

CS3307 – OO Design and analysis, Fall 2015

Ben Dumais, Adam Fuhrer, Jieni Hou

Liam Jowahir-Sindrey, Nicole Levy,

Leon Meyer, Jerridan Quiring, Amha Tsegaye

Benedict Von Holzendorff, Zeyu Wang, Zihao Wang

ACUITY STAR

Team Banana

Contents

[Minimum Requirements 3](#_Toc437450887)

[Stretch Requirements 5](#_Toc437450888)

[Data Persistence 5](#_Toc437450889)

[Data Editing 6](#_Toc437450890)

[System Design 7](#_Toc437450891)

[Use Case Diagram 7](#_Toc437450892)

[Class Diagram 8](#_Toc437450893)

[Sequence Diagram 9](#_Toc437450894)

[Package Diagram 10](#_Toc437450895)

[Design Patterns 11](#_Toc437450896)

[Code & Design Inspection 12](#_Toc437450897)

[C++ Implementation 18](#_Toc437450898)

[Development Plans 19](#_Toc437450899)

[Timeline 20](#_Toc437450900)

[Task List 21](#_Toc437450901)

[Retrospective Analysis & Lessons Learnt 21](#_Toc437450902)

[What Went Wrong? 21](#_Toc437450903)

[System Documenting 22](#_Toc437450904)

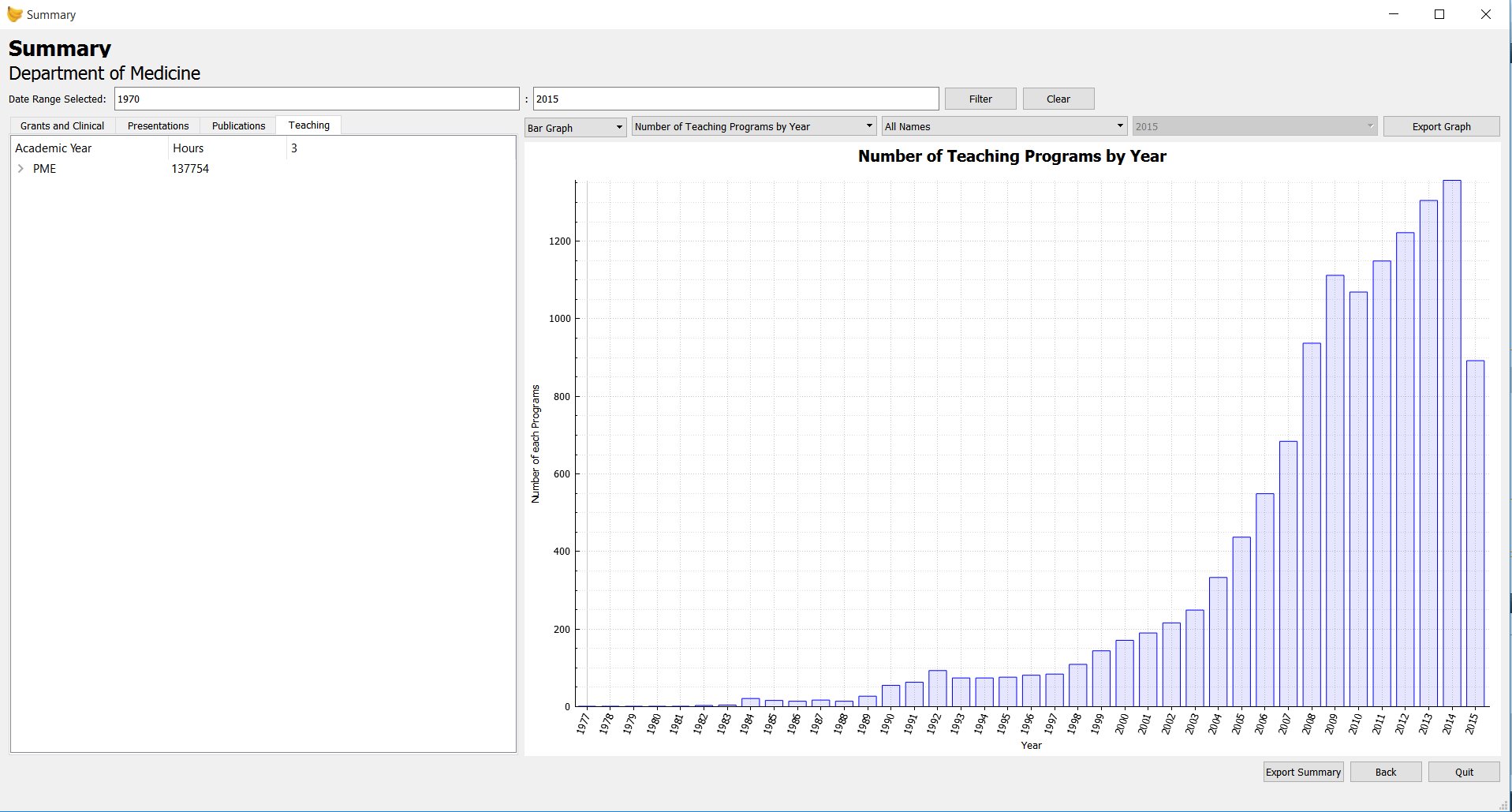
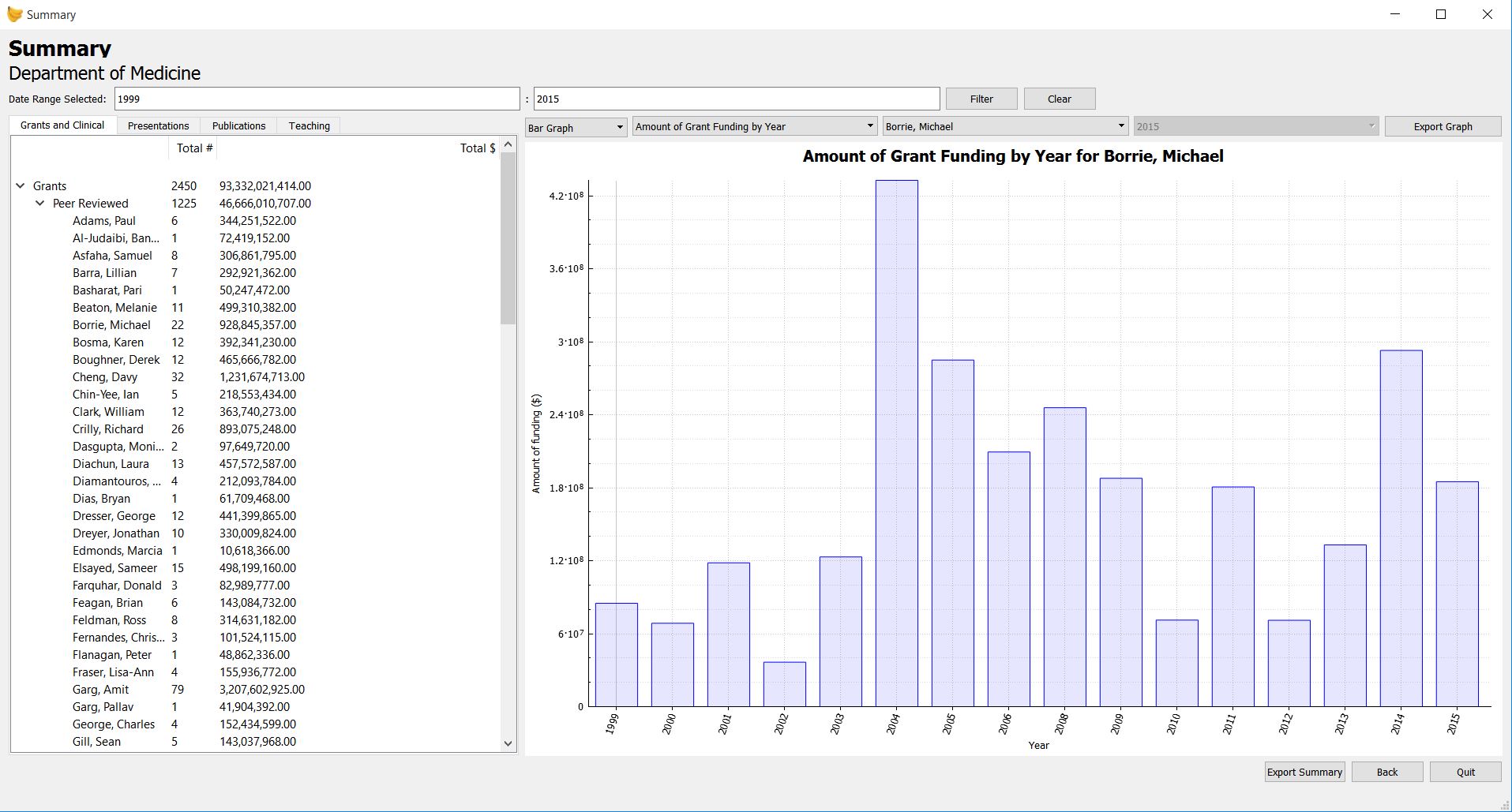
[Analysis of Teamwork 22](#_Toc437450905)

