EasyA
Software Design Specification

Spike Chen (sc), Joey Le (jl), Andrew Liu (al), Peter Nelson (pn), Krishna Patel (kp) - 2-5-2023 - v2.0

Table of Contents

1. SDS Revision History 2

2. System Overview 2

3. Software Architecture 3

- **3.1 Modules** 3
- **3.2 Interactions** 4
- **3.3 Rationale** 4

4. Software Modules 5

- **4.1 Instructor Data Parser** 5
- **4.2 Web Scraper** 6
- **4.3 Student Graphic User Interface (Student GUI)** 7
- **4.4 Instructor Data Archive and Searcher** 9
- **4.5 Search Results Graph Generator** 11
- **4.6 Administrator Data Updater** 12

5. Dynamic Models of Operational Scenarios (Use Cases) 14

- **5.1 Student** 14
- **5.2 System Administrator** 14

6. References 15

7. Acknowledgements 15

1. SDS Revision History

Date	Author	Description
4-30-2019	ajh	Created the initial document.
1-12-2023	kp	Document created for EasyA project
1-14-2023	jl	Created a sequence diagram for the student and system administrator use case
1-14-2023	al	Writeup for SW arch section + rationale
1-14-2023	kp	Added software modules
1-15-2023	pn	Added Prof. Anthony Hornof into acknowledgements
1-15-2023	jl	Revised use case diagrams
1-28-2023	jl	Updated use case diagrams to be higher-level state diagrams
2-2-2023	jl	Update “Operational Use Cases” to be higher-level state diagrams
2-3-2023	jl	Revised “System Overview” to be at a higher level and added and described diagrams, along with revisions and subheadings for “Software Architecture”.
2-4-2023	jl	Developed more accurate diagrams and descriptions of our modules within “Software Architecture” and added additional references and acknowledgements specified in the SRS.
2-5–2023	jl	Quick revisions to “System Overview” and “Instructor Data Parser”

2. System Overview

The EasyA system serves to store and present data on the University of Oregon (UO)’s instructors and the letter grade distribution for their different classes. The system supports the ability to format raw data (.JSON) and convert it for its own usage. EasyA presents the data in the form of graphs and supports side-by-side viewing of two different graphs; this is for students looking into which instructors or classes have the highest rates of receiving A’s or failing grades (D’s and F’s). The system also has features to modify the existing data with new data to keep the system up to date with information.

We received a System Requirements Specification (SRS) from Professor Anthony Hornof to develop this system as part of his class. As described in the SRS, the system takes inspiration from the “Emerald Grade Tracker” system, which allowed students to view the letter grade distribution of UO instructors and their classes from information gathered from 2013 to 2016. We use the same data provided there and seek to improve on that system by creating our own system. However, the data provided is limited. The system only uses data provided between 2013 and 2016. Not all class information can be found since some of that information was redacted. In other words, if your class doesn't show up here, it means the data was redacted.

There are two ways to interact with the system: students and administrators.

- Students use the graphic user interface (GUI) as a way to view the data in bar graphs.

This component interacts with a data search component and graph generation component.

- Administrators use a separate program designed to update the current data. It will interact with the data search component for that component to use the newly created data.
- For development purposes, there is a web scraper and data parser provided to compile and format raw data for usage.

3. Software Architecture

Below, there is a figure illustrating how our system is designed in six different modules. The different keys are listed and apply to all static diagrams that will be described.

- Circles indicated specific user classes or specific data.
- Rectangle boxes indicate modules.
- Dotted arrows indicate interactions involving data or data files.
- Solid arrows indicate interactions involving input and/or output.

