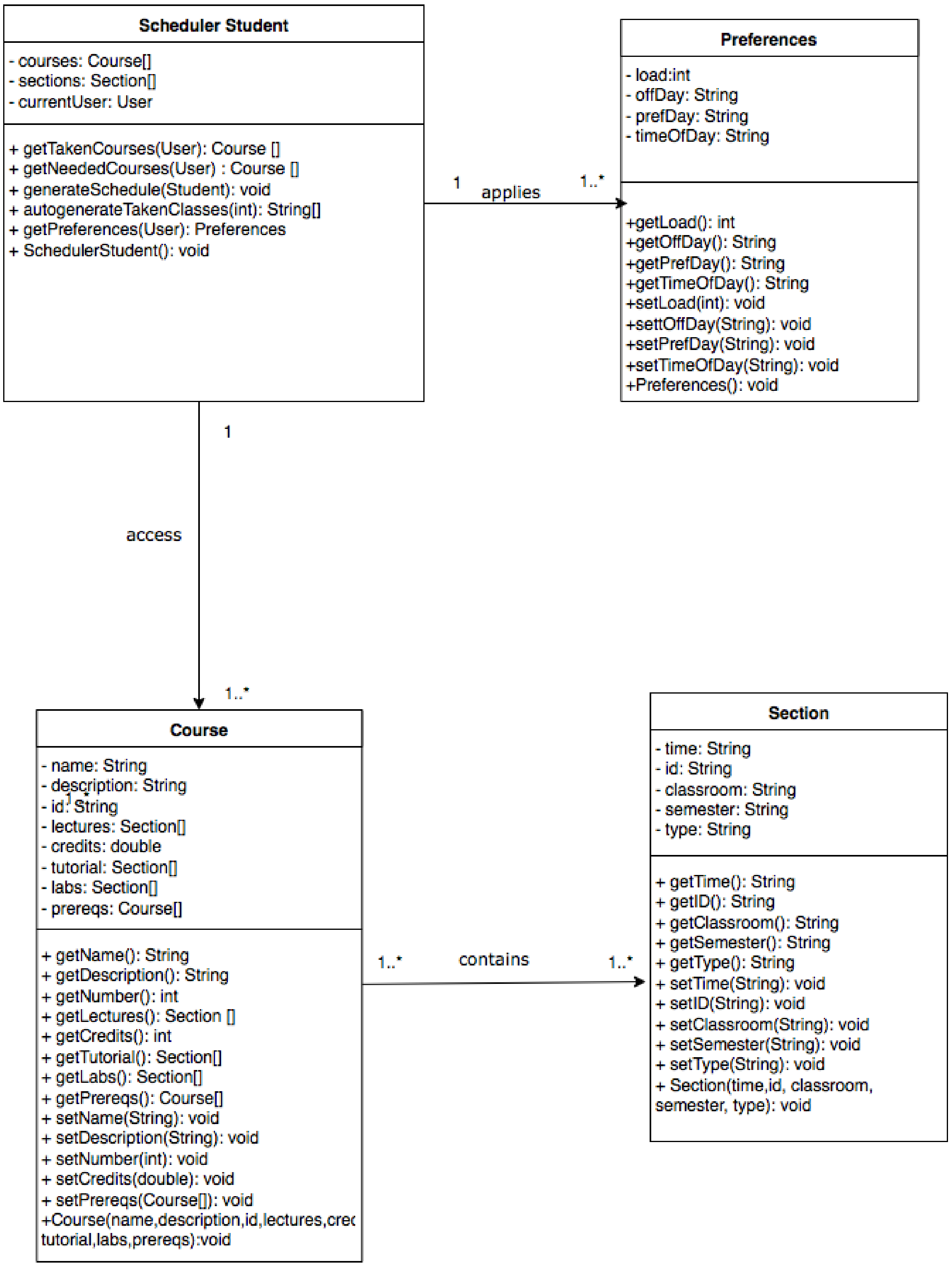
The *Student Scheduler* has a method called *generateSchedule(Student)* which generates a schedule based on the courses, preferences and sections. A method called *autogenerateTakenClasses(int)* shall automatically display the courses that are added even before finalization. The Student Scheduler contains Preferences, Course, Section and Scheduler Student. Each of these classes communicate between each other to generate schedule.

Student generated schedule is provided as part of the schedule generation component, the method listed below are elaborated:

Classes Involved	**Preferences, Course, Section, Scheduler Student**
Method(s) Implemented	generateSchedule(Student): void **Implemented in Class**: Scheduler Student **Description**: Method that generates schedule for Student **Input Parameter(s)**: An object Student **Return Type**: void getTakenCourses(User): Course [] **Implemented in Class**: Scheduler Student **Description**: Method that retrieves taken courses from database **Input Parameter(s)**: User **Return Type**: void

	getNeededCourses(User): Course [] **Implemented in Class**: Scheduler Student **Description**: Method that returns needed courses **Input Parameter(s)**: User **Return Type**: Course [] autogenerateTakenClasses(int): String [] **Implemented in Class**: Scheduler Student **Description**: Method that autogenerates taken classes **Input Parameter(s)**: int **Return Type**: String [] getPreferences(User):Preferences **Implemented in Class**: Scheduler Student **Description**: Method that returns preferences of a user **Input Parameter(s)**: User **Return Type**: Preferences

Student Generate Schedule



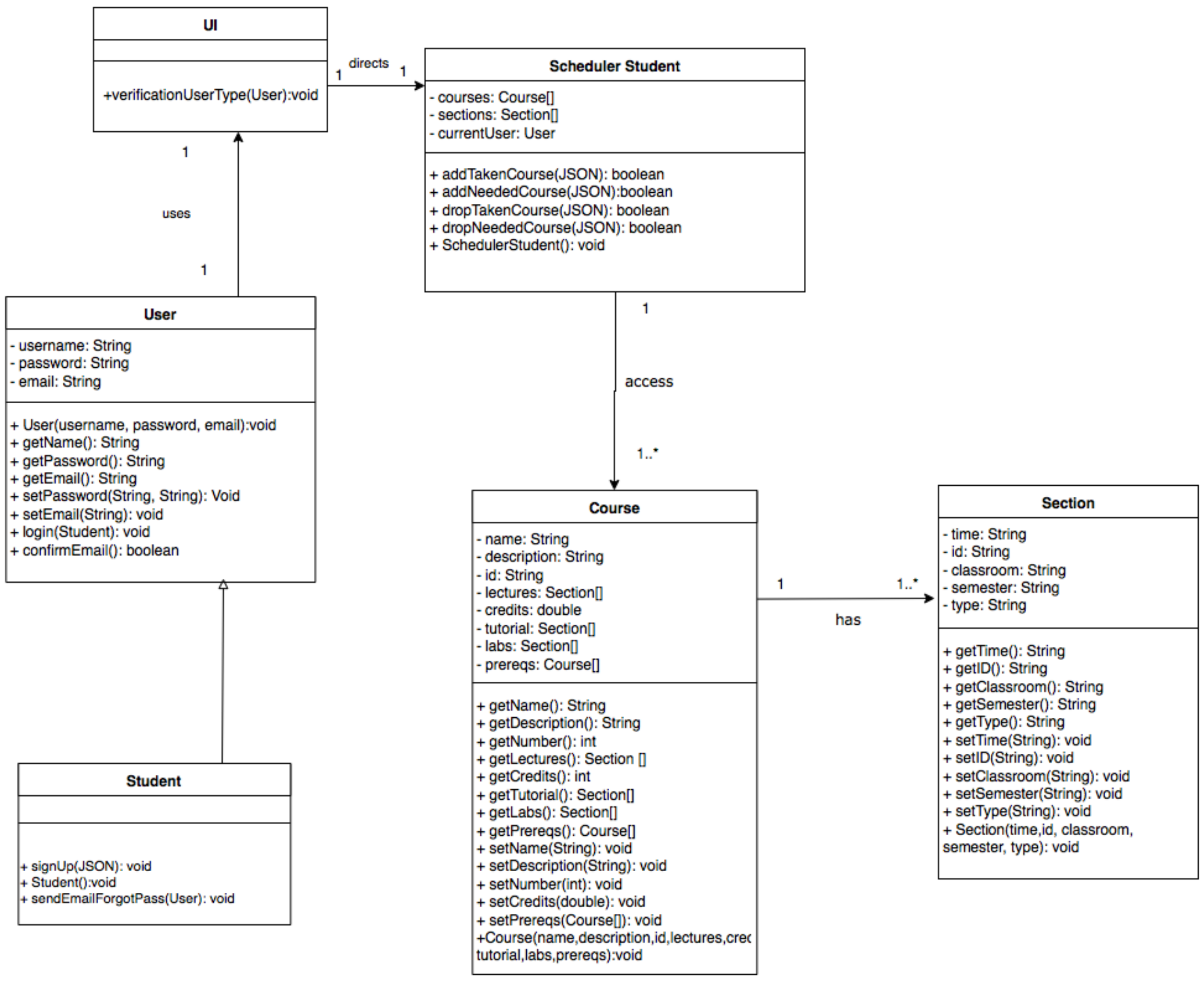
4.1.2 Student Manage Courses

This subsystem portrays the path for a student to manage his course load. The scheduler possesses methods such as *addNeededCourse(), dropTakenCourse()*, which allows for the user to simply add courses to his/her schedule or simply drop them. The methods *dropTakenCourse()* and *addTakenCourse()* are functions where the user himself creates his own transcript. The project and specifications defines the courses taken in the previous semesters as inputs to be put into the database.

Student Manage Courses is part of the component Course Management, the methods listed below are elaborated.