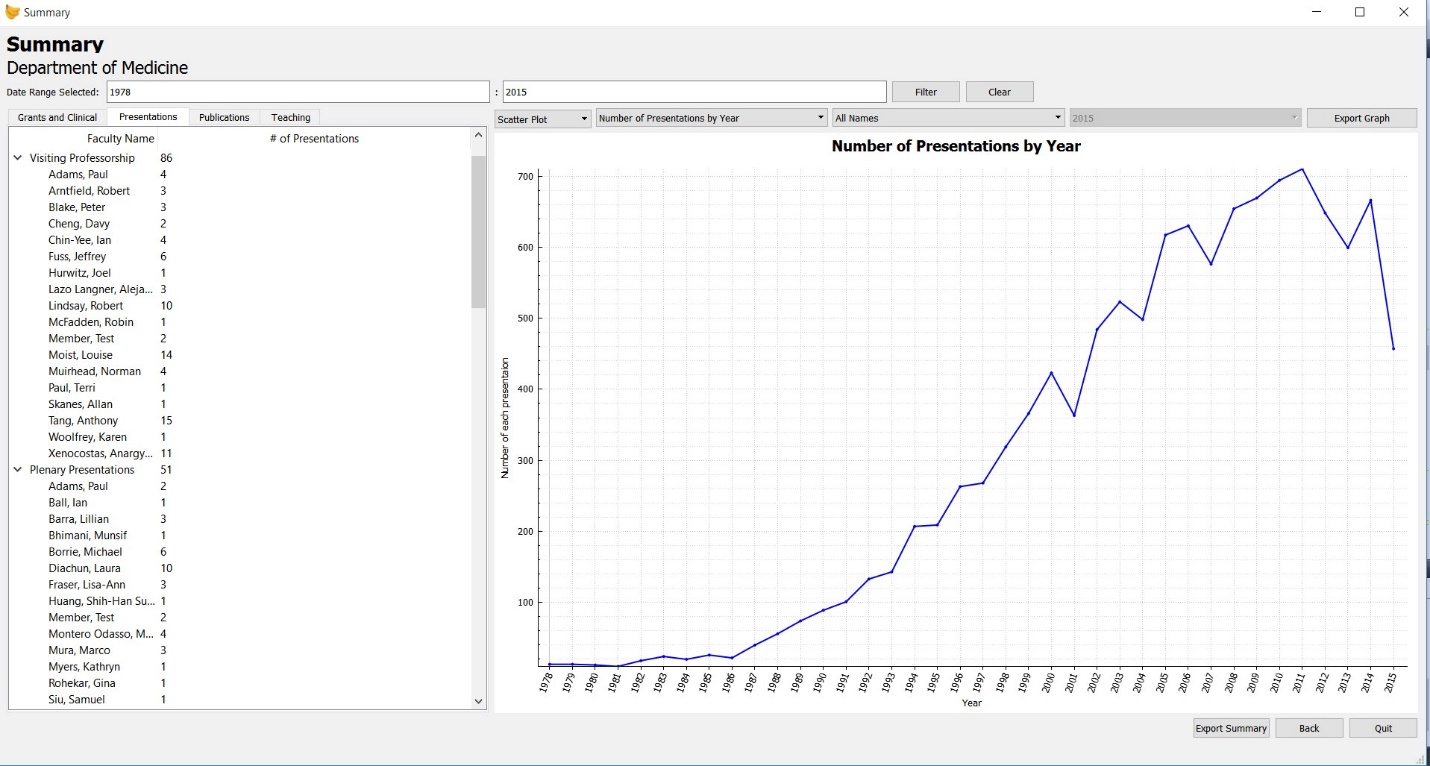
[Customer Interaction 22](#_Toc437450906)