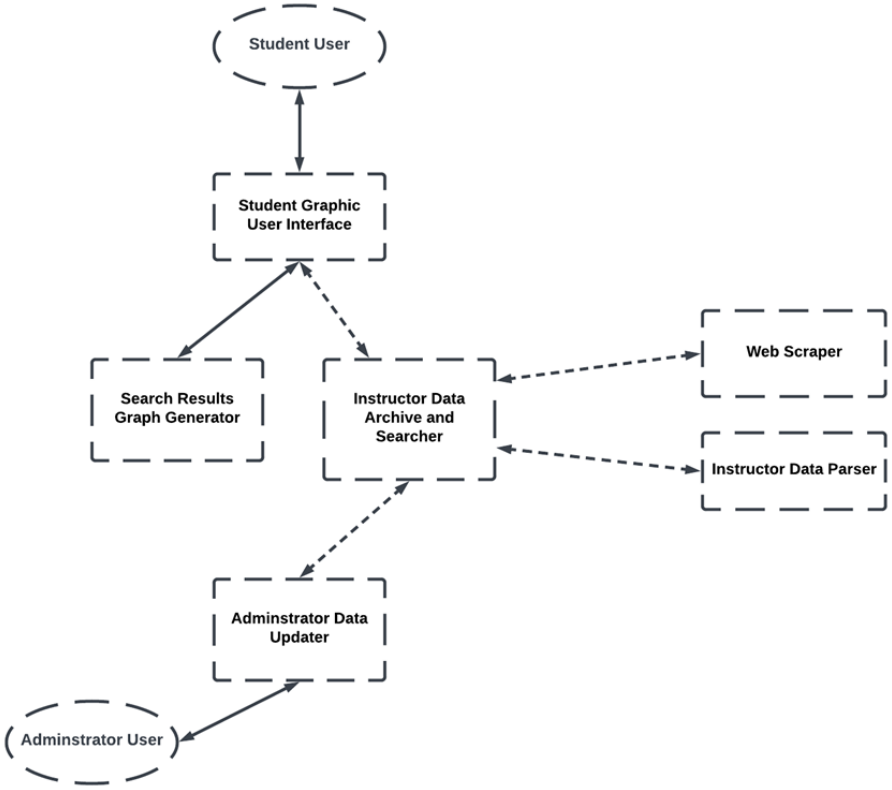

Figure 3.0. The overall architectural design for our EasyA system. It highlights the specific modules and what they interact with and how.

3.1 Modules

1. Instructor Data Parser: Will parse the original, raw data set to build a file (.CSV) for the system to use, and this component is used only once for the entire system.

2. Web Scraper: Accesses web page files (HTML) of faculty web pages to assess specific faculty members and provide a file (.TXT) listing faculty. Just like the Instructor Data Parser, this is used only once.
3. Student Graphic User Interface: Prompts and receives input from the user to determine what data they want to see.
4. Instructor Data Archive and Searcher: Opens and reads through the formatted data file containing the classes and grades and will also open a faculty file (.TXT) to determine whether an instructor is faculty. It will store the data from both files. When requested to search, it will also calculate the relevant data (ex. percentage A's in 100 level math courses by instructor).
5. Search Results Graph Generator: Receives relevant data and generates the graphs needed.
6. Administrator Data Updater: If an administrator wants to add new data, this module will replace the current data with the new data.

3.2 Interactions

The design contains entry points for the two different users: students and administrators. Starting with the students, the system handles the initialization of the GUI and data and what happens on exit. Students interact with the GUI when looking for instructor data to view and the GUI interacts with the Data Archive and Searcher and Search Results Graph Generator on behalf of the student. The administrator users send new data as inputs for the Administrator Data Updater to replace the current data, and the Data Archive and Searcher will know how to make use of the new data.

With the Web Scraper and Instructor Data Parser, we will have a separate TXT file from the HTML data and CSV file from the JSON data. The Instructor Data Archive and Searcher will rely on these data files when using the system.

3.3 Rationale

Our decision to split the design into the listed components is based on the minimum, but a concise degree of functionality would be required to run this program. In addition, the CSV format can easily be parsed, arguably more than the raw JSON data provided. Performance and speed were not a huge concern for this, so using a CSV file to store the data is fairly ideal for our purposes in facilitating multi-parameter queries from the user. Having multiple separate components with their own methods to handle individual system functions makes debugging much easier. This design essentially breaks the program down to its barest essential components which makes it fairly modular. If any type of functionality needs to be added or updated, doing so is relatively easy with the isolated components.

4. Software Modules

4.1 Instructor Data Parser

This module's role and primary function:
The purpose of this parser module is to turn the data provided into a usable format for interaction with the program. This module will take the JSON file and form a CSV file for local data storage.

Model:

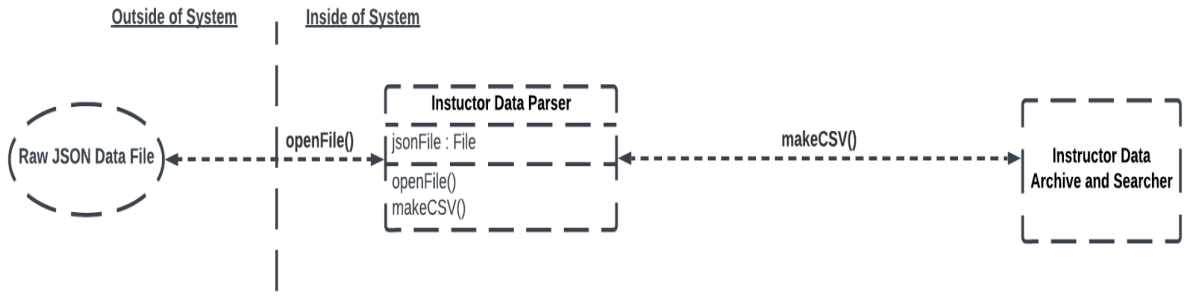

Figure 4.1.1 Static diagram showing how the Instructor Data Parser module would interact with the system.

