Four-Year Plan Tracker
Software Design Specification

River Veek (riverv) – 3-9-2021 – v2.7

Table of Contents

1. SDS Revision History 2
2. System Overview 2
3. Software Architecture 3
4. Software Modules 4
   4.1. Student/Advisor Interface 4
   4.2. Student Data 5
   4.3. Student Records 9
   4.4. Flask Middleware 11
5. Dynamic Models of Operational Scenarios (Use Cases) 12
6. Acknowledgements 13

1. SDS Revision History

This lists every modification to the document. Entries are ordered chronologically.

Date	Author	Description
2-12-2021	ezekielp	Initial document creation
2-13-2021	ezekielp	Added part 2 and sequence diagram for part 3
2-16-2021	ezekielp	Added part 4 (minus diagrams)
2-17-2021	ezekielp	Added diagrams and captions for parts 4 and 5
2-18-2021	ezekielp	Added section 3
3-7-2021	ezekielp	Added 4.4, updated 4.3, 4.1, 2, 3
3-7-2021	nkruss	Split section 4.2 into two submodules, added a diagram
3-7-2021	jkashuba	Edited typos/grammar, added to 4.2.1 alternative design
3-7-2021	ezekielp	Full document grammar check, added 4.4 diagram
3-8-2021	jkashuba	Added to 4.2.2 alternative design, Proofreading
3-9-2021	jkashuba	Added to 4.1.5 alternative design
3-9-2021	riverv	Edited grammar, fixed small formatting inconsistencies

2. System Overview

The Four-Year Plan Tracker is a system that allows users to construct and visualize a long-term plan for scheduling college courses, specifically for CIS students at the University of Oregon. The system permits data to persist in a “database” while also allowing it to deviate and be recalculated as needed depending on class completion.

The system is divided into four primary components. The first consists of the web interface (referred to below as the Student/Advisor Interface module), consisting of fields to enter recently completed courses and the term of completion, along with a generated matrix of suggestions for future course slates to complete the degree. It also contains an interface for choosing which user information to load and the option for adding a new user. The second component (referred below as the Student Data module) consists of the logic for determining which future courses to suggest and when. It contains all course, student, and major objects and relates them based on user input from the web interface component. It also generates the final plan based on course completion. The third primary component (referred to below as the Student Records module) consists of the logic for saving information in a “database” of pickle objects such that user information may be saved and later restored as students make progress in their curriculum. The final primary component (referred to below as the Flask Middleware module) consists of a framework for connecting the behavior of the other three components.

The Student Data module is further divided into Student/Degree/Course (SDC) Objects and Degree Planning submodules. The first of these submodules handles the actual Python class definitions, while the second takes the structure of these classes to calculate four-year plans.

3. Software Architecture

The software architecture (as briefly described in Section 2) consists of the following four components:

1. The web interface (Student/Advisor Interface) – Handles all user input and visible output.
2. The internal logic (Student Data) – Makes calculations for the plan and keeps track of entity relationships for a student object.
3. The pickle-based backend (Student Records) – Stores student objects and helps populate the student select screen.
4. The Flask controller (Flask Middleware) – Directs traffic and makes calls to functions provided by the other modules

The overall relationship between these components is described in Figure 3.1 below.