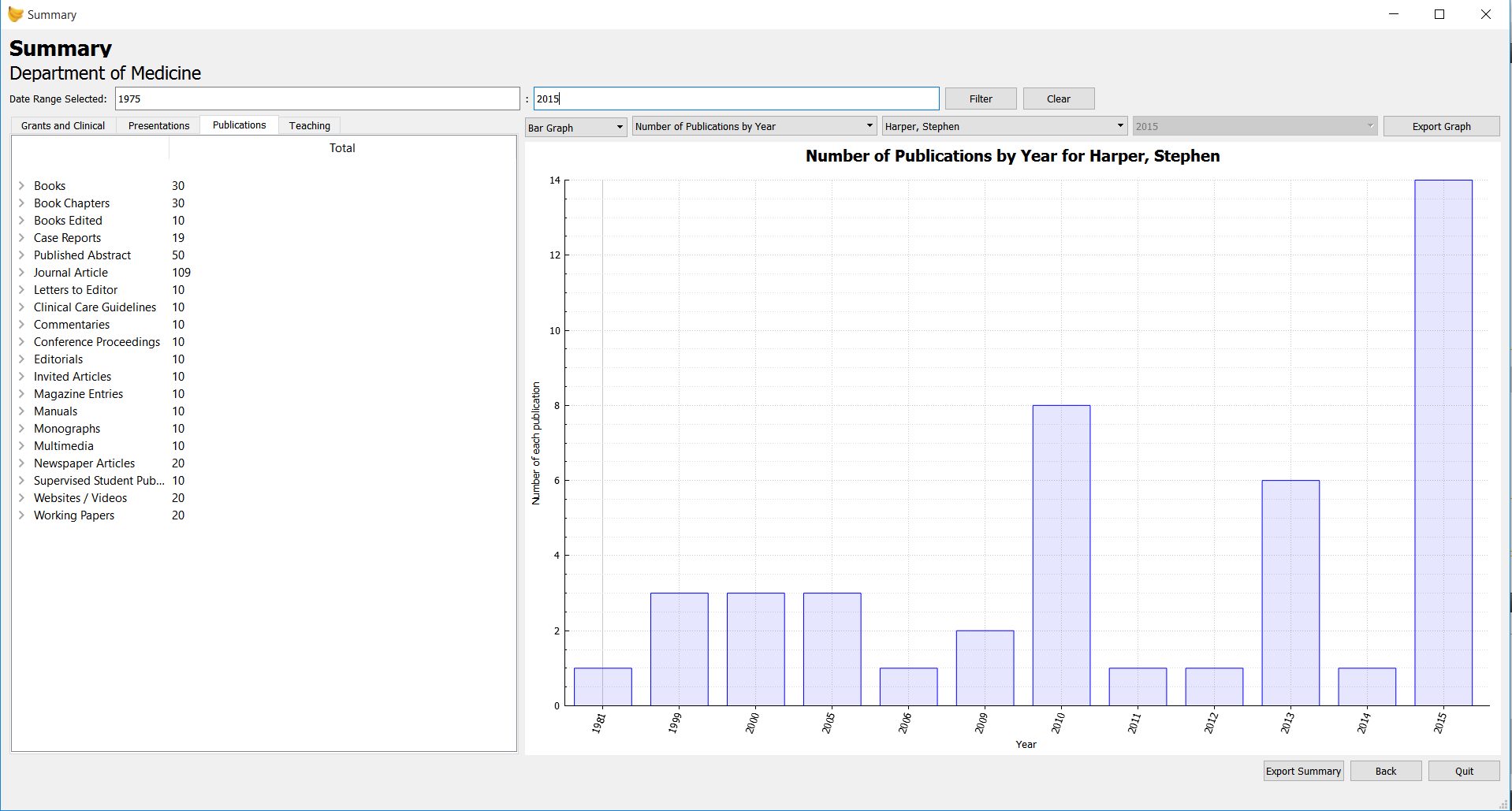
[Recommendations for Future Students 23](#_Toc437450907)

[Value of This Project 23](#_Toc437450908)

# Minimum Requirements







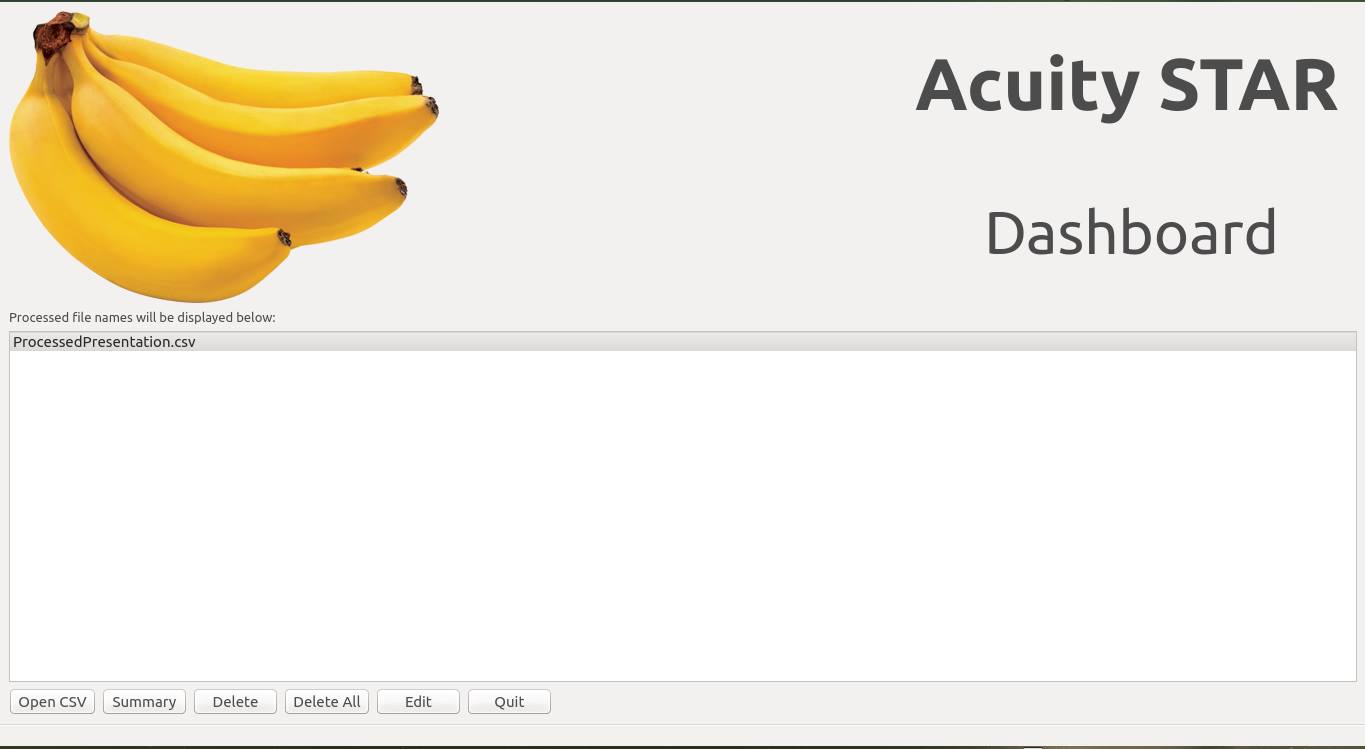
Above shows sample data summaries and graphs for the record types: funding, presentations, publications, and teachings. The dashboards are similar for different data types, where users can change chart type and expand summaries to see more data. The table shows aggregated data from the file. The graphs give visual representation of data. Graph types can be changed through buttons at top, and data can be filtered using date filters. User can also export the summary to pdf for later viewing or printing (program takes a screenshot of what current summary table widget looks like).