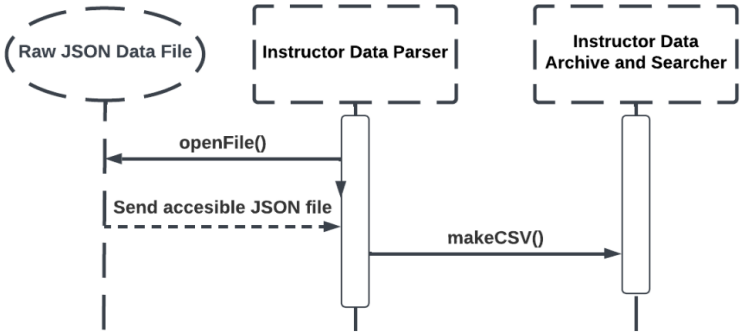

Figure 4.1.2. Dynamic diagram showing how the Instructor Data Parser module would interact with the system. This is done prior to the system's initialization.

Interface specification:
This module will not interact with other parts of the code outside of providing a file that the other parts of the program will rely on. The module must perform three main tasks: read a file, sort data in the file, and then output the file in the sorted format. Within the models, the tasks are as followed:

1. openFile() allows this module to open a JSON file.
2. Both openFile() and makeCSV() sort the data from the file.
3. makeCSV() will then incorporate the formatted data into the system as a CSV file.

Design rationale:
It provides the ability to have an operational system be active upon installation since the system relies on present instructor data, and the Instructor Data Parser servers to supply that data. Once the data file has been made, the parse module does not need to be used again. Therefore, this module is not strongly woven into the entire system. The data will also be stored locally as we decided that non-local storage adds a level of complexity that does not benefit this project in a meaningful way.

Alternative designs:
Originally, an idea was to have student users manually parse the data. However, this interfered with the specifications of this system.

A module that would build a MySQL database and query it was considered, however we decided that it added an unnecessary layer of complexity which led to the decision of local storage.

Another alternative design was to sort the file and create individual class instances for each course, however this was ultimately decided against because it could cause performance issues.

4.2 Web Scraper

This module's role and primary function:
The purpose of this scraper module is to search for faculty information from web pages, specifically the Wayback Machiene's archive containing the names of UO's faculty members. This module will take the HTML files from the different department web pages and form TXT files for local data storage.

Model:

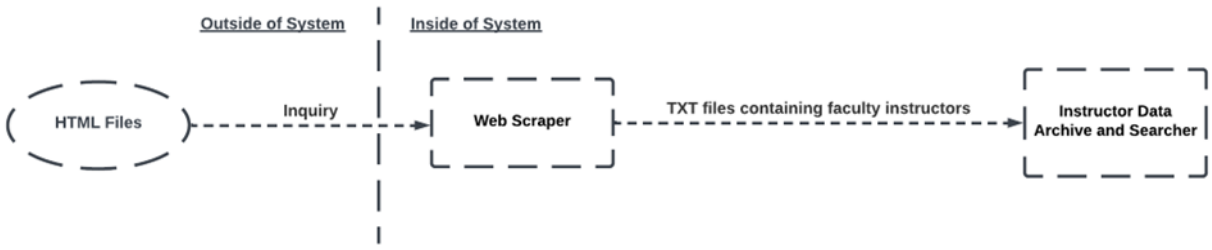

Figure 4.2. Simple diagram showing how the Web Scraper module would interact with the system. This is done prior to the system's initialization.

Interface specification:
This module will not interact with other parts of the code outside of providing a file that the other parts of the program will rely on. The module must read from HTML files and find the faculty name to enter for each line in a TXT file. Each TXT file contains all faculty from a specific department.

Design rationale:
It provides the ability to have an operational system be active upon installation since the system needs data on who's faculty or not in addition to the data on instructors, and the Web Scraper serves to supply that faculty information. Once the faculty TXT files have been made, the scraper module does not need to be used again. Therefore, this module is not strongly woven into the entire system like the parser module. The data will also be stored locally as we decided that non-local storage adds a level of complexity that does not benefit this project in a meaningful way.

Alternative designs:
This was not originally considered at the start of the system's development, but it was recommended to incorporate.

4.3 Student Graphic User Interface (Student GUI)

This module's role and primary function:
The purpose of the user interface is to allow the student to interact with the program so they can retrieve the data they want to view.