Classes Involved	**Scheduler Student**
Method(s) Implemented	addTakenCourse(JSON): boolean **Implemented in Class**: Scheduler Student **Description**: Method that adds a taken course in the scheduler **Input Parameter(s)**: takes JSON **Return Type**: boolean addNeededCourse(JSON): boolean **Implemented in Class**: Scheduler Student **Description**: Method that adds a needed course that has not been taken yet in the scheduler **Input Parameter(s)**: takes JSON **Return Type**: boolean dropTakenCourse(JSON):boolean **Implemented in Class**: Scheduler Student **Description**: Method that remove taken course requested by the student and returns a boolean **Input Parameter(s)**: JSON **Return Type**: boolean dropNeededCourse(JSON):boolean **Implemented in Class**: Scheduler Student **Description**: Method that remove a course that is needed by the student and returns a boolean **Input Parameter(s)**: JSON **Return Type**: boolean

Student Manage Courses



4.1.3 - Student manages preferences

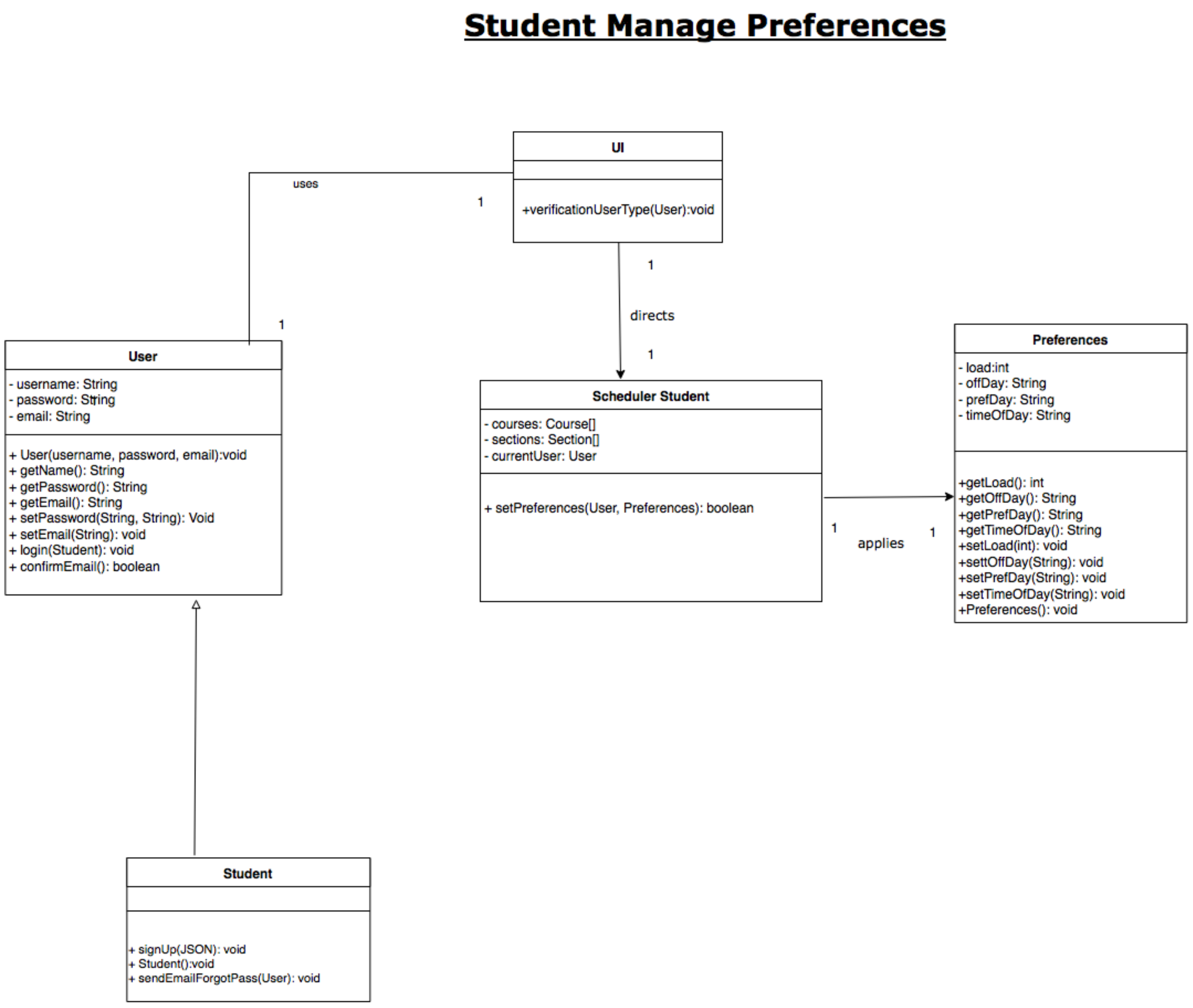
For this subsystem, the scheduler contains one method called *setPreferences()* where the user who is a student can set his preferences. The *Preference* class contains attributes such as *timeOfDay*, *offDay*.

Student manages preferences is part of the component preferences manager and the following methods are elaborated

Classes Involved	**Scheduler Student, Preferences**
Method(s) Implemented	setPreferences(User,Preferences):boolean **Implemented in Class**: Scheduler Student

	Description: Method that returns a boolean for preferences modification **Input Parameter(s)**: User and Preferences **Return Type**: boolean getPreferences(User):Preferences **Implemented in Class**: Scheduler Student **Description**: Method that returns preferences of a user **Input Parameter(s)**: User **Return Type**: Preferences setLoad(int):void **Implemented in Class**: Preferences **Description**: Method that sets the load for preferences **Input Parameter(s)**: int **Return Type**: void setOffDay(String):void **Implemented in Class**: Preferences **Description**: Method that sets the days that the students wants to be off **Input Parameter(s)**: String **Return Type**: void setPrefDay(String):void **Implemented in Class**: Preferences **Description**: Method that sets the preferred day that the student wants **Input Parameter(s)**: String **Return Type**: void setTimeOfDay(String):void **Implemented in Class**: Preferences **Description**: Method that sets the preferred time that the student wants the course to start **Input Parameter(s)**: String **Return Type**: void Preferences(): void **Implemented in Class**: Preferences

	Description: Default constructor that initializes preferences **Input Parameter(s)**: Empty **Return Type**: void

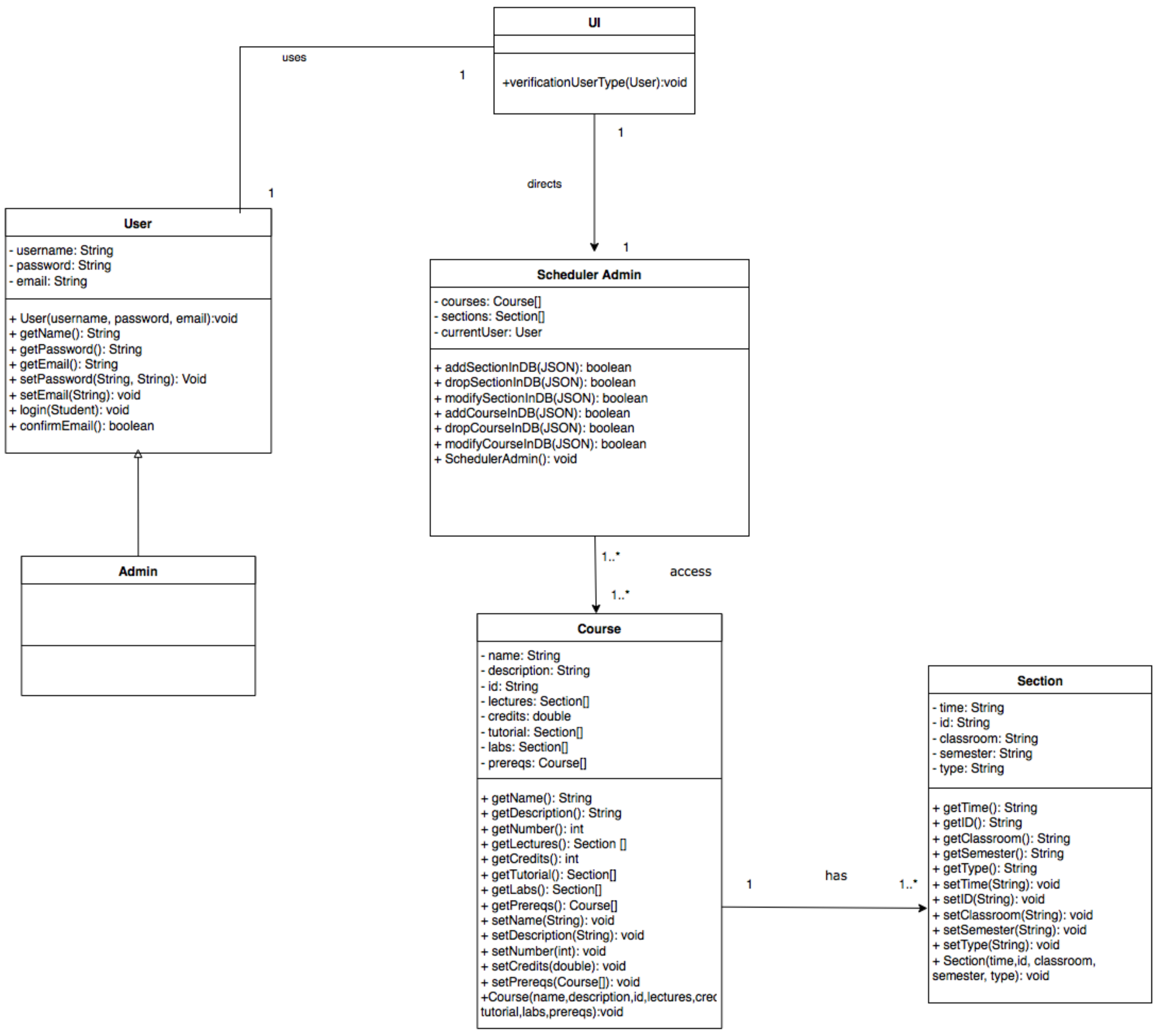


<u>4.1.4 - Admin Manage Course Database</u>

This subsystem portrays the relationship between the priviliges of the admin and the database. The admin will set all the courses and sections that students can take through the *Scheduler Admin* class. This system contains methods that manipulate the database such as *addSectionInDB(), modifyCourseInDB().*