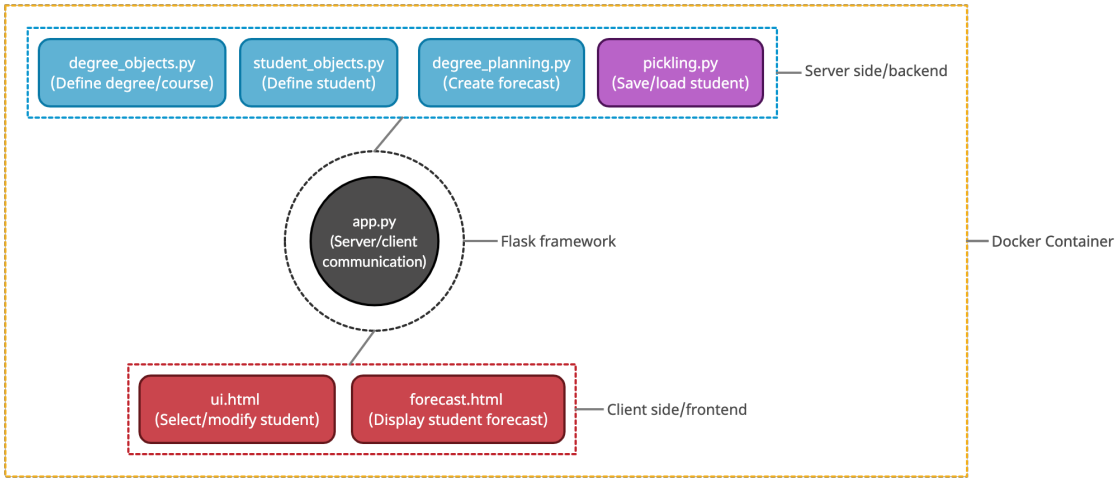

Figure 3.1 *Static architecture diagram. Shows the relationships between the four color-coded modules and their overall place within the Docker container.*

Generally speaking, the web interface is the first line of defense for avoiding erroneous user input. It avoids fill-in-the-blank style fields when possible and favors dropdown menus populated by both the pickle "database" and the degree class logic such that only valid data can be passed into the system. There are some potential error cases that the frontend logic will have to solve. For example, the case when a course is marked for removal from the user's table of staged changes and it does not exist, or when the user attempts to add a course with incomplete information. However, the Flask Middleware component, along with the largely fault-tolerant user interface, ensures that the final object stored as a pickle is in a consistent state.

As Figure 3.1 shows, the Flask framework largely serves as a "middle-man" for the other two modules. This allows for a separation of concerns of the UI, the data storage, and the plan computations (discussed in detail in Section 4). Not only does this allow for easy division of

labor during development, but it also helps to guide the decisions for creating interfaces between the modules.

4. Software Modules

4.1. Student/Advisor Interface

4.1.1. Role and primary function.

The role of the Student/Advisor Interface module is the primary interface between the two user classes and the planning functionality of our system. It accepts user input to pass into the Flask Middleware module and displays the outputted four-year plan. As the interface between the user and the system, its goal is to restrict choices (where possible) to minimize errors, while still giving the user the full range of the system capability.

4.1.2. Interface Specification

There are two primary interfaces for this module. One for the interface to the user, and one for the interface to the Flask Middleware module.

The interface to the user consists primarily of dropdown menus and simple buttons for collecting user information. This includes collecting information about a new student's ID, course names, and the term and year in which the courses were taken. It also involves showing the user the final four-year plan display.

The interface to the Flask Middleware module handles the tangential interface to the Student Records and Student Data modules by passing back the user information to load a student record, create a new one, modify those objects, and calculate a four-year plan. This interface also includes the Flask Middleware module's redirects to the appropriate templates (what the user sees).

4.1.3. A Static Model

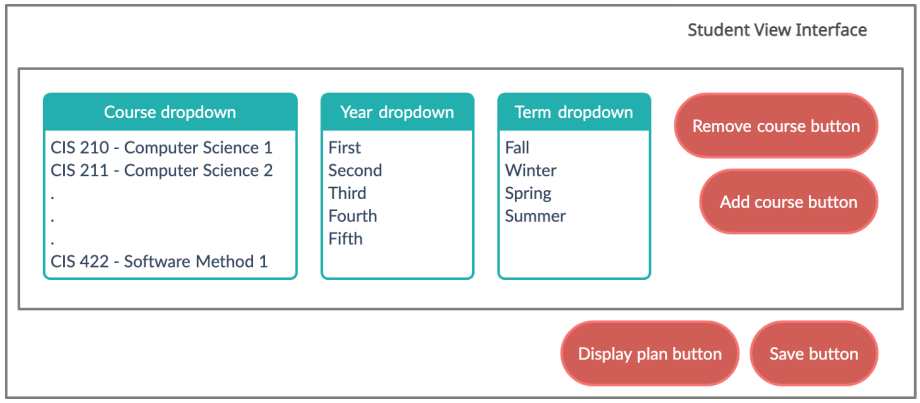

Figure 4.1 *Graphic representation of the web elements that the user will primarily interact with.*