Models:

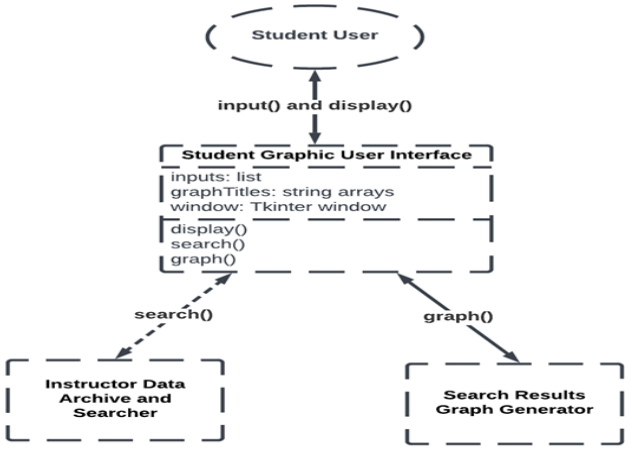

Figure 4.3.1. Static diagram describing how the Student GUI module interacts with other modules.

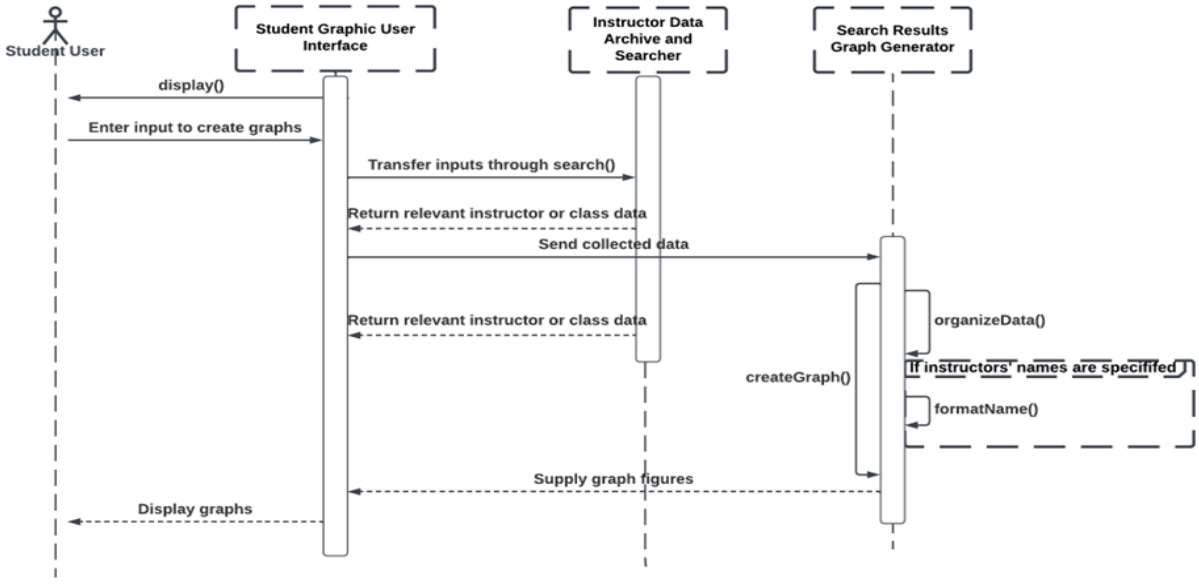

Figure 4.3.2. Dynamic diagram describing how the Student GUI interacts with the user and other modules within the system. This occurs at the start of the system's initialization.

Interface Specification:

The user interface will interact with the system's main file, Instructor Data Archive and Searcher, and Search Results Graph Generator. Below, a list of the order is provided.

1. The main file will run this module's display() function to provide a window for the user; in the module, it refers to this window as "window".
2. The user provides input that the Student GUI will use for calling on the Instructor Data Archive and Searcher through search(). The inputs get stored within this module and referred to as "inputs". A specific input that decides on the graph title gets stored elsewhere and it is known as "graphTitles".
3. Once the Instructor Data Archive and Searcher supplies the gathered data, the Student GUI will transfer that data over to the Search Results Graph Generator to compile graphs through graph().
4. Once graph() returns graph figures to display, the Student GUI will format those figures and incorporate them in display().

Design Rationale:

The Student GUI serves as a vital module for student users since we need a way to figure out what kind information that students would want to see. The student user will use the main file to start up the Student GUI. The GUI will always be running and refresh its window after the student closes the current graph displays. This ensures that the program will continue to run until the user does not need it anymore. The Student GUI communicates with the Instructor Data Archive and Searcher module as the information the GUI receives from the user will aid the

search of data.

Alternative Designs:
The GUI was originally just only going to be transferred over to the main file, but we can make more use of this module to make the system more efficient. The main file, which would have been the EasyA module, had a much bigger role in navigating data around and allowing data to interact. However, we found that it made more sense for the Student GUI to do this because it would already hold access to the inputs and data. It also would have added more data interactions and made for tight coupling if we placed much emphasis on the main file.