*General use case:*

* Open program
* Click “Open CSV” button
* Select CSV from file selector
* System displays the processed csv in list
* Select file from list
* Click “Summary” button at bottom of screen
* System displays graph and table of data from file
* Expand summary of data by pressing arrows in table
* Change graph type and filters through options at top right

# Stretch Requirements

## Data Persistence

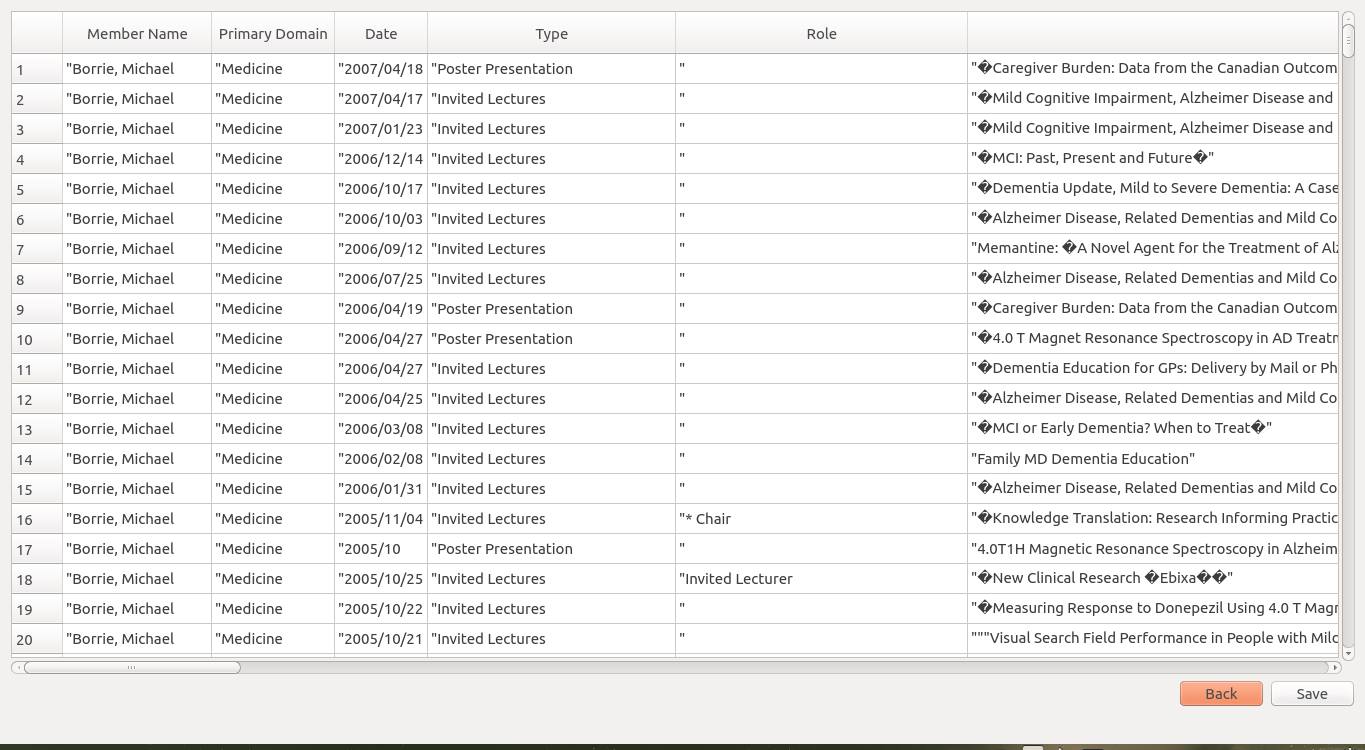


Data persistence has been implemented as part of the project. When a user loads a csv into the project, a new CSV containing the processed data is created. The program will also load up processed CSV files when it opens, meaning that when a user closes the program and then re-opens the processed CSV is still available for use. This allows ease of use for the customer and saves them time and processing.

*Use Case:*

* User opens project
* User clicks “Open CSV” button
* User selects desired CSV from selector
* System processes CSV file and displays it in the processed file list
* User closes project
* User reopens project
* System displays processed CSV in list

## Data Editing



As a stretch goal, we have implemented the ability to edit a csv file. This is meant to help users make quick changes or fix errors in the data without needing to re-download and reprocess an entire file.

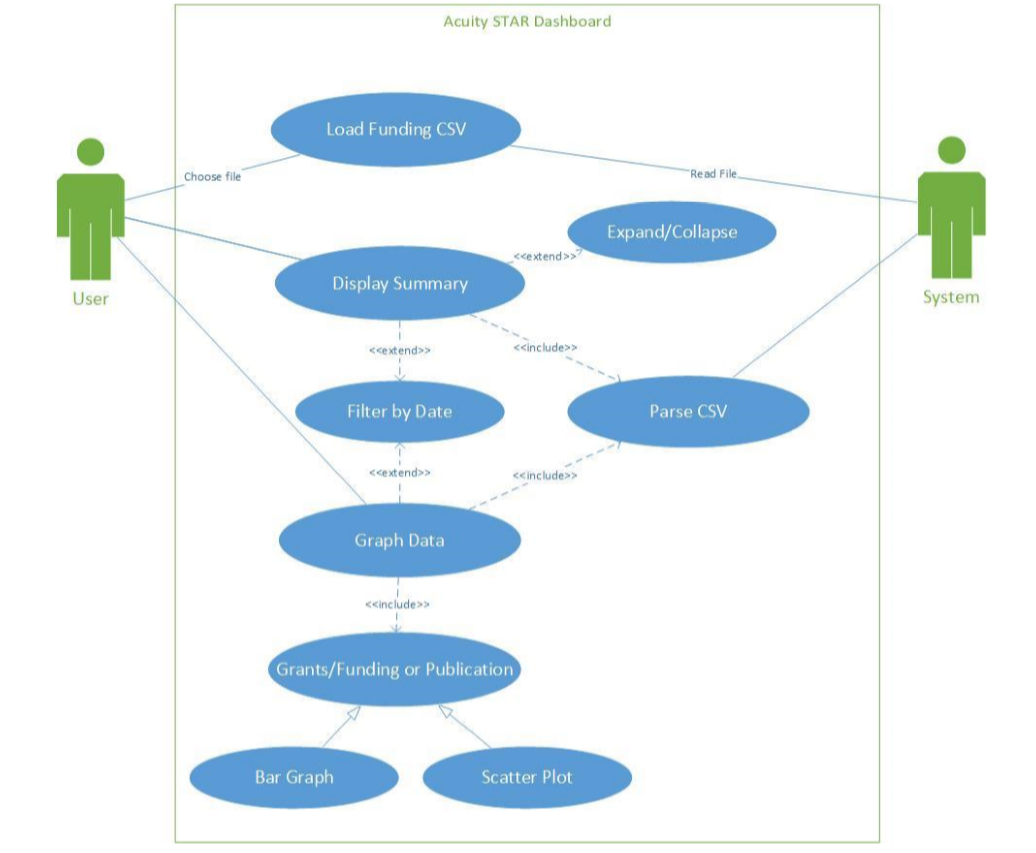
*Use Case:*

* User opens program
* User selects processed file from list
* User clicks “Edit” button at bottom of screen
* System loads csv into an editable table and displays it to user
* User makes changes to data
* User clicks “Save” button
* System updates CSV

# System Design

Original documentation can be found in both repository and in final submission package, if images are too blurry.