4.1.4. Design rationale

The rationale behind factoring out the interface of the system is largely to identify the range of user inputs so that there is a clear idea of the information that gets passed to the rest of the system. It also allows for a separation of the programming languages used in our system, as this module mainly deals with HTML and JavaScript, while the other pieces are written in Python.

Additionally, it allows for the factoring out of *how* information will be displayed from *what* information will be displayed. More broadly, it allows for a concentration of all things that the user sees and how a user would interact with said visuals.

4.1.5. Alternative designs

Initially, the primary landing page for our system was going to prompt for a first and last name to then query the database. However, due to the relatively low cost of querying the pickle file names and the improved fault tolerance of a dropdown interface, we scrapped that idea, so the user can only see and choose objects that actually exist (or create a new one).

We also discussed the option of having "Student" and "Advisor" buttons with a pretend login step to gatekeep the pickle files that a user has access to. This would be with the intention of differentiating the needs of the different user classes. However, as discussed below, due to the local storage nature of pickle files, we deemed this unnecessary.

Additionally, we originally planned to include a tree view of the prerequisites of the major. However, since students are much more used to the matrix format, we decided to also scrap that idea and focus more on the matrix view, perhaps revisiting it if we had time. We thought it would be easier and more user friendly to instead display all classes in the current major in a list on the main student page, color coded by completion.

Lastly, we would like to implement color coding into the system for courses already completed vs courses that still need to be completed as a visual aid for the user. Due to the complexity of Bootstrap and our given time constraints, this functionality is not set up in the current system. Given more time, we would have passed a flag of some kind with each course so that the HTML could apply the right Bootstrap class to each particular cell and have logic in Jinja or JavaScript to make that call based on the flag in order to color code.

4.2. Student Data

4.2.1 Student/Degree/Course (SDC) Objects

4.2.1.1. Role and primary function

The SDC Objects module consists of the major class definitions that make up the system. It is responsible for parsing the data retrieved from the Student Records module and passing the appropriate information to the Student/Advisor Interface module (and vice-versa). Overall, this module is responsible for storing and organizing user data in a logical manner.

4.2.1.2 Interface Specification

There are two primary interfaces for the SDC Objects module, both via connections made in the Flask Middleware module.

The first is the interface to the Student/Advisor Interface module. It consists of the ability to pass plan information of a particular format to the visualization methods and the ability to retrieve user input (either course information, student information, or degree information to be used to modify the main class objects).

The second is the interface to the Degree Planning submodule. It consists of passing the data stored in the Student Object to the module where a plan can be generated for the student based off of the passed data.