4.4 Instructor Data Archive and Searcher

This module's role and primary function:
The purpose of this module is to create data storage when the system is active and apply the user's input to look for relevant data in the CSV file, which will incorporate the TXT files. When any relevant data is found, it will be used to ultimately calculate the numbers needed to make a graph.

Models:

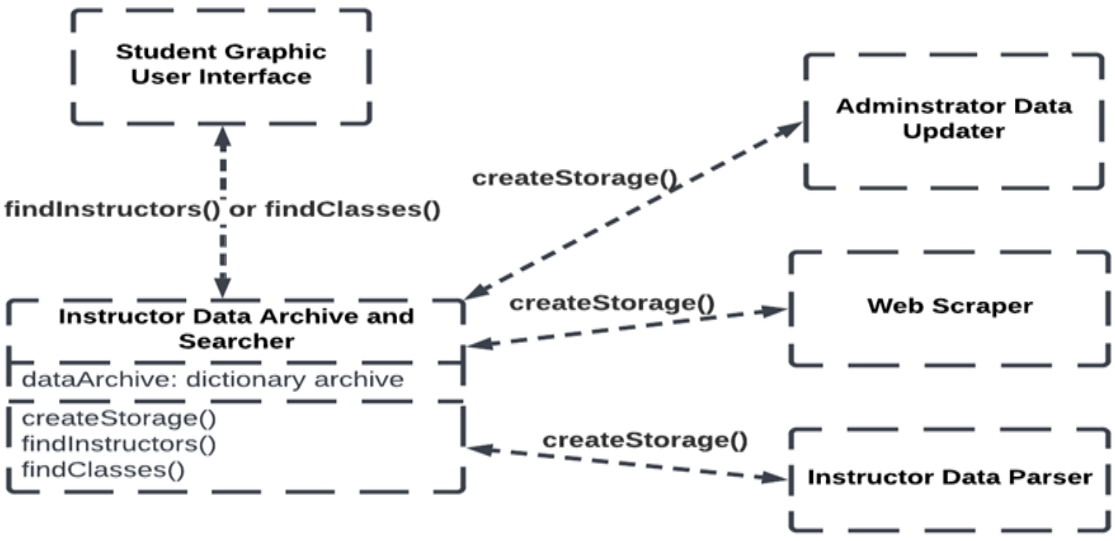

Figure 4.4.1. A static class diagram describing how the Instructor Data Archive and Searcher will interact with other modules.

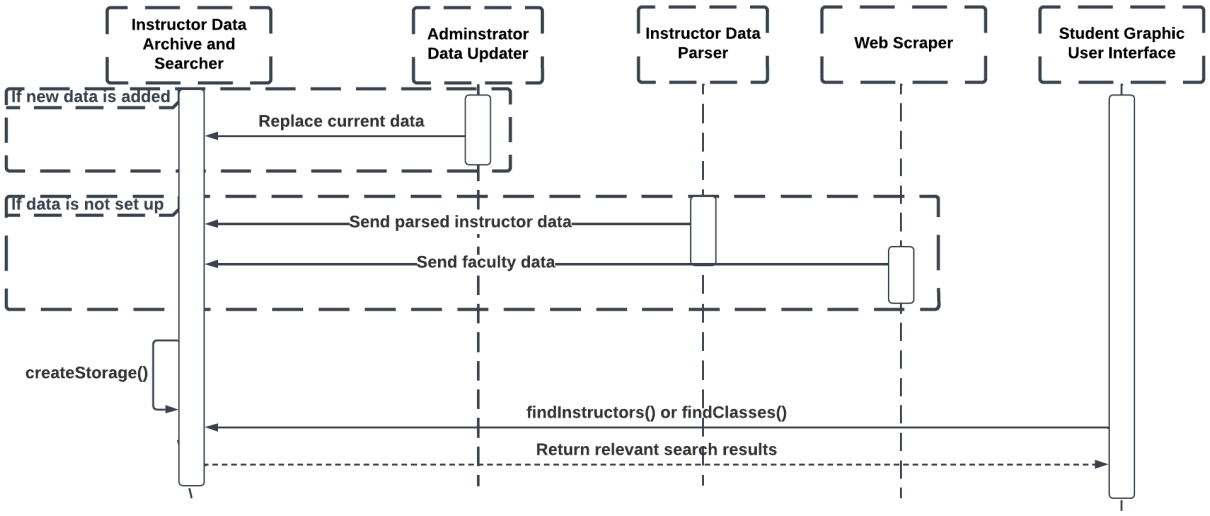

Figure 4.4.2. A dynamic diagram describing the order of how the Instructor Data Archive and Searcher will interact with the other modules; the Administrator Data Updater, Instructor Data Parser, and Web Scraper modules can be considered interchangeable in their sequence and done outside of students trying to interact with the system.