Classes Involved	Scheduler Administrator
Method(s) Implemented	addSectionInDB(JSON): boolean **Implemented in Class**: Scheduler Admin **Description**: Method that adds a section in the database **Input Parameter(s)**: takes JSON **Return Type**: boolean dropSectionInDB(JSON): boolean **Implemented in Class**: Scheduler Admin **Description**: Method that removes a section from the database **Input Parameter(s)**: takes JSON **Return Type**: boolean modifySectionInDB(JSON):boolean **Implemented in Class**: Scheduler Admin **Description**: Method that modifies section for a course inside database **Input Parameter(s)**: JSON **Return Type**: boolean addCourseInDB(JSON):boolean **Implemented in Class**: Scheduler Admin **Description**: Method that adds course inside database **Input Parameter(s)**: JSON **Return Type**: boolean dropCourseInDB(JSON):boolean **Implemented in Class**: Scheduler Admin **Description**: Method that removes course from database **Input Parameter(s)**: JSON **Return Type**: boolean modifyCourseInDB(JSON):boolean **Implemented in Class**: Scheduler Admin **Description**: Method that modifies the course inside database **Input Parameter(s)**: JSON **Return Type**: boolean

Admin manage course database



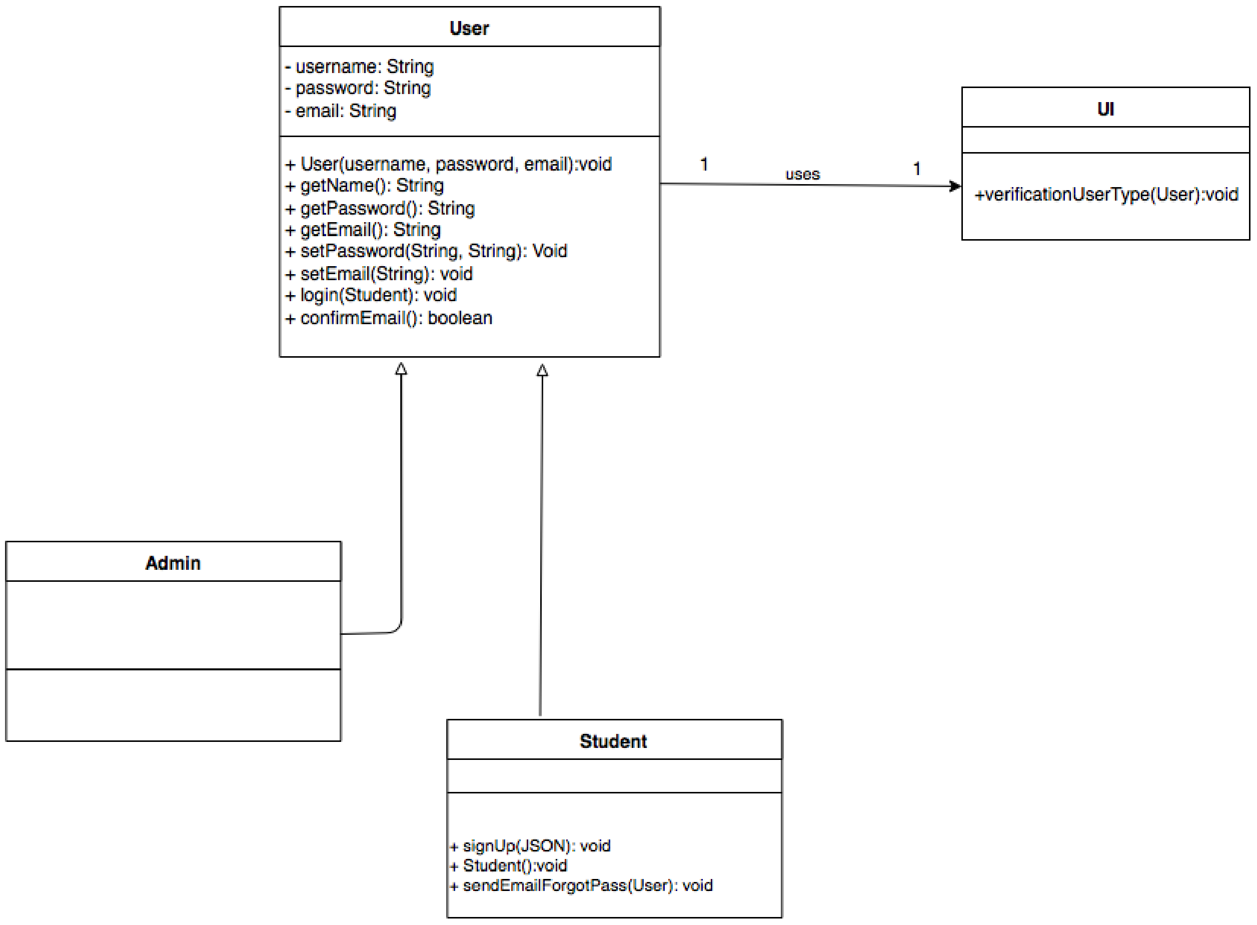
4.1.5 - Admin and Student Verification

This subsystem has one functionality which is to verify which type of user is accessing the system(student/admin). The *UI* class represents the bridge between the systems and will direct the user to their corresponding Scheduler system depending on their type(student/admin).

Classes Involved	**Student, User, UI**
Method(s) Implemented	login(Student): void

	Implemented in Class: User **Description**: Method that allows user to login. For security purposes, hashing will be used in order to avoid potential breaches and unwanted access. **Input Parameter(s)**: takes User **Return Type**: void signUp(JSON): void **Implemented in Class**: Student **Description**: Method that signs up the student **Input Parameter(s)**: takes JSON **Return Type**: void verificationUserType(User): void **Implemented in Class**: UI **Description**: Method that checks the user type and redirects to their respective scheduler according to their type **Input Parameter(s)**: takes User **Return Type**: void

Student and Admin Verification



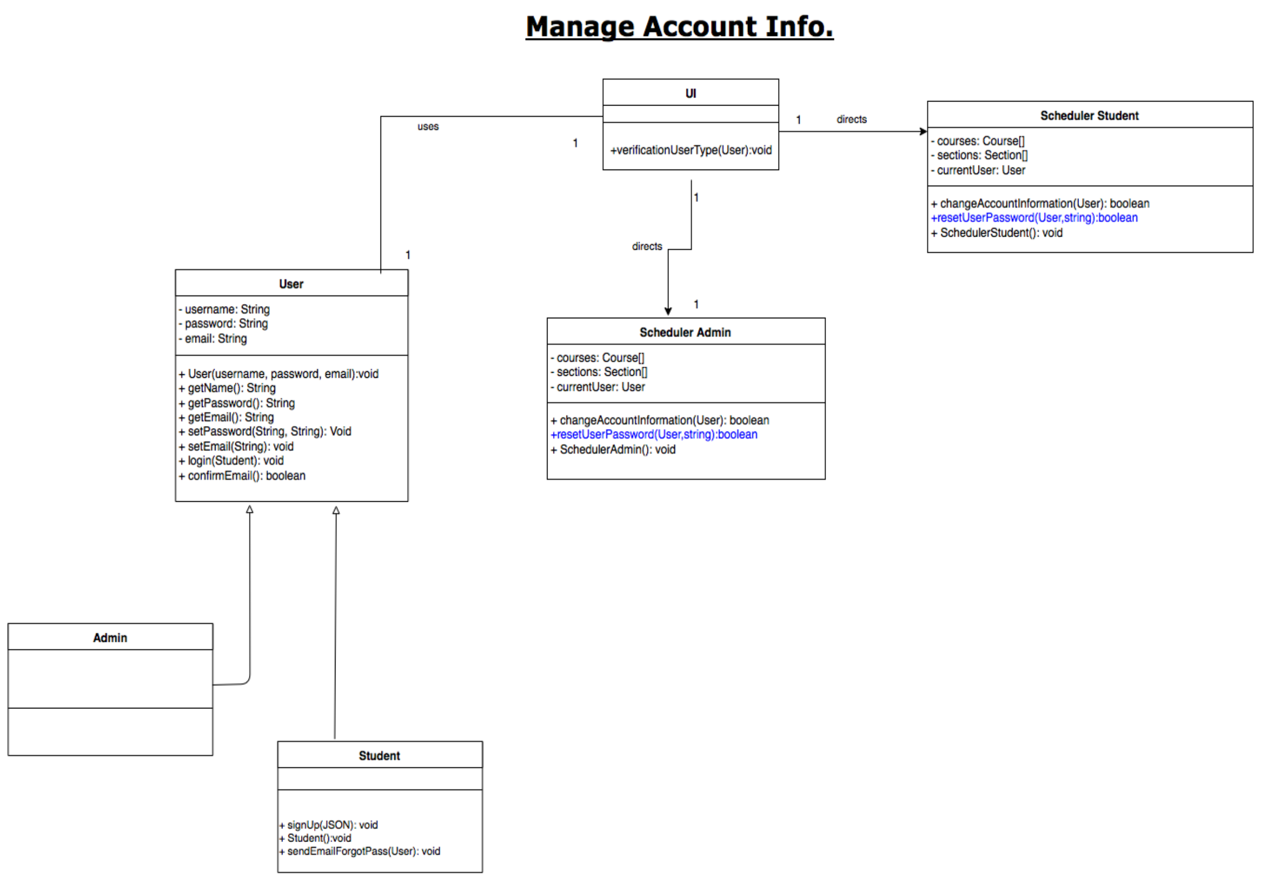
4.1.6 - Admin and Student Managing Account Information

In this subsystem, both Scheduler systems(admin/student) contain methods that allow for the user to change its account information: *changeAccountInformation(), resetUserPassword()*. This subsystem simply permits the user to change his/her information and characteristics.