4.2.1.3. A Static Model

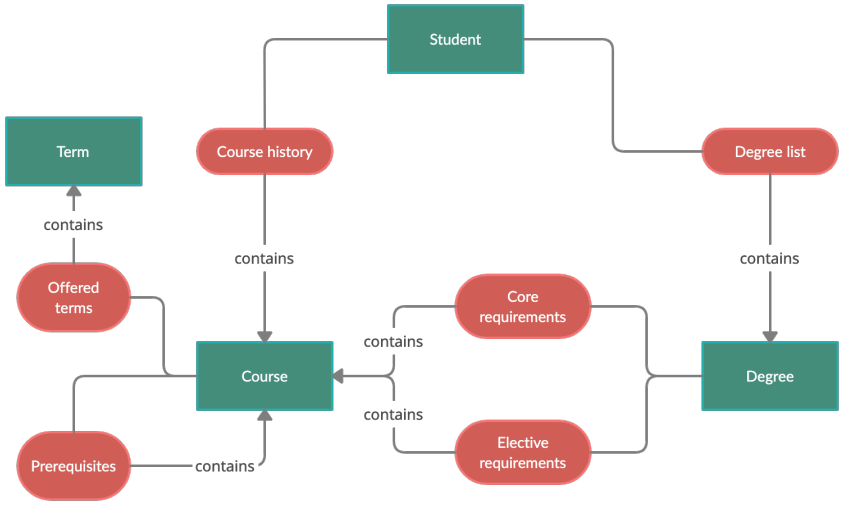

Figure 4.2.1 *Diagram showing the relationships between the Degree, Course, Student, and Term classes.*

4.2.1.4 Design rationale

The Student Data module is designed primarily to factor out the structure of the “unseen” elements of the system. The Student Records can be seen as pickle files and the Student/Advisor Interface can be directly accessed to input and show user information, however, this module is not apparent to the user at all. It acts as a both a bridge and a staging ground for the meat of the software system. It holds data in a form that can be manipulated easily with OO programming idioms but can be stored or displayed when

asked. In a sense, it is split into its own component to more easily understand the system as a whole and to improve computational efficiency.

4.2.1.5 Alternate design

Early in the design process, we identified the need to factor out the actual decision making of our software system to its own module. We thought perhaps that we would have had to use JSON objects in order to achieve this functionality, but due to our change in design for the data storage of our system, we were permitted to use proper OO techniques and Python classes (as further discussed below).

The actual information stored in each Python class has also changed somewhat from our initial designs, as we identified more information that would be needed to give a holistic final output. For example, our original data structures did not consider year or term of completion for each class. However, we quickly realized that in order to give an accurate matrix representation of the past, we would need to store that information.

Originally, we had the intention of handling unique cases such as whether the user has an adequate math placement test score. This would affect whether the user is required to take MATH 101 & MATH 112. An eventual alternative design we mapped out to handle this case (as well as others) would be at the beginning of the creation of a new student object. The webpage would route to an interim page with a series of questions such as "do you have an adequate math placement test score?", "do you intend on taking summer classes?", and "in how many years do you want to finish your degree?". After checking these boxes and answering these conditional questions, the system would then route the user to a page corresponding to their unique identifying data. Unfortunately, due to time constraints and the way we had to structure the webpage (discussed in more detail in Section 4.4.5), we were unable to route the user to different pages in this manner. Ultimately, given more time, we would have re-built the front end to handle these unique circumstances for each student or advisor user.

4.2.2 Degree Planning

4.2.2.1. Role and primary function

The Degree Planning Logic module is responsible for parsing course data to assemble a possible plan for a user that meets all the degree requirements. This module leaves the visualization aspect of that information to the Student/Advisor Interface module.

4.2.2.2 Interface Specification

This module receives student data in the form of a Student Object from the SDC Objects module and uses the provided data to construct a reasonable degree plan for the inputted student. The degree plan generated by this module is then returned to the SDC Objects.