Interface specification:
This search module will directly interact with the Student GUI module and rely on the data for the other modules. These are done through createStorage() for the Student GUI interactions and findInstructors() and findClasses() modules for the Administrator Data Updater, Instructor Data Parser, and Web Scraper.

- createStorage() uses the provided data to make the data accessible for the system.
- findInstructors() and findClasses() calculate the requested data on instructors or class levels respectively.

Design Rationale:
This module serves as a central component for data interactions since it draws on the data from other modules and will use their compiled information to store. This module locally stores the different data and allows for efficient, local searches when requested. It interacts with the Student GUI because the Student GUI can extract what the user wants and transfer it to this module to apply their request. Furthermore, we decided that the search results will transfer to the Student GUI to graph because it would also contain the user inputs that the graphs need to know.

Alternative Design:
We thought of using a main module known as the EasyA module that would be responsible for much of the data storage and interactions in our system. We still had a Data Searcher module, but we found it to be more efficient and comprehendible to incorporate the data storage access aspect into this module. This allows the search aspects to happen more locally and become less

complicated as a result.

4.5 Search Results Graph Generator

This module's role and primary function:
The purpose of this module is to generate graphs with data obtained from the Instructor Data Archive and Searcher module.

Models:

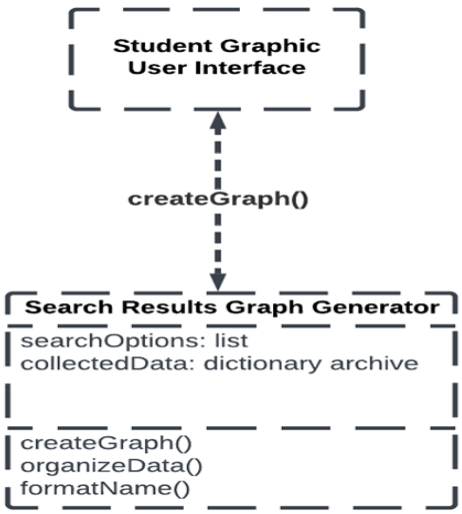

Figure 4.5.1. Static diagram of the Search Results Graph Generator.

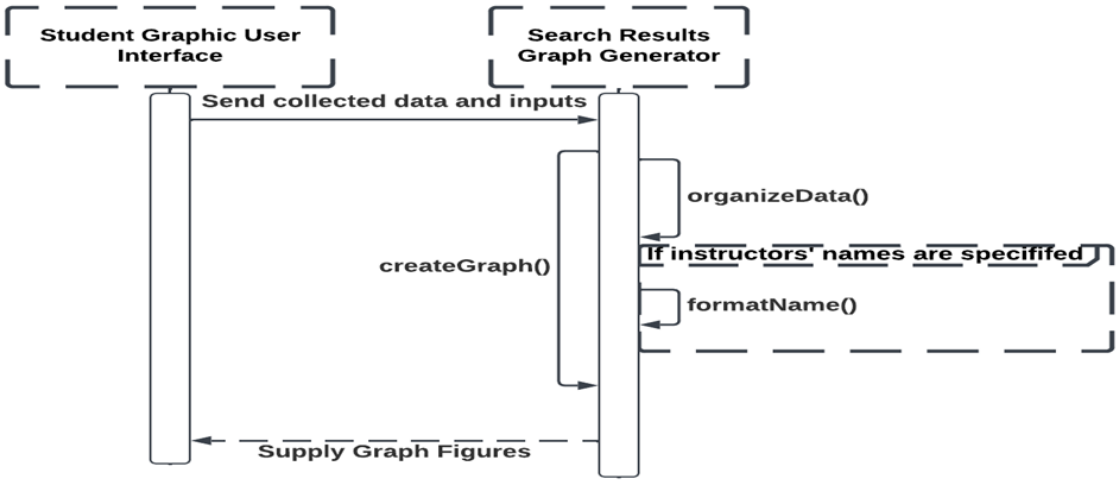

Figure 4.5.2. Dynamic diagram of the Search Results Graph Generator. This occurs after the Student GUI has interacted with the Instructor Data Archive and Searcher module.

Interface Specification:
With the system, this module only interacts with the Student GUI module. There is only one function binding the two together: createGraph(). This generates the graph based on the data results from the Instructor Data Archive and Searcher module and input from the student user. The collected data results are referred to as collectedData and the student inputs are contained in one spot known as searchOptions within the system. createGraph() also calls upon other functions to help it work properly and establish a clear organization within the module.

Below, it describes how this module and the Student GUI module interact with createGraph().