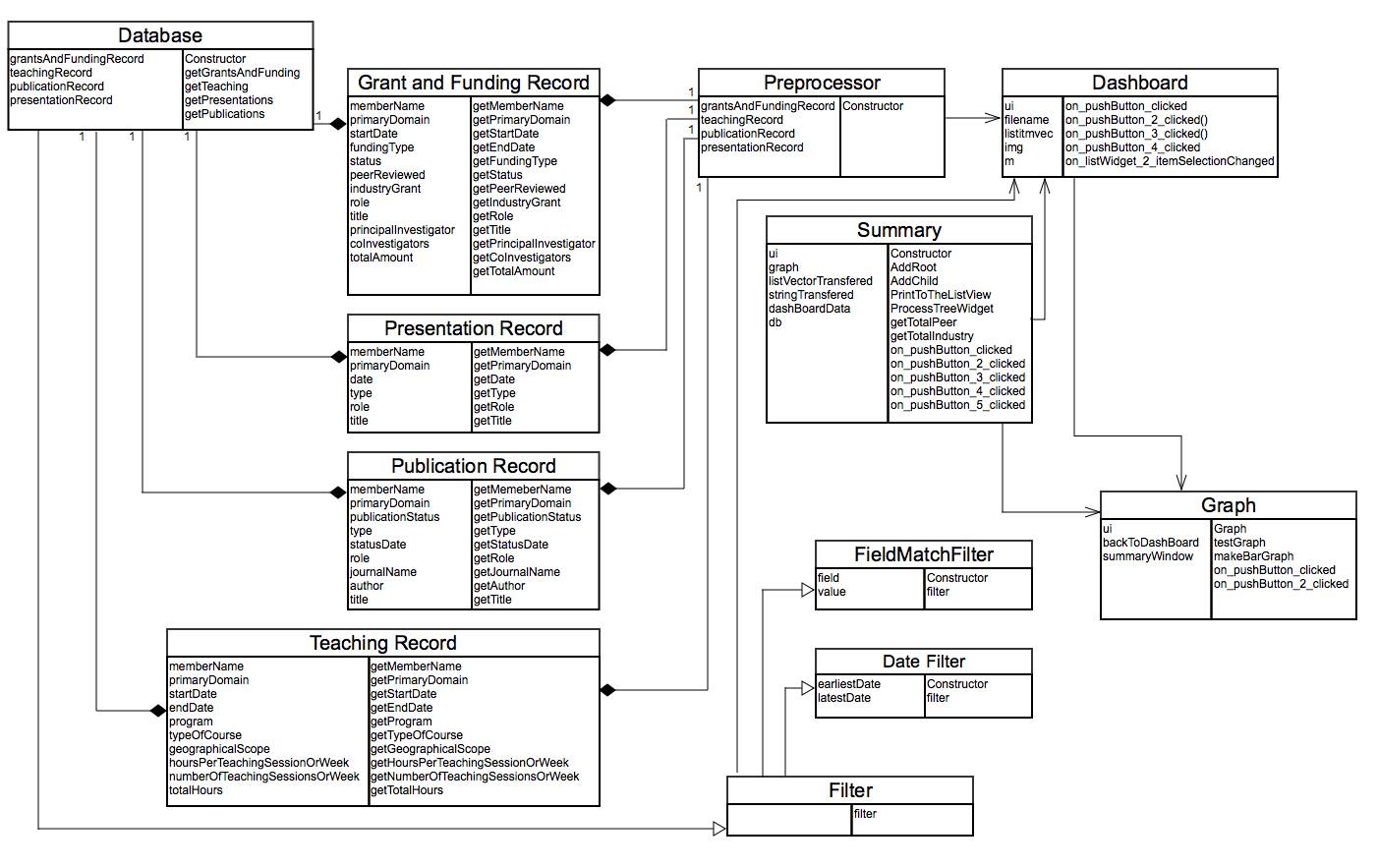
## Use Case Diagram



This use case diagram conveys the different use cases for each actor, and the relationships between them. The User’s use cases come directly from the customer requirements. The User can: choose the CSV to load, display the summary for grants/publications (optional expand/collapse), and display a graph of the. The System can: read the chosen CSV, and parse the data for the summary and graphs.

Above depicts use cases for Stage 2; in the final product all of the customers’ requirements are covered by a use case. Some other use cases are editing data, and viewing data for presentations or teachings.

## Class Diagram

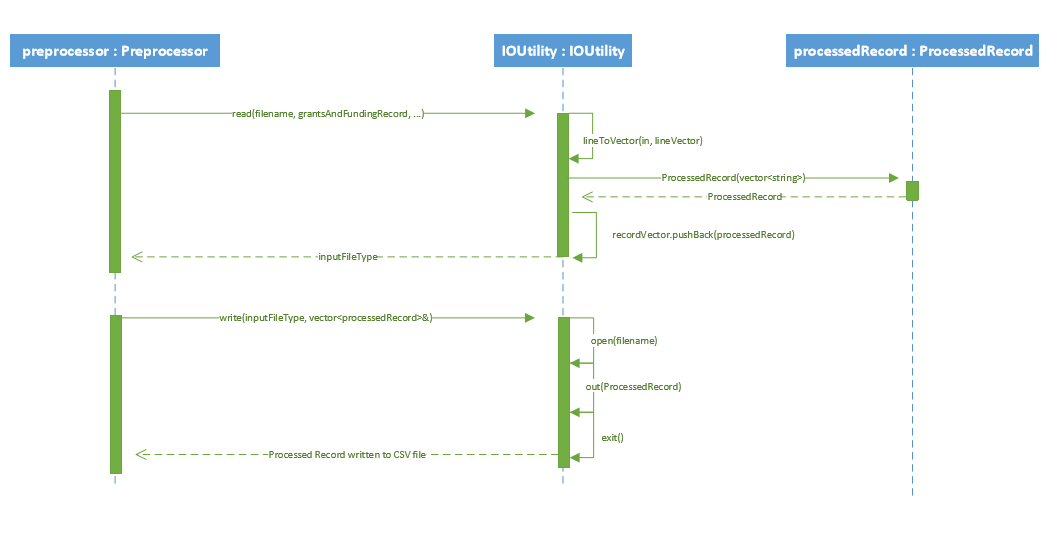
The UML Diagram better conveys how each object in the system interacts with one another. It further breaks down the front and back end into classes and shows how they use or implement one another to create the final system. It also shows the separation of the front and back end.

When approaching this assignment, it was clear that there were two broad issues to tackle:

1. Reading the raw data and store it internally to allow for fast access and search.
2. Displaying the data for consumption by the user.