Admin and Student Managing Account Information is part of the component Account Management.

Classes Involved	**Student, User, Admin, Scheduler Student, Scheduler Admin**
Method(s) Implemented	setPassword(String, String): void

	Implemented in Class: User **Description**: Method that allows user to change their password **Input Parameter(s)**: takes two Strings, first string is the old password and second string the new password **Return Type**: void setEmail(String): void **Implemented in Class**: User **Description**: Method that allows to change email address for the user **Input Parameter(s)**: takes String **Return Type**: void sendEmailForgotPass(**User**): void **Implemented in Class**: Student **Description**: Method that sends the password to the Student **Input Parameter(s)**: takes User **Return Type**: void changeAccountInformation(User): void **Implemented in Class**: Scheduler Admin **Description**: Method that changes the account information of a user **Input Parameter(s)**: takes User **Return Type**: void changeAccountInformation(User): void **Implemented in Class**: Scheduler Student **Description**: Method that changes the account information of a user **Input Parameter(s)**: takes User **Return Type**: void



4.2 Unit Descriptions

Class Name	**Admin**
Description	A type of user inherited from the User class representing the Admin.
Attribute(s)	N/A
Operation(s)	Ø Admin(): void o Creates an Admin object which allows Admin to interact with the rest of the system. Ø sendEmailForgotPass(UserObj : User): void o Sends an email to Admin explaining how to access system if they forgot password. Ø AlterDB(): void o Admin alters information inside the database such as Course and Section information.

Class Name	**Student**
Description	A type of user inherited from the User class representing the Student.

Attribute(s)	● neededCourses : Course[] ● preferences : Preferences ● takenCourses : Courses
Operation(s)	Ø signUp(JSON_File : JSON): Boolean o Registers the student into the system for the first time. Ø Student(): void o Creates a preset Student object. Ø sendEmailForgotPasswordt(UserObj : User): void o Sends an email to the student explaining how to log back on if they forgot their password.

Class Name	**User**
Description	The generic model of the user of the system from which Student and Admin inherent from.
Attribute(s)	● username : String ● password: String ● email : String
Operation(s)	Ø User(username : String, password : String, email : String): void o Generates a new account for a new user. Ø getName(): String o Returns the user's username. Ø getPassword(): String o Returns the user's password. Ø getEmail(): String o Returns the user's email. Ø setPassword(oldpassword : String, newpassword : String): void o Allows the user to the change their password Ø setEmail(newEmail : String): void o Changes the user's email.

	Ø login(StudentObj: Student): void o Provides the student user access to the system. Ø ConfirmEmail(): Boolean o Sends an automated confirmation email to the user confirming that a new account has been successfully.

Class Name	**UI**
Description	Provides the front end user interface (GUI) to the user.
Attribute(s)	N/A
Operation(s)	Ø VerificationUserType(UserObj : User): void o Determines whether a user is a Student or Admin and only gives them access to parts of the system that they are allowed to interact with. Ø changePage(): void o Changes the page displayed to User based on User's interaction with the system. Ø displayIcons(): void o Displays different icons depending on page being displayed to User such as buttons and checkboxes.

Class Name	**Preferences**
Description	Manages and stores the preferences the Student User selects if any. Used by Scheduler Student.
Attribute(s)	● load : int ● offDay : String ● prefDay : String ● timeofDay : String
Operation(s)	Ø getLoad(): int o Returns the number of courses a student wishes to take per semester. Ø getOffDay(): String o Returns the selection of days the Student selected to try and have off in a semester. Ø getPrefDay(): String

	o Returns the preferred day(s) that the Student wants to have classes on. Ø getTimeOfDay(): String o Displays the time of the day the Student wishes to not have classes at. Ø setLoad(NumOfCourses : int): void o Student sets the number of courses they wish to take per semester. Ø setOffDay(DayOff : String): void o Student sets the days they wish to have no classes.. Ø setPrefDay(PrefDay : String): void o Student sets the days they wish to have classes on. Ø setTimeOfDay(TimeOfDay : String): void o Sets the time of day Student would like to have classes at. Ø Preferences(): void o Generates the preferences of the Student User.

Class Name	**Course**
Description	The object model of an academic course inside the system from which Section is derived. Used by Scheduler Student to construct a Schedule for the Student User.
Attribute(s)	● name : String ● id : String ● description : String ● lectures : Section [ ] ● credits : double ● tutorial : Section [ ] ● labs : Section [ ] ● prereqs : Course [ ]
Operation(s)	Ø getName(): String o Retrieves and displays the name of Course for Admin.

	Ø getDescription(): String o Retrieves and displays the description of Course for the Admin User. Ø getLectures(): Section [ ] o Retrieves and displays the Section(s) of Course that contain lectures. Ø getCredits(): double o Retrieves the number of credits for the Course. Ø getTutorial(): Section [ ] o Retrieves and displays the Section(s) of Course that contain lectures. Ø getLab(): Section [ ] o Retrieves and displays the Section(s) of Course that contain labs. Ø getPrereqs(): Course [ ] o Retrieves and displays the name of Course for Admin.

Class Name	**Section**
Description	Contains the information of a particular section for a given Course which can be accessed and manipulated by an Admin User.
Attribute(s)	● time : String ● id : String ● classroom : String ● semester : String ● type : String
Operation(s)	Ø getTime(): String o Admin retrieves and displays the time the course is offered during a week. Ø getId(): String o Returns the ID of the Course to the Admin. Ø getClassroom(): String o Retrieves and Displays the Classroom number to the Admin.

	Ø getSemester(): String o Returns and displays the semester the Section is being offered to the Admin. Ø getTypye(): String o Returns and displays the type of the Section (Lab, Tutorial or Lecture). Ø setTime(time : String): void o Admin can set the time of a section during the week. Ø setID(ID : String): void o Admin sets the ID of the Section. Ø setClassroom(classroom : String): void o Admin sets the classroom number of the Section. Ø setSemester(semester : String): void o Admin sets the semester the Section is offered in. Ø setType(type : String): void o Admin sets the type of Section (lab, tutorial, lecture) for a particular Course. Ø Section(time : String, id: String, classroom : String, semester : String, type : String): void o Generates a new Section for a Course for the Admin.

Class Name	**Scheduler Admin**
Description	The object model that allows only an Admin User to modify all aspects of the database.
Attribute(s)	• courses : Course [ ] • sections : Section [ ] • currentUser: User
Operation(s)	Ø changeAccountInformation(UserObj : User): boolean o Allows an Admin User to change their account information. Ø resetUserPassword(UserObj : User, oldpass : String, newpass : String): boolean o Checks if User is Admin and allows Admin to change password. Ø addSectionInDB(JSON_File : JSON): boolean o Adds a new Section object to the database. Ø dropSectionInDB(JSON_File : JSON): boolean

	o Removes a Section object from the database. Ø modifySectionInDB(JSON_File : JSON): boolean o Admin modifies the information of a Section object inside the database. Ø addCourseInDB(JSON_File : JSON): boolean o Adds a new Course object to the database. Ø dropCourseInDB(JSON_File : JSON): boolean o Removes a Course object from the database. Ø modifyCourseInDB(JSON_File : JSON): boolean o Admin modifies the information of a Course object inside the database. Ø SchedulerAdmin(): void o Generates a new Scheduler Admin object which allows an Admin User to modify the database.

Class Name	**Scheduler Student**
Description	The object model that creates a schedule for the Student based on information taken from the Preferences, Course, Section and Student classes.
Attribute(s)	• courses : Course [ ] • sections : Section [ ] • currentUser: User
Operation(s)	Ø addTakenCourse(JSON_File : JSON): boolean o Updates database JSON file with record of Courses taken by Student. Ø addNeededCourse(JSON_File :JSON): boolean o Updates database JSON file with record of Courses that Student User needs to take. Ø dropTakenCourse(JSON_File : JSON): boolean o Removes a Course that was taken by Student from database JSON file. Ø dropNeededCourse(JSON_File : JSON): boolean o Removes a Course that Student needs to take from JSON file. Ø getTakenCourses(UserObj : User): Course [ ] o Returns an array of Course that the Student has taken. Ø getNeededCourses(UserObj : User): Course [ ] o Returns and displays an array of Course that Student needs to take. Ø changeAccountInformation(UserObj : User): boolean o Student User may change certain information on their account. Ø resetUserPassword(UserObj : User, oldpass : String, newpass : String): boolean o Student changes their password and is shown whether action was successful or not. Ø generateSchedule(StudentObj : Student): void o Generates and displays a new schedule to the Student based on information from Preferences, database, Course and Section. Ø autoGenerateTakenClasses(numj : int): String [ ] o Generates and displays to Student classes that have already been taken. Ø getCoursesFromDB(): Course [ ] o Retrieves all Courses from the database. Ø getSectionsFromDB(CourseObj : Course): Section [ ] o Retrieves all Course Section(s) from the database.

	Ø getPreferences(UserObj : User): Preferences o Retrieves all of Student's selected preferences. Ø setPreferences(UserObj : User): boolean o Student can set their preferences and are told whether action was successful or not. Ø SchedulerStudent(): void o Default constructor.