4.2.2.3. A Dynamic Model

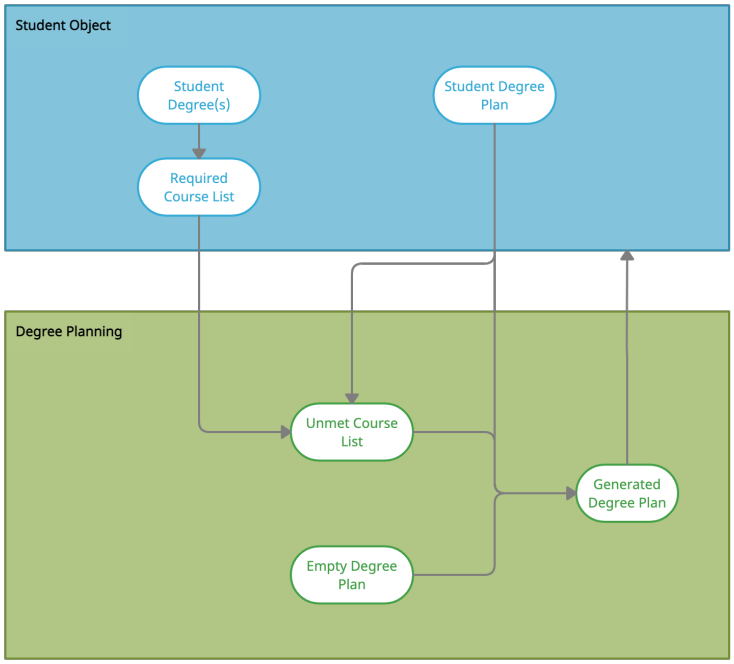

Figure 4.2.2 *Diagram showing the relationship between the Student Object from the SDC Objects module and the Degree planning module along with base logic.*

4.2.2.4 Design rationale

This module is the core of our system thus it needed to be created. We chose to make it its own module though to ensure isolation of components.

4.2.2.5 Alternate design

One design decision we had to make when creating this module was determining how we wanted to handle electives for a degree. One example of this question involved how we wanted our system to handle upper division CIS elective. According to University of Oregon CIS major requirements, students pursuing a CIS major are required to take 16 credits of upper division electives, 8 credits of which can be satisfied by a 300+ level CIS elective with the rest having to be 400+. To handle these electives in a generated plan, we had two main options. The first option was to pick specific default CIS electives that we would add to the generated plan. The other option was to create a placeholder course object to represent these requirements and then add the placeholder to the plan. We opted to implement the second option with the rationale of not wanting to confuse our users by causing them to think that they had to take some specific courses to meet those elective requirements. With this implementation, if a user specifies that they have taken a CIS course that counts towards the elective requirement, then when a plan is generated for said student one of the placeholder courses will be removed from the generated plan and replaced with the course.

Another alternate design we had for this module involved taking course difficulty into account. Difficulty logic would have adjusted the plan such that several challenging classes would not be placed in the same term. However, this logic was deemed to be a "could have" functionality, thus the main reason it was not implemented was due to time constraints.

One design flaw that we were unable to find a workaround for regards upper division CIS electives. Due to the logic in the back end, the generated plan will "correctly" place 400+ CIS electives earlier in the plan than 300+ CIS electives. The logic that incurs this flaw is that CIS electives above the 400 level are generally assumed to be used for the 400+ electives rather than 300+, so are checked against these first. Given more time, this minor flaw would have been handled as well.

4.3. Student Records

4.3.1. Role and primary function.

The Student Records module has the primary responsibility of saving the state of a student for later reference and updates. The objects saved by this module are then used to actually calculate four-year plans. For advisors, this module can save a record for each different student they are assisting, to be retrieved as needed, likely during advising appointments. In fact, student users may also take advantage of the ability to save multiple plans to explore particularly divergent options/majors.

4.3.2. Interface Specification

The Student Records module mainly interacts with the Flask Middleware module, though it is also tangentially useful to the Student/Advisor Interface and Student Data module as well. In particular, the initial dropdown menu of existing records collects file name information from the pickle objects (if there are any) and presents them as a choice for the user (Student/Advisor Interface). However, the primary interactions involve the saving and loading of Python objects defined by the Student Data module. When saving, the Student Records module is passed a student ID from the Flask Middleware module and an object reflecting the current state of the student. It then saves a pickle file of the student object using the student ID as a file name. When loading, the Student Records module is passed a student ID, whereupon it loads the corresponding pickle and passes it to the Flask Middleware module for later modification.