1. The Student GUI calls upon createGraph() first with the necessary information.
2. createGraph() receives input to establish collectedData and searchOptions.
3. createGraph() calls upon helper functions organizeData() and formatName() to help format the data to be presented in a bar graph.
4. createGraph() uses information in searchOptions to label the graphs correctly then display it on the Student GUI.

Once the graph is displayed, the user will have control to remove certain graphs and continue searching data to display in future graphs.

Design Rationale:
This module serves as the last stage within a single iteration within the system. Therefore, it only needs to interact with the Student GUI module since that module will have everything else established and collected prior. It receives the info needed to make a graph then sends the graph to display for the user. Once this is done, the program will display the image until the user is finished looking at it, and the program offers an option to remove a displayed graph for room to generate a new graph.

Alternative Designs:
The original plan was to have the Instructor Data Archive and Search interact directly with this module as opposed to indirectly through the Student GUI. The purpose for this shift in design is that the Instructor Data Archive and Searcher did not need all the input collected from the GUI. As a result, the Search Result Graph Generator could not acquire all the necessary student inputs from the Instructor Data Archive and Searcher module; it needed those inputs from the Student GUI.

4.6 Administrator Data Updater

This module's role and primary function:
The purpose of this module is to enable an administrator to add data to replace the current list of classes and instructors and whether instructors are faculty or not.

Figure 4.6.1. Static diagram of how the Administrator Data Updater interacts with the Administrator User and Instructor Data Archive.

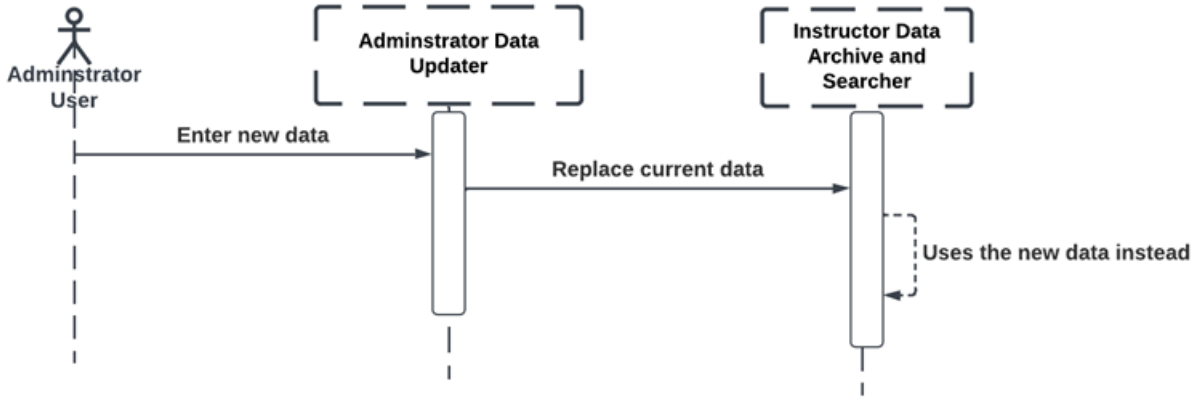

Figure 4.6.2. Dynamic sequence diagram of how the Administrator Data Updater interacts with the Administrator User and Instructor Data Archive and Searcher. This is done outside of any student user interactions with the system.

Interface specification:
This module will interact with the main program file and the user input module. If an administrator indicates they want to add data through the user input module, the add data module will be called. Then based on the data the user inputs it will be added to the end of the existing CSV file containing instructors and classes.

Design Rationale:
The rationale behind having Administrator Data Updater as the last part to be implemented is that once the other parts are working; the addition of new data will not affect the program. This will ensure that the core functionality of the program has been implemented and tested before the ability to update data is implemented. This module will replace the current instructor data CSV file and faculty TXT file.