5. Dynamic Design Scenarios

5.1 Generate Schedule

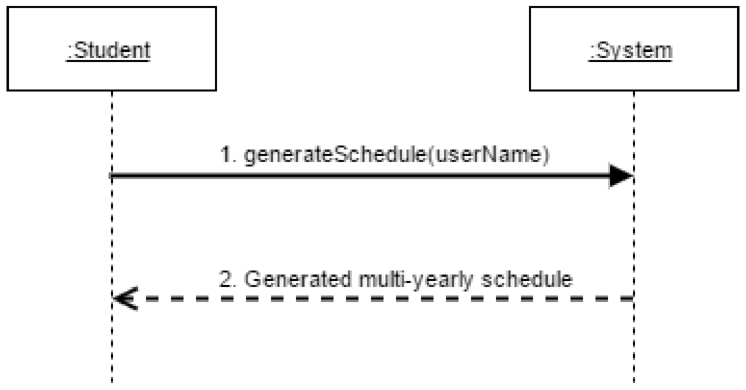
Generate schedule, as the name says, consists of creating a schedule for a student actor by using his/her preferences, list of taken classes and list of needed classes.

5.1.1 Full Use Case

Name:	Generate Schedule	Author	Ying-Chen Chu
Identifier:	UC13	Version:	2.0
Date Created:	2015-02-03	Last Modified:	2015-03-22
Importance:	5/5		
Actor(s):	Student		
Goal:	Generate a schedule		
Summary:	Based on the course selection list and or without the list of preferences, the system generates schedule(s) for the selected courses.		
Related use-cases:	UC "**View Schedule**"		
Preconditions	-User has been authenticated		
Trigger:	User activates the "Generate Schedule" process		
Basic Flow:	1. System generates multi-yearly schedules by fetching and using the preferences, needed courses list, taken courses list. 2. Schedules are generated are displayed to the user.		
Post-Conditions:	Schedule for the course selected (and or without the preferences) is generated.		
Minimum Guarantee:	Previous state of the system remains the same.		
Risk Assessment:	High		
Notes	A schedule could always be generated even though the preferences and needed/taken courses where not set by the student. A schedule based on the regular course sequence would display starting from the current semester,		

	assuming the student is a first year.

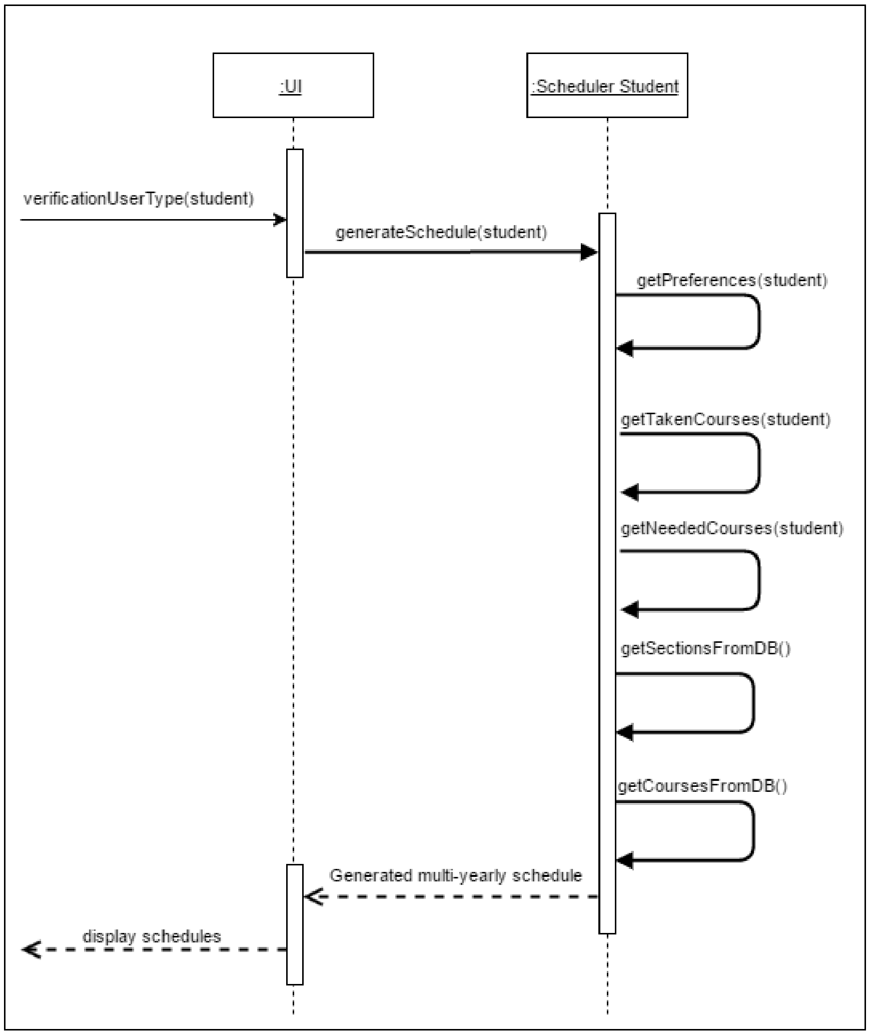
5.1.2 System Sequence Diagram


:Student
:System
1. generateSchedule(userName)
2. Generated multi-yearly schedule

5.1.2.1 generateSchedule*(userName)* Operating Contracts & Respective Sequence Diagram

Name	Contract 1.1 Generate Schedule
Operation	generateSchedule(Student:astudent)
Cross Reference	UC13 Generate Schedule
Pre conditions	-User is logged into the system
Post conditions	-If there is any restriction (preferences or course lists), the schedule is displayed according to those restrictions. -If there are no restrictions, the schedule is displayed according to the general course sequence, starting from the current semester, and assuming this is the students first semester.

Sequence Diagram for contract 1.1



5.2 Reset Password

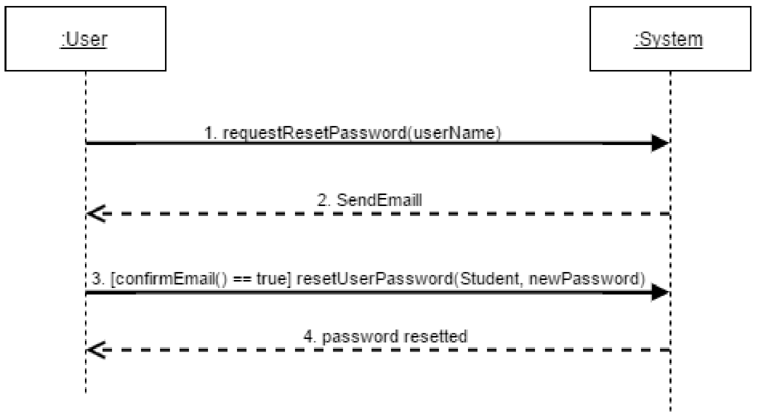
Reset password consists of changing a forgotten password to a new one, meaning the user cannot login into the system. An email is sent to the user. Once confirmed, the user can proceed and change the password.

5.2.1 Full Use Case

Name:	Reset Password	Author	Salma Aly
Identifier:	UC3	Version:	2.0
Date Created:	Jan 30, 2016	Last Modified:	2015-03-22
Importance:	5/5		
Actor(s):	Student, Admin		

Goal:	To reset password
Summary:	The user requests password reset and the system sends them an email that should be confirmed. The user then enters a new password.
Related use-cases:	UC "**Confirm Email**"
Preconditions	1-The user has already signed up to the system 2-The user has provided a valid email address
Trigger:	The user initiates a password reset
Basic Flow:	1-System receives a request to reset the password to a specific user 2-System sends an email to user 3-If email confirmed, the user enters a new password 4-System displays a password reset successfully message
Post-Conditions:	-Previous password has been replaced by the new password -User can now login with the new password.
Minimum Guarantee:	The user can view the login page with reset password option
Risk Assessment:	Low

5.2.2 System Sequence Diagram

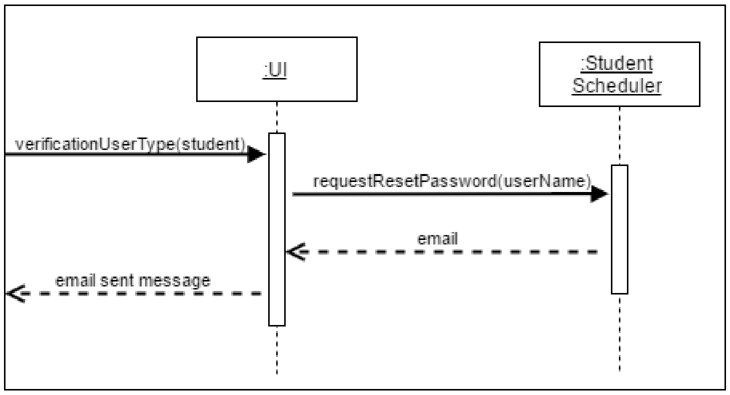

:User
:System
1. requestResetPassword(userName)
2. SendEmaill
3. [confirmEmail() == true] resetUserPassword(Student, newPassword)
4. password resetted

5.2.2.1 *requestResetPassword(userName)* Operating Contracts

Name	Contract 2.1 Request Password Reset
Operation	requestNewPassword(String: userName)

Cross Reference	UC3 Reset password
Pre conditions	-User has an existing account -User has a valid email linked to their account
Post conditions	-Email is sent to the user to be confirmed

