Team Hercules

Amber Ali, Shanice Hanlon,
Bowen Jiang, Martin Kilonzo,
Michael Song, Michael
Tassone

Deliverable 5

CS 3307A: Object Oriented Design & Analysis
2016

CONTENTS

1 EXECUTABLE

See .exe for an example of the sample executable.

2 FEATURES IMPLEMENTED

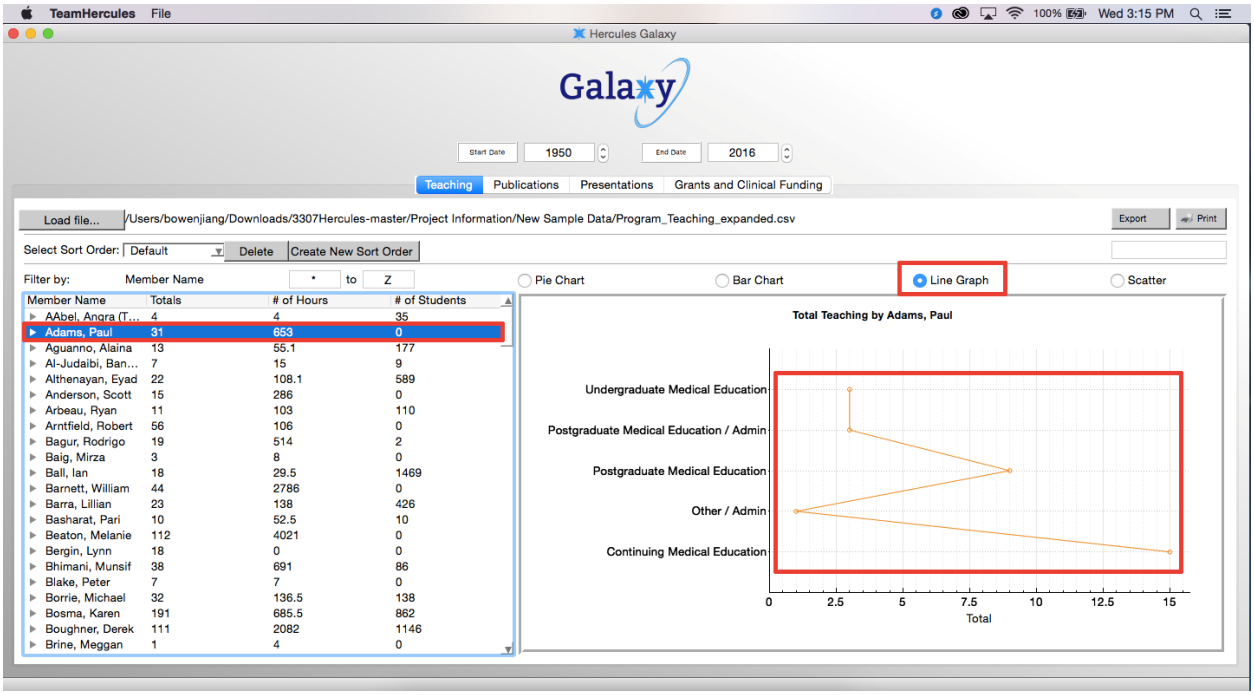
2.1 REQUIREMENTS IMPLEMENTED

2.1.1 Additional Graphs

For each subject area, the user can display a pie chart or bar graph of the record information by clicking on the member name within the dashboard view. This functionality will be expanded by adding two additional charts. These additional views were requested by the client.