4.3.3. A Static Model

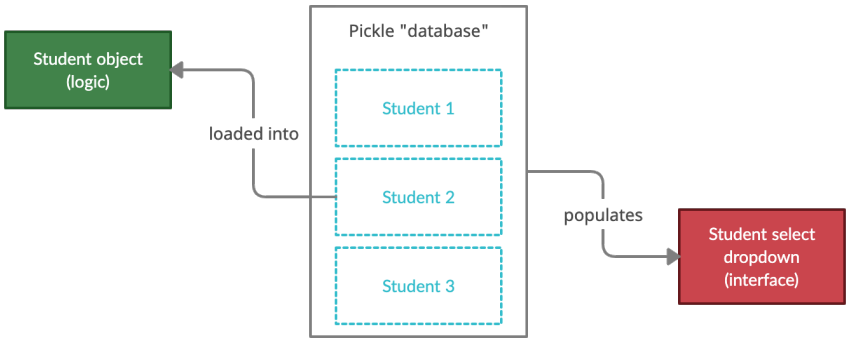

Figure 4.3 *Data storage of student records as it relates to the other primary system components.*

4.3.4. Design rationale

The design rationale for the Student Records module is central to the usability of the system as a whole. If we did not have a method of data persistence, then a couple of clear problems would arise.

First, a student would have to input their course history every single time they wanted a generated plan. For the first year or two, that may not be a major cause for concern, but after a certain point, it becomes difficult and tedious to remember the term, year, and course number of every single class taken. Users would also likely have to refer to another piece of software for that information and make the current system less independent and flexible.

Second, and perhaps more importantly, without data persistence, then the system would not fundamentally solve the problem at hand. One of the issues with the current system for course planning, is the static nature of degree guides. Often, a plan comes down to checking boxes on a paper list of required courses every time a student makes an appointment. Without data persistence, then that situation would effectively be what the student user would need to undergo to use this system.

Therefore, the Student Records module is factored out and designed in a way to improve usability, increase independence from other software, and provide a positive contribution to the current methods of course planning.

4.3.5. Alternative designs

Initially, we planned to use MongoDB to store our data instead of pickles. However, since MongoDB does not properly persist student data after each execution of the Docker container, we needed to explore other options.

By choosing pickles over MongoDB, we gained the ability to use Python classes and their relationships to one another to more clearly represent the information processed in the Student Data module. This is because MongoDB requires JSON formatted information, which only supports the most basic of Python objects. With the full range of

Python's functionality at our disposal, we could more easily save course "dependencies" as other course objects.

Additionally, pickles only provide local storage, which negates the practical necessity of a log in mechanism. On a less visible note, cutting MongoDB also meant that we did not have to establish a docker-compose framework, which adds more chances for implementation errors.

4.4. Flask Middleware

4.4.1. Role and primary function.

The Flask Middleware module is primarily responsible for managing the communication between user input from the Student/Advisor Interface module and the logic from the Student Data module. It passes back input about which objects (if any) from the Student Records module will get loaded, input about courses to be added to class objects, and once the user is ready, passes the calculated four-year plans back to the Student/Advisor Interface.

4.4.2. Interface Specification

As the application framework, the Flask Middleware module has many of the major module interfaces that make up the system. It loads objects from the Student Records module or alternatively creates a new object defined in the Student Data module. It modifies the new or loaded object based on user input from the Student/Advisor Interface module. It saves modified objects back to the Student Records module. Finally, it collects the four-year plan generated by the Student Data module and displays it back to the Student/Advisor Interface module.

4.4.3. A Static Model

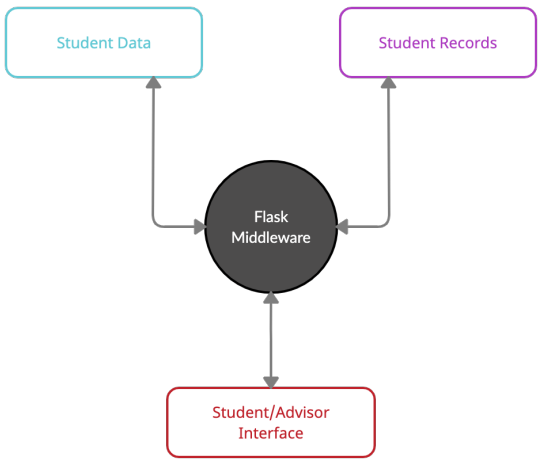

Figure 4.4 *Diagram showing the role of Flask Middleware as the communication hub for the other modules*

4.4.4. Design rationale

The design rationale behind the Flask Middleware module had not only the software architecture in mind, but also the software development phases.