This lead to our breaking into front and back end teams, which allows us to better focus on each issue and separate them from one another. But first we needed an interface defining how these two components of the system would interact. The interface consists of a simple accessor method that uses filter objects to implement a kind of query language. The filter class is an abstract class that can be extended as necessary to provide any functionality needed. In this way, the project can be divided into two separate areas of concern with a single point of access that can be extended to provide functionality as needed.

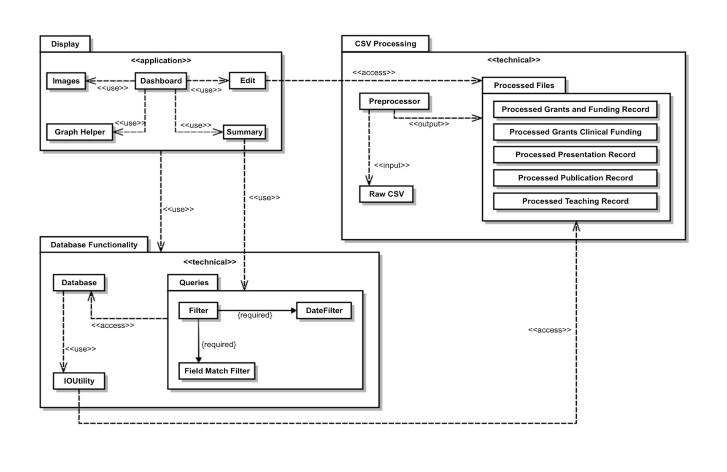
## Sequence Diagram



The above diagram shows how data is processed and stored from the raw CSV files. The Preprocessor object calls the IOUtility class to read lines of the input CSV, store them as processed records, and then add them to the list of records. It also shows how the system saves the processed records into a new CSV file for later use, hence avoiding extra processing. (Note: Processed Record refers to a type of processed record object, ie ProcessedGrantsAndFundingRecord, ProcessedTeachingRecord, etc).

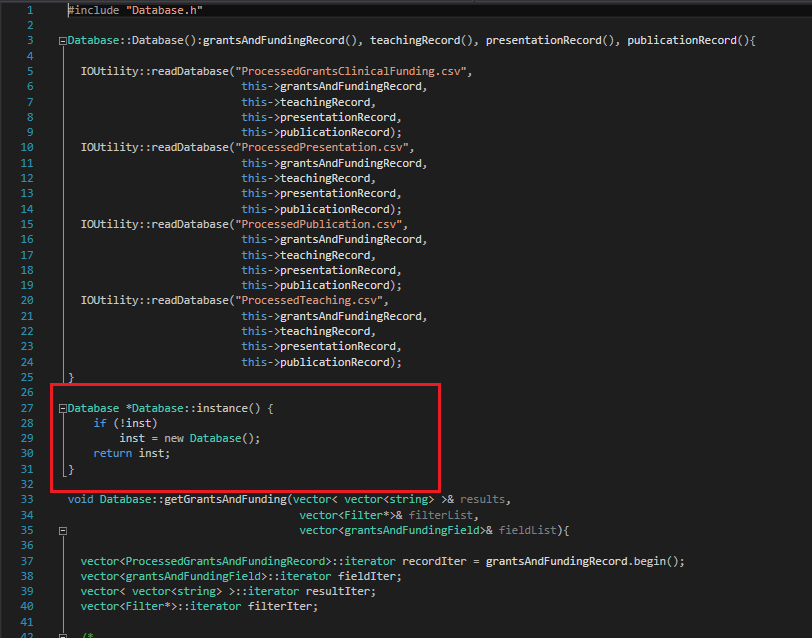
This design was chosen as it prevents redundant processing of data: with this design, an input file only needs to be read and processed once, rather than needing to read and process and reprocess data over the course of usage.

## Package Diagram

With the package diagram, the goal was to show how the different parts of the application are structured and how they relate to each other. Our program has been separated into three separate packages that are effectively the front end (display), the back end (CSV Processing), and the interface between them (Database Functionality).

The Display package consists of the various classes that are required to display the information to the user and allow the user to interact with the data. Database Functionality contains the classes and methods that mimic database queries to make it simple for the front end to access the data, as well as to create a layer of abstraction between the front end and the raw data files. CSV Processing is made up of the unprocessed csv files, the preprocessor which interprets these files, and the processed csv files used to create data permanence within the program. This allows changes made by the user to be saved and used later without reprocessing.

# Design Patterns



The Singleton design pattern WAS going to be used for the Database object. As the Database stores vectors of all processed records, there should only ever be one instance. Having one instance ensures all data is stored in the same location, allowing for better management and control of data. The Singleton pattern would have allowed us to ensure only one database instance ever exists and that we are not reading or writing data to somewhere else.

The above highlights what was coded for the instance method for Database.

However, due to time constraints and the uncertainty surrounding changing references to a fundamental object near the deadline of a project, we ultimately scrapped implementing the singleton design pattern. While it could have been helpful in managing data, development was too far along for such changes to be made with ease and without extensive debugging. In the end, we chose to stick with the existing database constructor in the interest of time and delivering a product that was tried and true throughout development.

# Code & Design Inspection

Using Tool and analyzing the code for displaying summaries of data. Original tool and document can be found with submission and in repository.

**Structural correspondence between Design and Code:**

Are all the classes and interrelationships programmed in the application explicitly represented in the class diagram of the system?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Analyzed by comparing UML design to summary.h

Comment on your findings: Additional relations to other classes not in UML exist, and some additional UI classes are not in UML

**Functionality:**

Do all the programmed classes perform their intended operations as per the requirements?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Compared functions implemented in .cpp files to design

Comment on your findings: Each class needed for a summary has an intended purpose and each fulfills it. There are no classes that lack functionality specified in system design

**Cohesion:**

Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion: the functionalities embedded in a class, accessed through its methods, have much in common, e.g., access common data)

🞏 Yes 🞏No 🞏Partly (Can be increased)

Comment on your analysis: Analyzing code for summary.cpp and examining the inputs or function for each method for common data and purpose

Comment on your findings: The purpose of the class is very clear, and while it deals with a lot of data, it is specific about which data it works with. However, there are several functions that handle some processing for other features such as graphs.

**Coupling:**

Do the programmed classes have excessive inter-dependency? (High Coupling: In this case a class shares a common variable with another, or relies on, or controls the execution of, another class.)

🞏 Yes 🞏No 🞏Partly (Can be reduced)

Comment on your analysis: Analyzed what data or classes functions rely on in summary.cpp

Comment on your findings: Many classes are used and relied upon for functionality. While this is an unfair class to analyze for this (as it serves as a hub for analysis), it depends a lot on data, references and other classes to function

**Separation of concerns:**

Is the scoped problem decomposed into separate concerns where each concern is encapsulated in a construct such as a class with well-defined interface and cohesive functions with minimal of connections with other concerns?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Analyzing main purpose of functions in summary.cpp

Comment on your findings: While Summary serves a purpose of summarizing the input data and serves mostly just visualization purposes, it does also handle some functionality for graphing.