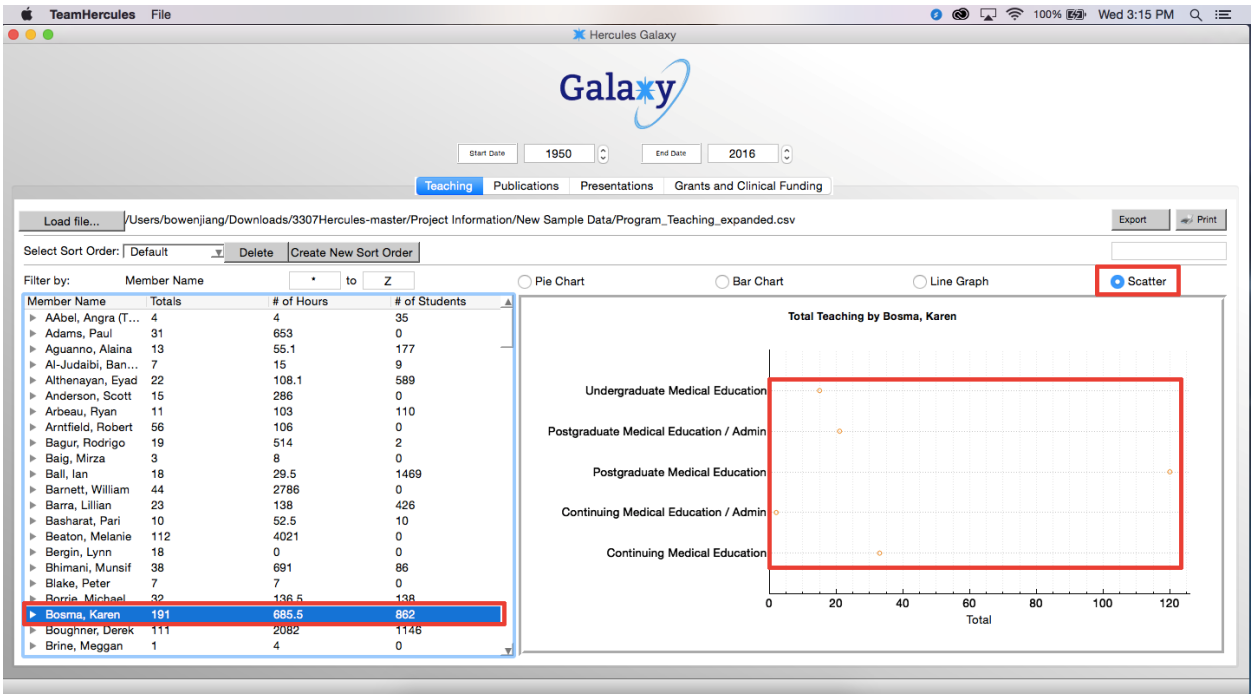
2.1.1.1 Line Graph

A line graph compares two values, each on a separate axis. Line graphs are useful as they allow the user to visualize trends more easily in the data, and are also effective in showing how variables affect each other. It is also easy to see specific values through a line chart and to estimate future outcomes. An improvement to the user experience would be to randomize the colors of the graph or have them randomly selected from a pre-approved list to improve the visuals and the interactivity of the graph. Additionally, the axes should be reversed to better reflect the data. Although it seems unorthodox, the axis were flipped such that the user would be able to read the data labels, and to keep the visuals consistent with the other three graph types.