5. Dynamic Models of Operational Scenarios (Use Cases)

5.1 Student

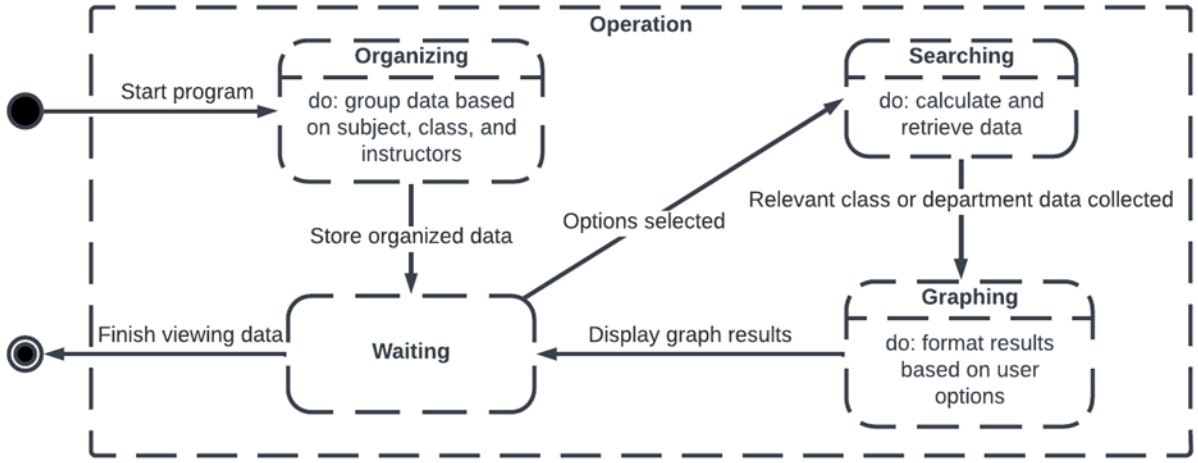

Figure 5.1. The state diagram depicts the different system states when a student interacts with the system. The system starts out in its Organizing state to make the data usable for the system. A student looking into MATH 111 would choose “MATH 111” for one option and select other relevant options during the system’s Waiting state. The system transitions into its Searching state then its Graphing states to generate the desired results.

5.2 System Administrator

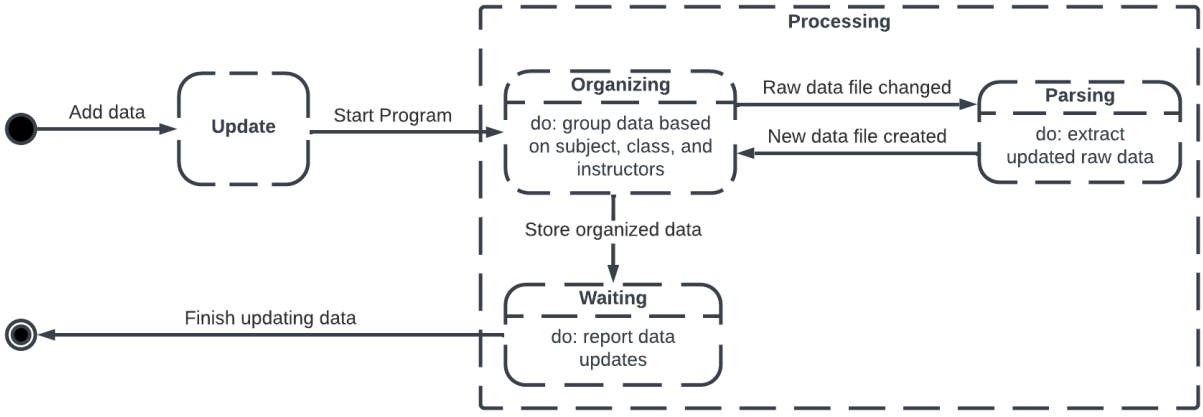

Figure 5.2. The state diagram depicts the use case of administrators updating the data within the system. They would supply their own data to be used by the system during the Update state. When using the Administrator System Updater module, the system goes into its Operation superstate to modify the current data in its Modifying state. Once it's done, the data will have been updated and the system reaches its Data Incorporated state right before the user is done.

6. References

Faulk, Stuart. (2011-2017). CIS 422 Document Template. Downloaded from https://uocis.assembla.com/spaces/cis-f17-template/wiki in 2018. It appears as if some of the material in this document was written by Michal Young.

Hornof, Anthony “Project 1: EasyA (or JustPass),” CIS 422 Software Methodologies, 03-Feb-2023. [Online]. Available: https://classes.cs.uoregon.edu/23W/cs422/. [Accessed: 03-Feb-2023].

IEEE Std 1016-2009. (2009). IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. https://ieeexplore.ieee.org/document/5167255

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM, 15*(12), 1053-1058.

Langlois, Peri, Robert Husbands, and Kathleen Darby, “Emerald Grade Tracker,” Daily Emerald. [Online]. Available: https://emeraldmediagroup.github.io/grade-data/. [Accessed: 15-Jan-2023].

Marcus, Andrew “College of Arts and Sciences,” College of Arts and Sciences < University of Oregon, 01-Sep-2014. [Online]. Available: https://web.archive.org/web/20140901091007/http://catalog.uoregon.edu/arts_sciences/. [Accessed: 31-Jan-2023].

7. Acknowledgements

This template builds slightly on a similar document produced by Stuart Faulk in 2017, and heavily on the publications cited within the document, such as IEEE Std 1016-2009. This student group received permission to use this template from Prof. Anthony Hornof.

Additionally, Professor Hornof also supplied the system with a Systems Requirements Specification to base our work from.

The data used for the EasyA system came from the “Emerald Grade Tracker” on https://emeraldmediagroup.github.io/grade-data/ and the data was copied on January 13th, 2023.