Do the classes contain proper access specifications (e.g.: public and private methods)?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Analyzed summary.h declarations

Comment on your findings: The only public methods are to do with loading and savings settings, as well as simply calling up the class. All other functionality is private.

**Reusability:**

Are the programmed classes reusable in other applications or situations?

🞏 Yes, most of the classes 🞏No, none of the classes 🞏Partly, some of the classes 🞏Don’t know

Comment on your analysis: Examined dependencies of summary.h

Comment on your findings: If creating a similar program to analyze csv files, then the summary would useful to reuse. However, it is very dependent on QT, so attempting to develop outside of QT would not be possible.

**Simplicity:**

Are the functionalities carried out by the classes easily identifiable and understandable?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Looked at commenting and followed logic of methods

Comment on your findings: At first glance it seems like a mess of variables, methods, and conditional statements but if you read the code names are easily understood and the logic is easy to follow.

Do the complicated portions of the code have /\*comments\*/ for ease of understanding?

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Read through summary.cpp for comments

Comment on your findings: There are comments for methods and sections of code

**Maintainability:**

Does the application provide scope for easy enhancement or updates? (e.g., enhancement in the code is not anticipated to require too many changes in the original code)

🞏 Yes 🞏No 🞏Partly (Can be improved) 🞏Don’t know

Comment on your analysis: Analyzed functions to do with processing and storing data

Comment on your findings: Additional types of data can easily be added, provided they are processed and stored in similar processed record objects.

**Efficiency:**

Does the design introduce inefficiency in code (e.g., causes too many nested loops or delays in concurrent processing)?

🞏 Yes 🞏No 🞏Partly (Can be improved) 🞏Don’t know

Comment on your analysis: Looked for complex code or nested loops

Comment on your findings: As we separated backend from frontend, there are very few loops or processing methods in the class. Loops are mostly from looping through a vector of data. However, due to time constraints, we did have to add some processing into summary, which has hurt the efficiency. There are also some redundant queries or sorting of data

**Depth of inheritance:**

Do the inheritance relationships between the ancestor/descendent classes go too deep in the hierarchy? (The deeper a class in the hierarchy, the greater the number of methods it will probably inherit from its ancestors, making it harder to predict its behaviour).

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Looked through classes for inheritance from or to summary class

Comment on your findings: There are no inheritance relations with summary but for implementing some QT classes for the GUI

**Children:**

Does a parent class have too many children classes? (This could possible suggest an abstraction problem.)

🞏 Yes 🞏No 🞏Partly (Can be improved)

Comment on your analysis: Looked through classes for inheritance from or to summary class

Comment on your findings: There are no inheritance relations with summary but for implementing some QT classes for the GUI

**Behavioural analysis:**

From the system’s requirements, **create several scenarios** starting from the **user’s** point of view: consider identifying one or more **typical** scenarios (e.g., those expected to be used with high frequency) and one or more **low-frequency** scenarios .

Each scenario is described as follows:

1. Title of scenario
2. Anticipated frequency of use (high, normal, low)
3. End-user trigger (starting point) for the scenario.
4. Expected type of outputs.
5. List of bullet points linking end-user inputs and identifying all the key features of the system expected to be “touched” by the scenario and producing the anticipated outputs.

Follow the code (structured walkthrough) to ascertain whether this scenario is properly implemented both in terms of logic and design.

Comment on your findings, with specific references to the design/code elements/file names/etc.:

**Scenario 1: Opening a CSV**

Frequency of use: High

Starting point: Dashboard screen (opens on launch)

Expected output: Processed CSV that can be summarized and graphed

Workflow:

* Upon launch the program, user clicks ‘Open CSV’
* User selects their desired CSV file
* The processed CSV appears in the dashboard box, and available for summarizing and graphing.

**Scenario 2: Edit a CSV**

Frequency of use: Medium

Starting point: Dashboard screen, with the CSV in question already opened

Expected output: Error-free CSV, ready for summarizing and graphing

Workflow:

* User highlights the CSV that they want to edit in the dashboard box
* User clicks ‘Edit’
* A new dialog opens showing the CSV file contents as tabular data
* The user makes the appropriate changes within the dialog box
* The user saves the changes

**Scenario 3: Viewing a Summary and Graph**

Frequency of use: High

Starting point: Dashboard screen

Expected output: Summary and Graph of data

Workflow:

* From the Dashboard, the user clicks ‘Summary’. Note that there must already be at least one CSV opened
* The Summary screen will be opened, displaying summary data on the left, and the graph on the right
* Opened CSV files can be switched by choosing the appropriate tab (ex ‘Grants and Clinical’)
* The graph shows the data of the currently opened summary tab

**Scenario 4: Filtering Data**

Frequency of use: Medium

Starting point: Summary screen

Expected output: Summary and graph displaying filtered data

Workflow:

* As an example, imagine the user wants to filter data between the years 1990-2000
* At the top of the Summary screen, the user locates the ‘Date Range Selected’
* The user enters 1990 in the first input and 2000 in the second input
* The user clicks ‘Filter’, and the summary table and graph are automatically changed
* The user may click ‘Clear’ to clear the input boxes and remove the filter

**Scenario 5: Exporting the Summary and Graph**

Frequency of use: normal

Starting point: Summary screen

Expected output: A PDF of the summary table or graph

Workflow:

* To export the summary table, the user clicks ‘Export Summary’ at the bottom right of the screen
* To export the graph, the user clicks ‘Export Graph’ above the graph, top the right
* In either case, a save dialog will open, allowing the user to select the name and location of the exported PDF
* The user clicks ‘Save’, and the PDF is generated

**Scenario 6: Changing the type of Graph**

Frequency of use: high

Starting point: Summary screen

Expected output: A specific type of graph, based on the summary table data

Workflow:

* Once the appropriate data has been chose in the summary table and filtered appropriately, the user may want to change how the data is graphed
* Above the graph, there are 3 select boxes
* In the first select box, the user may switch between a Bar Graph and a Scatter Plot
* In the second, the use may toggle between available graphs for the current data set, for example, for Grants and Clinical data:
  + Amount of Graph Funding by Year
  + Number of Grants per Year
  + Roles taken in grants for single year

In the third select box, the user may choose to include all researchers in the data, or choose a single one by name

# C++ Implementation

At final submission of the project, our code satisfies our original design and specifications. As per our design, or code is split into front and back end systems while passing references to data between them.

Included in the back-end system is our code to process data into a processed object and storing it. Our IOUtility handles this functionality, parsing the raw csv files and reading out the specific fields that we are then able to store in one of processed record objects (ProcessedPublicationRecord, ProcessedTeachingRecord etc). These objects are then able to be stored in our database object, which houses vectors of each type of record. While this uses up more memory than simply using data that has been queried, this cuts down the amount of processing needed for each command or query that the customer inputs. Overall, this makes it faster for the user and system, using up less resources.