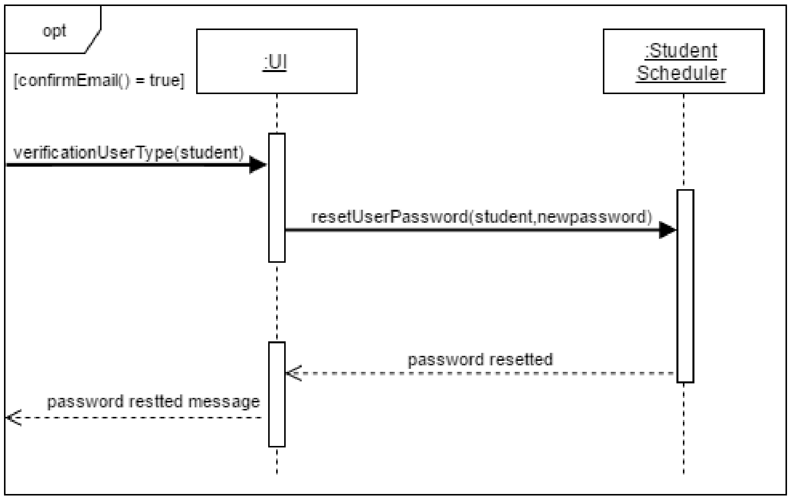
Sequence diagram for contract 2.1



5.2.2.2 *[confirmEmail() == true] resetUserPassword(Student, newPassword*) Operating Contracts

Name	Contract 2.2 Reset Password
Operation	resetUserPassword(User:auser,String:newPassword)
Cross Reference	UC3 Reset password
Pre conditions	-User has an existing account -User has a valid email linked to their account -User has confirmed the email sent
Post conditions	-Old password has been replaced by the new password -User can login into the system using the new password

Sequence diagram for contract 2.2



5.3 Set Preferences

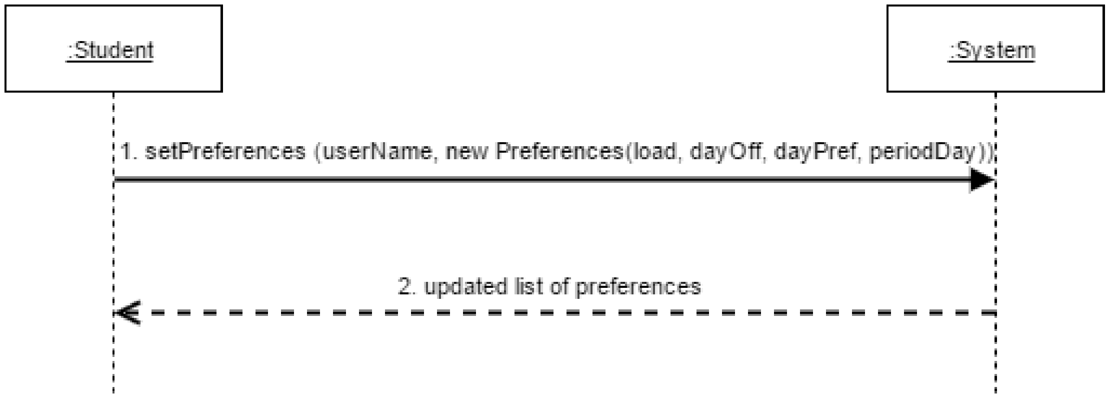
Set preferences consists of saving the students preferences into the database, so they can be used later while generating the schedules.

5.3.1 Full Use Case

Name:	Set Preferences	Author	Ying-Chen Chu
Identifier:	UC11	Version:	2.0
Date Created:	2015-02-03	Last Modified:	2015-03-22
Importance:	5/5		
Actor(s):	Student		
Goal:	Input the schedule preferences to the system		
Summary:	Set the schedule preferences and save them to the system.		
Related use-cases:	-		
Preconditions	User has been authenticated		
Trigger:	Student sets his preferences and initiates the "set preferences" process		
Basic Flow:	1. System replaces the old preferences by the new ones. 2. System displays the list of the new preferences.		
Post-Conditions:	-The user’s old preferences are replaced by the new preferences. -Preferences are associated with the current student.		
Minimum Guarantee:	Previous state of the system remains unchanged.		

Risk Assessment:	Low
Notes	Preferences are not mandatory for the schedule. Therefore, the "old preferences" could also be the default preferences, which is basically no preferences (all elements set to null).

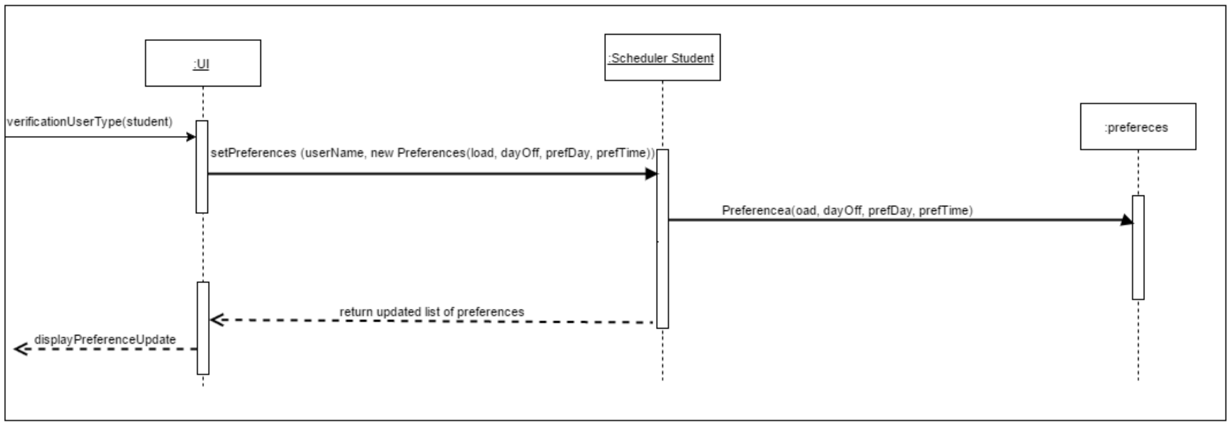
5.3.2 System Sequence Diagram


:Student
:System
1. setPreferences (userName, new Preferences(load, dayOff, dayPref, periodDay))
2. updated list of preferences

5.3.2.1 *setPreferences (userName, new Preferences(load, dayOff, dayPref, periodDay))* Operating Contracts

Name	Contract 3.1 Set Prefrences
Operation	setPreferences(Student: student, Preferences: pref)
Cross Reference	UC11 Set Preferences
Pre conditions	Student logged into the System
Post Conditions	-Student's preferences are saved into the database and associated to this specific user. -Old preferences are replaced by the new preferences.

Sequence Diagram for Contract 3.1


:UI
:Scheduler Student
:prefereces
verificationUserType(student)
setPreferences (userName, new Preferences(load, dayOff, prefDay, prefTime))
Preferencea(oad, dayOff, prefDay, prefTime)
return updated list of preferences
displayPreferenceUpdate

5.4 Add Course to Program

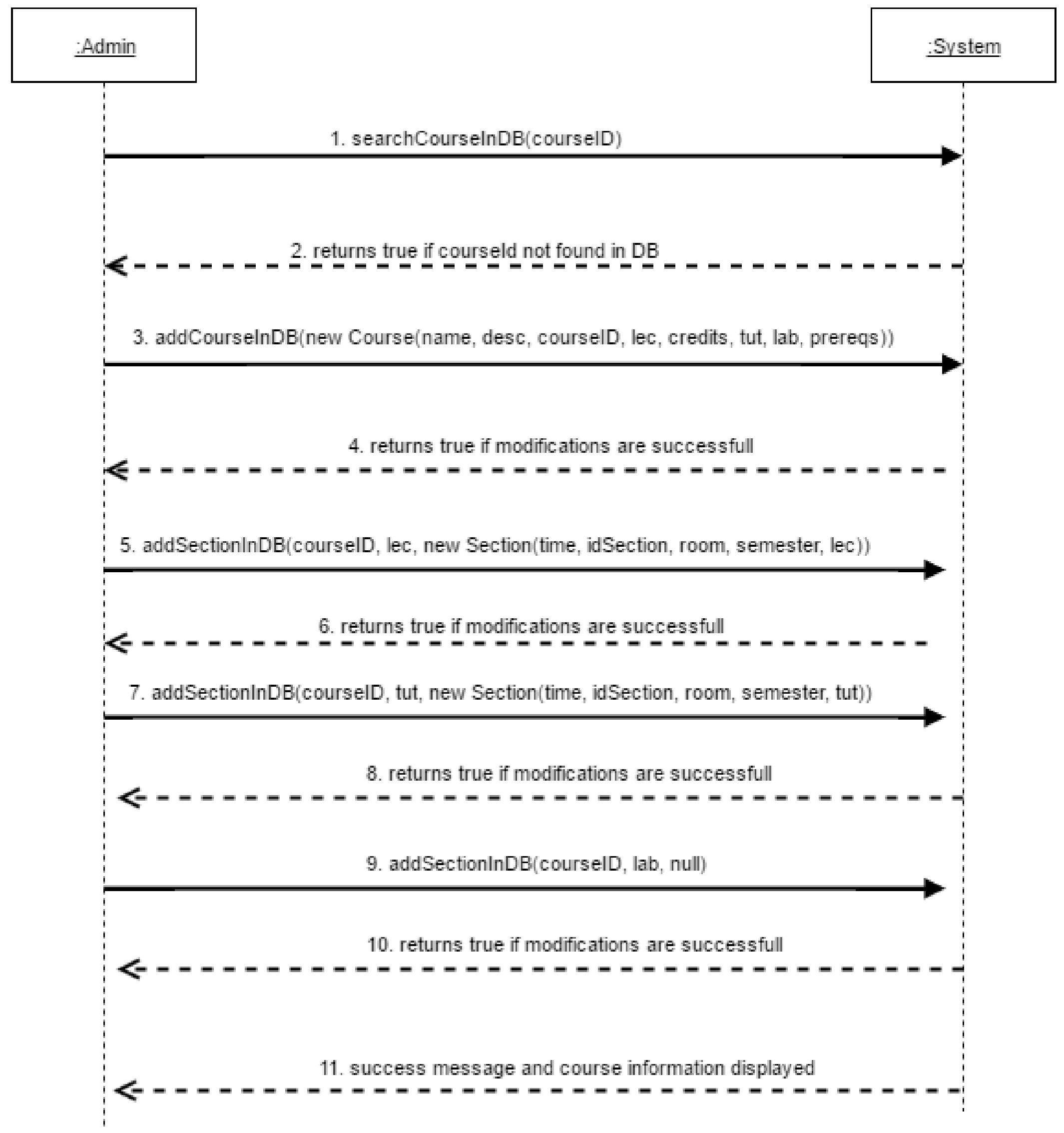
Add course to program consists of adding a course to the entire database of courses by the admin. By adding a course, the admin has to add the corresponding sections for lab, tutorial and lecture to this course.