From an architectural standpoint, it allows for proper separation between what the user sees and the actual structure behind the scenes. The user only needs to know the specifications of the interfaces and have a high-level mental model about how the system works to be able to use the software. The module provides the framework for the entire application and uses the well-defined functions provided by the rest of the system to merely make a series of calls at the appropriate times.

From a development standpoint, the Flask Middleware module allowed for separation of concerns among the developers such that we were largely able to function independently. In any system, care must be taken to be clear about inputs and outputs of the various modules such that information flows without errors. However, since this module was tasked with connecting them all (in effect as a crossroads), as long as each module was clear about type signatures and return values, this was a relatively simple task. It permitted all other modules to have the smallest interface possible with each other while the Flask Middleware directed traffic.

4.4.5. Alternative designs

Given this was a web application connecting Python logic with an HTML/JavaScript interface, there was always going to be a need for some middleware to make that connection seamless. Since many of our developers had experience working with Docker and Flask, the design of this module did not deviate much from the original specifications.

That said, the overall structure of the Flask routing needed to be tweaked to accommodate both Bootstrap (as a focused library despite Flask's affinity for <form>) and the complexity of passing information to and from a table. In a perfect world, this module would have loaded a landing page for users to select their user information and then redirected to the object modification page, passing the appropriate fields. However, due to time constraints and the work around provided by rebuilding the same landing page, that reality did not come to pass. In effect, we would have liked to have three templates and a different routing function for each of them instead of the two we ended up with that needed to use if blocks to differentiate the routing.

5. Dynamic Models of Operational Scenarios (Use Cases)

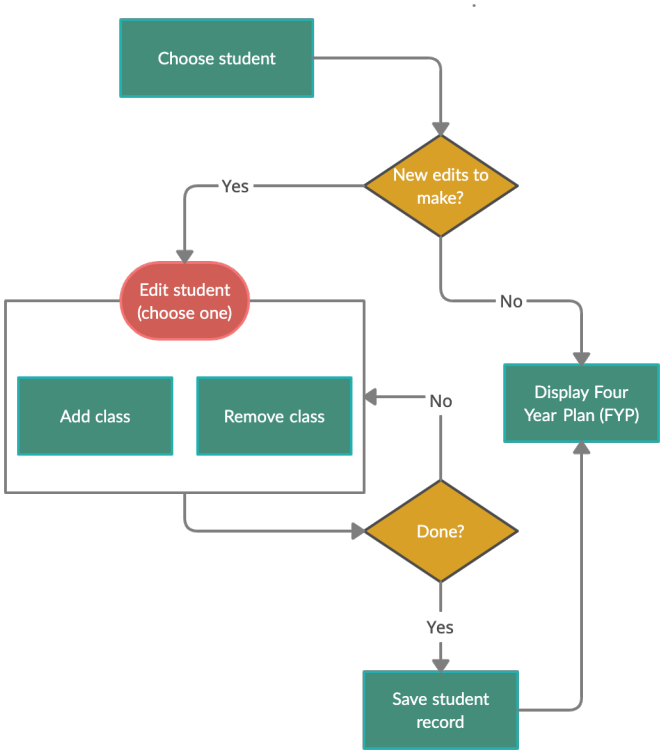

Figure 5.1 *Flow chart describing actions taken by the user while using the software.*

6. Acknowledgements

The UML diagrams in this document were created with the Creately web app.

The SDS template was provided by Juan Flores, who updated it from Anthony Hornof for CIS 422.