This efficiency is further improved by passing references to record and filters between front end and back-end. By passing references only, we cut down on the amount of data being sent back and forth between the two systems and making it faster for the user. This also allows for better control over data, making sure we only have one instance of a record and are working with/changing only that record. This keeps all data consistent across queries and changes. To implement this, we used vectors of vectors of pointers when passing data. The vectors allow us to pass entire lists (vectors) of data between parts of the system and minimize the amount of data being passed.

For our front-end, our design revolved around a straight forward and easy to use interface. As per our original GUI design, the interface revolves around allowing the user to select a file to load into the system or selecting a file to view. Due to our backend design, we can have multiple processed files ready to view by user. This list of processed files allows the user to change quickly between data sets that they may want to view or see summaries of.

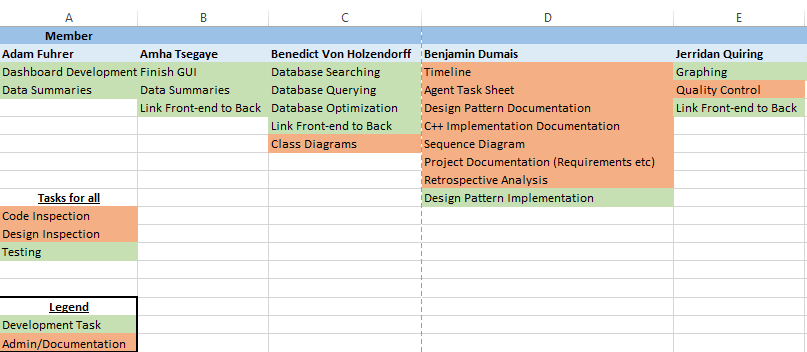
This mentality for easy-to-use interfaces was kept in the summary that comes up when selecting a processed file: users can resize and move around the graph or summary to better view what they are interested in. Summaries are also clearly displayed, showing fields of interest and allowing for the user to easily break the data down into more detail.

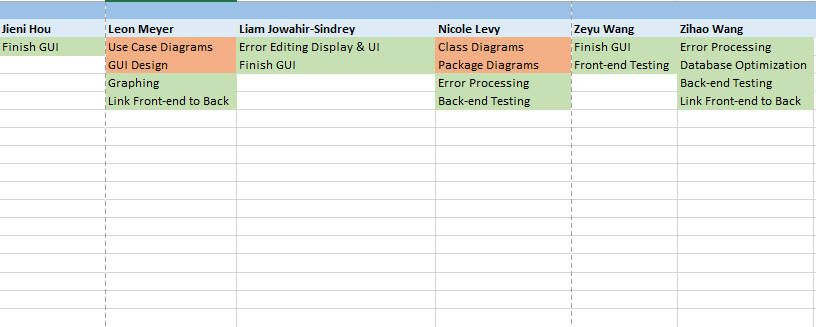
# Development Plans

Original documentation can be found in both repository and in final submission package if content is too blurry.

## Timeline

## Task List





# Retrospective Analysis & Lessons Learnt

## What Went Wrong?

One of the largest issues we ran into was not completely following our designed modularization for the system. From the start we planned on separating the system into front and back end to better focus on the issues of CSV processing and display, opting to have references to data passed between them as to avoid any processing on the front-end. However, as coding went on, the teams became rather disconnected. Combined with a time crunch due to other commitments team members had, the disconnection forced us to incorporate some processing in the front-end in order to make deadlines meet, thus defeating the purpose of separating the two systems.

Another issue we ran into was properly building the program with QT and on different operating systems. As team members used different operating systems and versions of QT to build the project, making sure all necessary tools were on both systems and on the same version when building and compiling the product caused a lot of headaches and slowed progress down quite a bit.

The final major issue we ran into came about due to a lack of automated testing (ie Jenkins). As many of us coded in on our own time rather than at a set time with the team, it was made it difficult to tell if pushed code was breaking the system due to recent changes made by somebody else. Many times some team members would attempt to push their code, only to find unexpected changes to the repository since they last pulled. This caused some confusion with version control and resulted in a few code breaking pushes.

The three lessons that came about from these challenges are:

1. When separating a project into different teams, communication across them is vital
2. Make sure developers are using either an automated build process or the similar environments
3. Version Control and Automated testing are life and time savers

## System Documenting

The majority of the system design was done at the start of the project, well in advance of any coding. Before coding, we designed the system by breaking up the front and back ends as well as agreeing upon how to send data between the two.

Comparing the project to previous projects in which members of the team documented the design after development, creating the ACUITY STAR program was much easier in terms of organization and vision. Having such a clear image of how the system should look and interact with other components made it much easier to split up tasks and code each to specification, rather than justifying code after development. It took a lot of guesswork out of the mix, allowing us to focus on things such as efficiency rather than worrying about how each component needs to interact or what the final product should look like.

## Analysis of Teamwork

Overall, we feel our team was well organized and delivered a great final product to the customer. While there was some disconnection between the front and back end teams, there were no major issues with team contributions or organization.

With that said, there were still some minor issues such as attendance for meetings or communication across the team as a whole. As we are a large team of busy students, issues with attendance are to be expected since we aren’t all free at the exact same times. It was also an issue near the end of the project getting everybody on the same page as we were all busy with other courses or preparing for finals. While we did have an agreed upon meeting time and day set from the start of the semester, the attendance for this meeting slowly dropped off until the meetings no longer took place. This had an adverse effect on deadlines, causing a time crunch near major milestones.

In retrospect, the team could have been better organized if the weekly meetings were made mandatory and enforced more by the team managers. However, in a university setting, such things are unavoidable and we do not feel these minor issues had any major impact on our project performance.

## Customer Interaction

Outside of a few questions near the start of the project, we did not interact much with our customer. While infrequent, our communications with the customer were very smooth and professional. The outlier to this, however, is the issue we had with our stage 2 submission. Due to some of the problems mentioned before with building the project, the client was missing a required .dll to compile and run our code. The issue was further worsened by our very slow response to the customer when they asked us to help fix the problem. While the problem was resolved eventually, it was a rather poor interaction we had with the customer from a customer management stance.

## Recommendations for Future Students

Start early, work steadily, and never underestimate the value of a good design. The majority of our issues stemmed from not working steadily on the project over the semester and instead doing the majority of work close to deadlines. As well, one of the greatest assets our group had was our system design we laid out from the start. Having a thorough and clear image of what the final product should be makes it much easier to organize and build such a large project.

## Value of This Project

This project does a great job of highlighting the importance of design in software development, as well as giving practical experience in collaboration with other developers and customers. Attempting a complex project such as this by yourself is a very difficult task, but tackling the project with others without a clear design would be chaos. Further, the soft skills learned in terms of communication, team management, and simply finding an area of a project to add value to are skills that translate to all areas of life.