5.4.1 Full Use Case

Name:	Add Course to Program	Author	Adil Hssaini
Identifier:	UC23	Version:	2.0
Date Created:	Feb 2, 2016	Last Modified:	2015-03-22
Importance:	5/5		
Actor(s):	Admin		
Goal:	To add a new course to a specific program.		
Summary:	The Administrator updates the list of required courses for a specific program by adding a new course.		
Related use-cases:	-		
Preconditions	1. Actor is logged on as administrator. 2. The course is not part of the program sequence stored by the system.		
Trigger:	Administrator activates the “Add Course to Program” process.		
Basic Flow:	1.System searches for a specific course id into the database. 2. If the course is not found, system returns a success message. 3. System takes the inputted specifications of the course and adds it to the database. 4.System returns a success message and the results of adding the course 5. System takes in the inputted data for the lecture section and adds to the current course database.		

	6.System displays success message and result of adding new course. 7. System takes in the inputted data for the tutorial section and adds to the current course database. 8.System displays success message and result of adding new course. 9. System takes in the inputted data for the laboratory section and adds to the current course database. 10.System displays success message and result of adding new course. 11. Success message and the entire course information and its sections are displayed.
Post-Conditions:	Course now exists in the database.
Minimum Guarantee:	List of courses in the program stored by the system will not be affected.
Risk Assessment:	Low
Note:	If a course does not have a lab for example, the admin will be able to specify this while filling in the information. The non-existant type of section, in this case, lab, will simply be saved as null.

5.4.2 System Sequence Diagram

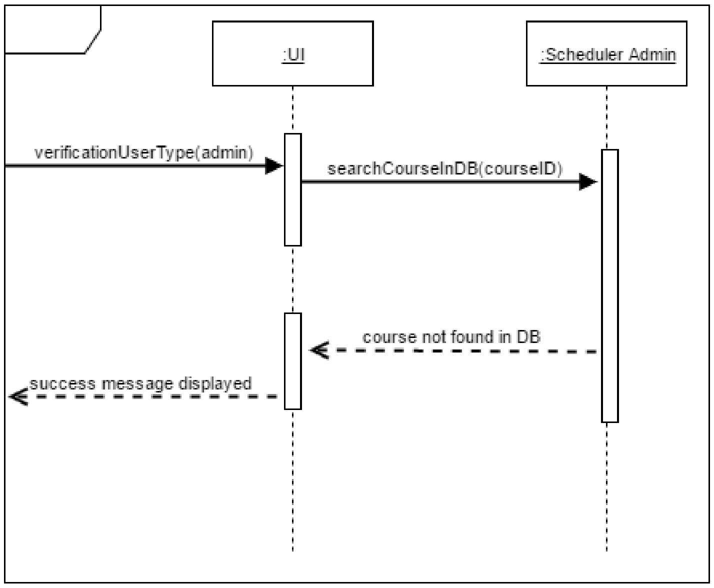


:Admin
:System
1. searchCourseInDB(courseID)
2. returns true if courseId not found in DB
3. addCourseInDB(new Course(name, desc, courseID, lec, credits, tut, lab, prereqs))
4. returns true if modifications are successfull
5. addSectionInDB(courseID, lec, new Section(time, idSection, room, semester, lec))
6. returns true if modifications are successfull
7. addSectionInDB(courseID, tut, new Section(time, idSection, room, semester, tut))
8. returns true if modifications are successfull
9. addSectionInDB(courseID, lab, null)
10. returns true if modifications are successfull
11. success message and course information displayed

5.4.2.1 *searchCourseInDB()* Operation Contracts

Name	Contract 4.1 Search course in database
Operation	searchCourseInDB()
Cross Reference	UC23 Add Course to Program
Preconditions	-User Logged in as administrator
Postconditions	-Course is successfully not found in database.

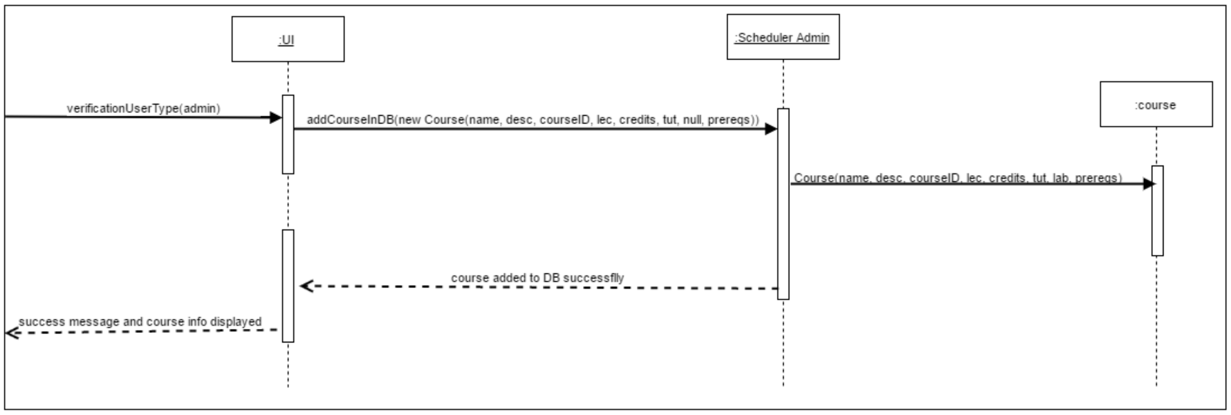
Sequence Diagram for Contract 4.1


:UI
:Scheduler Admin
verificationUserType(admin)
searchCourseInDB(courseID)
course not found in DB
success message displayed

5.4.2.2 *addCourseInDB(new Course(name, desc, courseID, lec, credits, tut, lab, prereqs))* Operation Contracts

Name	Contract 4.2 Add course in database
Operation	modifyCourseInDB(JSON)
Cross Reference	UC23 Add Course to Program
PreConditions	-User Logged in as administrator -Course does not already exist
Post Conditions	-Course can searched and found in database.

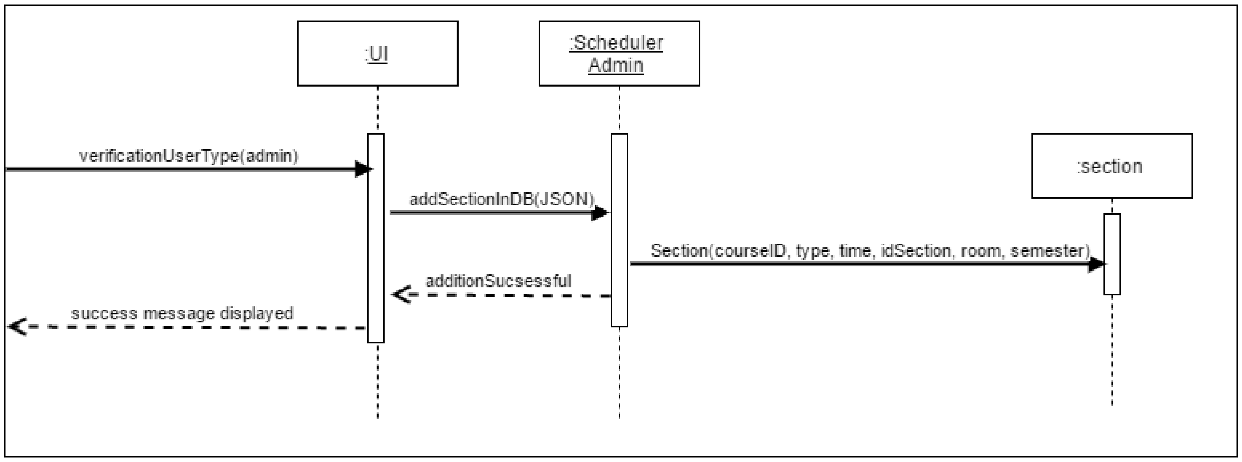
Sequence Diagram for Contract 4.2


:UI
:Scheduler Admin
:course
verificationUserType(admin)
addCourseInDB(new Course(name, desc, courseID, lec, credits, tut, null, prereqs))
Course(name, desc, courseID, lec, credits, tut, lab, prereqs)
course added to DB successflly
success message and course info displayed

5.4.2.3 *addSectionInDB(JSON)* Operation Contracts

Name	Contract 4.3 Add section in database
Operation	addSectionInDB(JSON)
Cross Reference	UC23 Add Course to Program
PreConditions	-User Logged in as administrator -There is an existing course to which the section will be added.
Post Conditions	-New section can be found inside of the current course database.

Sequence Diagram for Contract 4.3


:UI
:Scheduler Admin
:section
verificationUserType(admin)
addSectionInDB(JSON)
Section(courseID, type, time, idSection, room, semester)
additionSucsessful
success message displayed

Note: The last three arrows in the system sequence diagram for this use case, are calling the same method, but adds different types of sections (Lecture, Tutorial, Lab). Therefore, contract 4.3 and its sequence diagram represent these 3 last arrows.

6. Estimation

Few corrections needed to made to the of deliverable 2, mainly due to the unexpected amount of work involved with the different designs.

Previously Incurred Costs:

	Deliverable 0	Deliverable 1
Total Hours	5	70
Cost Estimate ($25/hr)	$125	$1750

The cost estimate for deliverable 2 revealed itself to be slightly higher than previously forecasted as detailed in the table below.

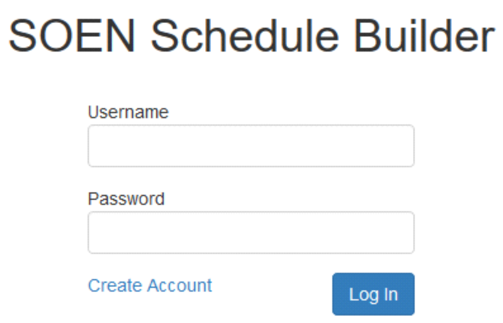
Artifact		Estimated Cost in Hours	Final Cost in Hours
4+1 Architectural View	Logical	10	10
	Development		3
	Physical		3
	Process		5
	Scenarios		1
Subsystem Interface Specification		25	20
UML Class Diagram		12	10
Dynamic Design Scenario		6	10
Estimation		7	5
Rapid Prototyping Report		22	25
Testing		15	15
Risks		4	4
Total Hours		101	111
Cost Estimate ($25/hr)		$2525	$2775

The cost of the following tasks is not expected to be higher than what was estimated in the first deliverable. The total amount of hours calculated for the entire project was 352 hours, amounting to the estimated cost of $8800. Due to the 10 hours increase incurred during the realization of the second deliverable, the new total number of hours rises to 362hrs, which corresponds to a total project estimate of $9050.

7. Rapid Prototyping and Risk

7.1 Front-End Work

All the major pages were completed for the prototype, except for the schedule page, which will be completed after the scheduling algorithm is in place.



SOEN Schedule Builder

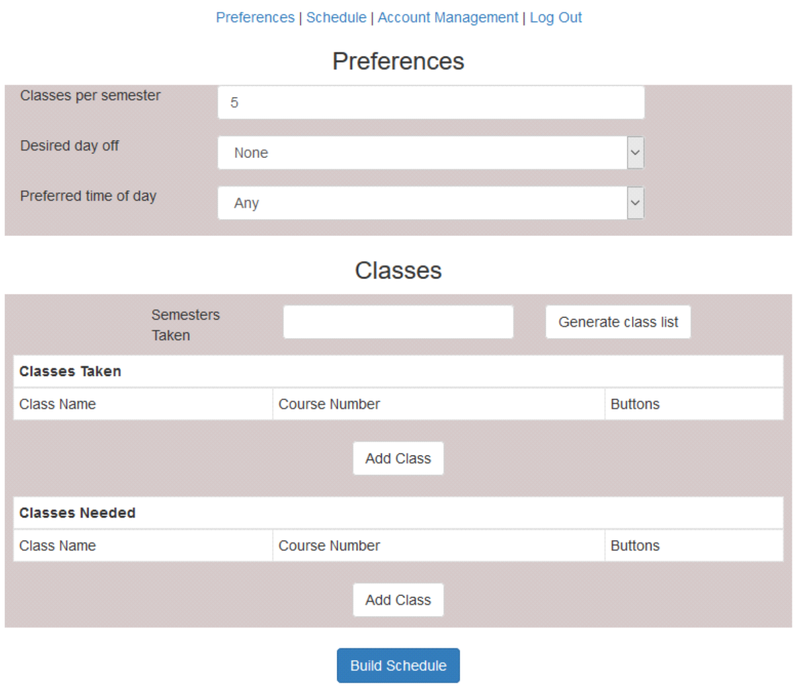
Username

Password

Create Account

Log In

Figure 7: The login page



Preferences | Schedule | Account Management | Log Out

Preferences

Classes per semester	5
Desired day off	None
Preferred time of day	Any

Classes

Semesters Taken

Generate class list

Classes Taken		
Class Name	Course Number	Buttons

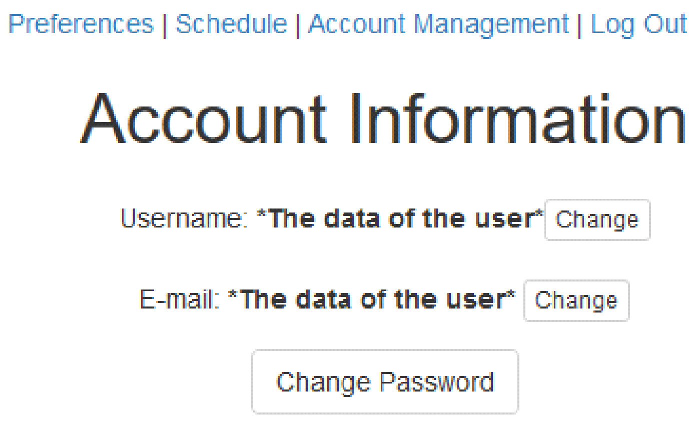
Add Class

Classes Needed		
Class Name	Course Number	Buttons

Add Class

Build Schedule

Figure 8: The preferences/classes page



Account Information

Username: ***The data of the user***

E-mail: ***The data of the user***

Figure 9: The account page

The front-end work progressed as expected without any scope changes. React continues to be a simple and elegant solution for the front-end that only gets complicated when it comes to communicating with the back-end.

7.2 Back-End Work

For the prototype, the plan was to get a working interface that interacted with the database. As planned, one of the first steps was to design and set up the database that was to be used. A document explaining how to create and populate the Mysql database was also shared with the team. The second step was to work with the front-end team to link the set of pages with the databases through server calls. The register and the log in modules were working. However, they were not using the Laravel framework. Incorporating the current pages in the MVC model and the Laravel folder architecture as well as using the features of the framework (such as routing) were the last step of the prototype implementation. As a confirmation, the application was shared with the team and implemented on their end to test the modules that were implemented.

7.2.1 Front-end and back-end communication issues

There were a lot of issues with the usage of React and Laravel. React was initially designed for use with NodeJS, where the Javascript can be rendered on the server, but this is not an option when our back-end is running with a PHP framework. Since we are using React generate the views (this is operation is done client-side) and Laravel to handle the architecture, this makes it complicated for Laravel to manage the views for instance. While the original prototype did prove React renders the pages flawlessly, we had not yet implemented them with the framework and its architecture. Our initial prototype was merely testing log-in authorization and registration. Once Laravel came into play, this complicated the project. These issues did not affect our design decisions.

7.3 Added Technology

7.3.1 Twitter Typeahead

We have implemented Twitter's typeahead component to improve the user experience. It is a jQuery open source text component that provides auto-completion suggestions as the user types (much like the google search bar). We will be using it to help the user add their needed and taken courses. The list of courses given in the auto-complete will be retrieved from the server. This will not only make a better user experience, but make it less likely for the user to input a course that doesn't exist by accident.

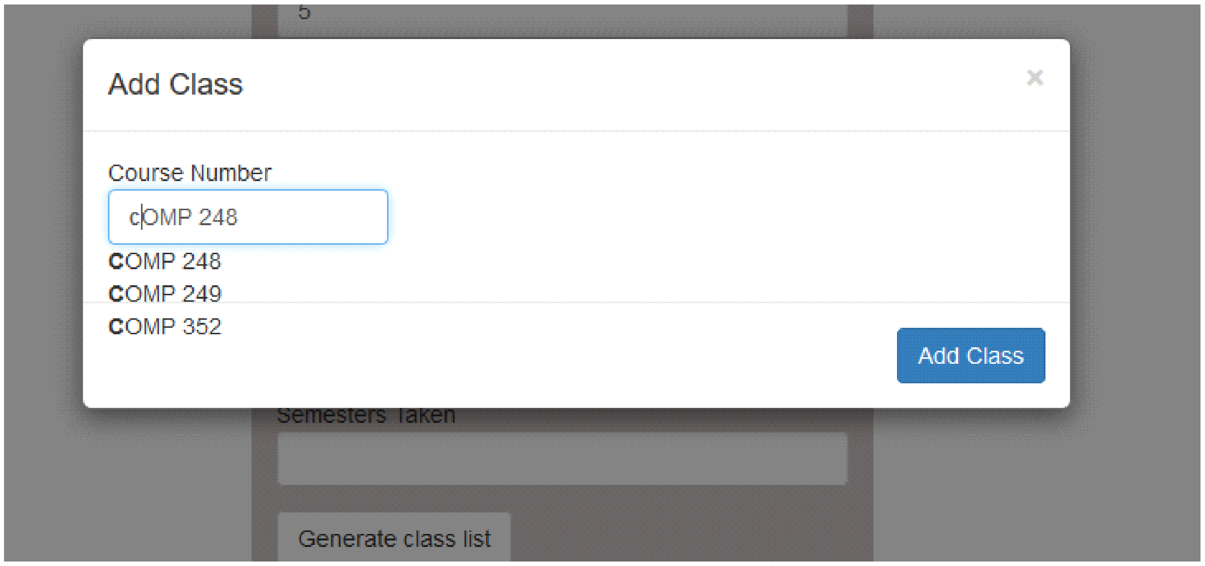


Figure 10: Twitter Typehead