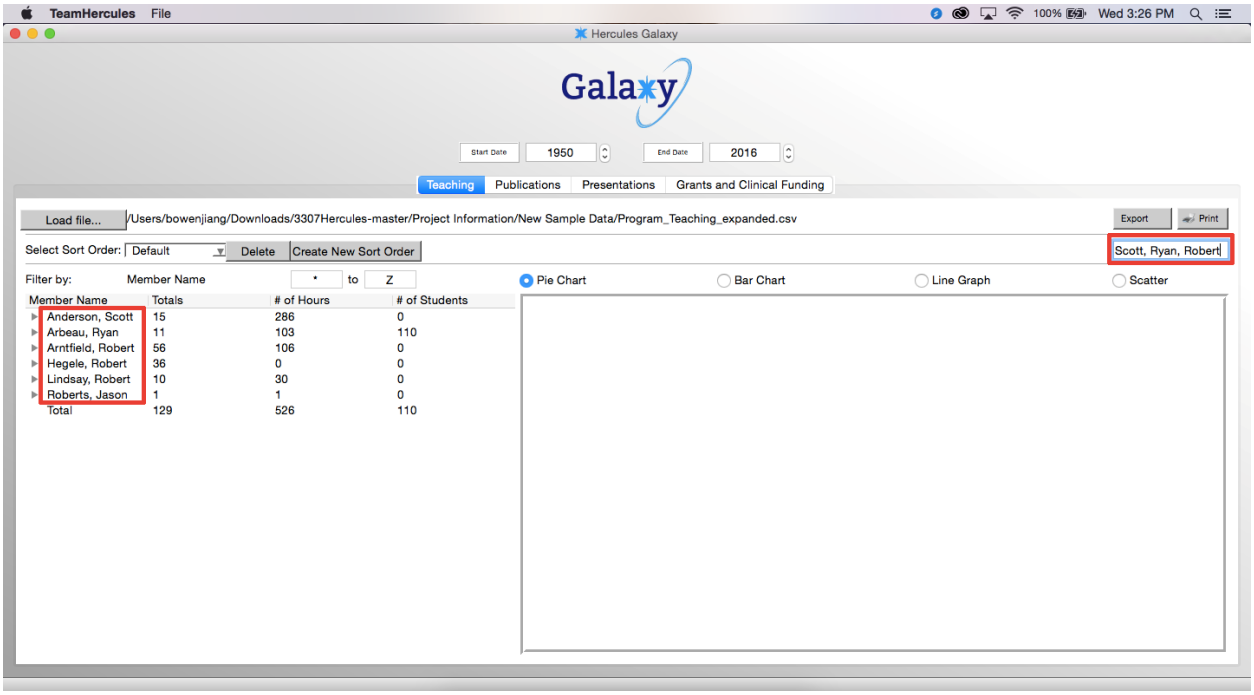
2.1.1.2 Scatter Plot

Scatter plots were implemented to allow the user to better abstract their data and visualize trends and patterns. The goal of the scatter plot was to allow the user to identify clusters, which they may use to divide the set to identify patterns within groups. This perspective would allow the user to identify which groups (departments) are performing similarly or dissimilarly to others providing context for a more in-depth analysis and comparison of these groups. In this analysis, the user can find particular behaviours or reasons which explain the similarities or dissimilarities, and use these findings to modify or create new policies or practices to improve the department. An improvement to the user experience would be to randomize the colors of the graph or have them randomly selected from a pre-approved list to improve the visuals and the interactivity of the graph. Additionally, the axes should be reversed to better reflect the data.



2.1.2 User Selected List

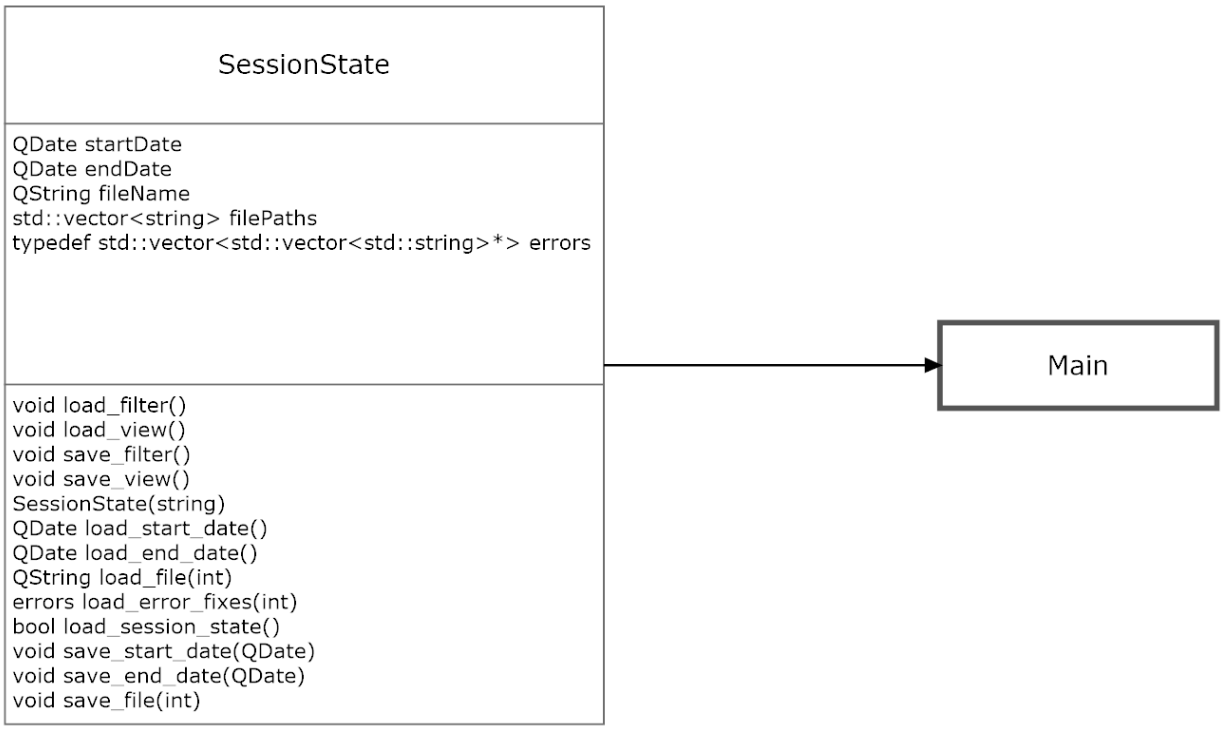
The client requested that the user of the system be able to filter the data by name. To accomplish this, we included an input field on the top right corner of the window, below the *Export* and *Print* buttons (as highlighted below). Here, the user is able to input a list of comma separated strings. MainWindow takes this list, and seperates them into a string vector. This string vector is passed into the RecordsManager class, through the appropriate TreeModel class, which applies a filter to the database, returning only strings that match any of the comma-delimited strings provided by the user (case-sensitive), in real time. This filter is applied explicitly to the main ID of each data element, which is typically their name, but when different "Sort Orders" are applied, it can be the department, division, or any other data presented in the first column (as highlighted below).



2.1.3 Session State

For this system, the client asked us to include session saving and as such we have implemented the ability to save each of the file paths that the user has provided for each of the different data categories. Here, so long as their paths remain unchanged, the application can retrieve the files each time it starts up. Likewise, the start and end date filter ranges were saved. This was all done in an effort to improve the user experience.

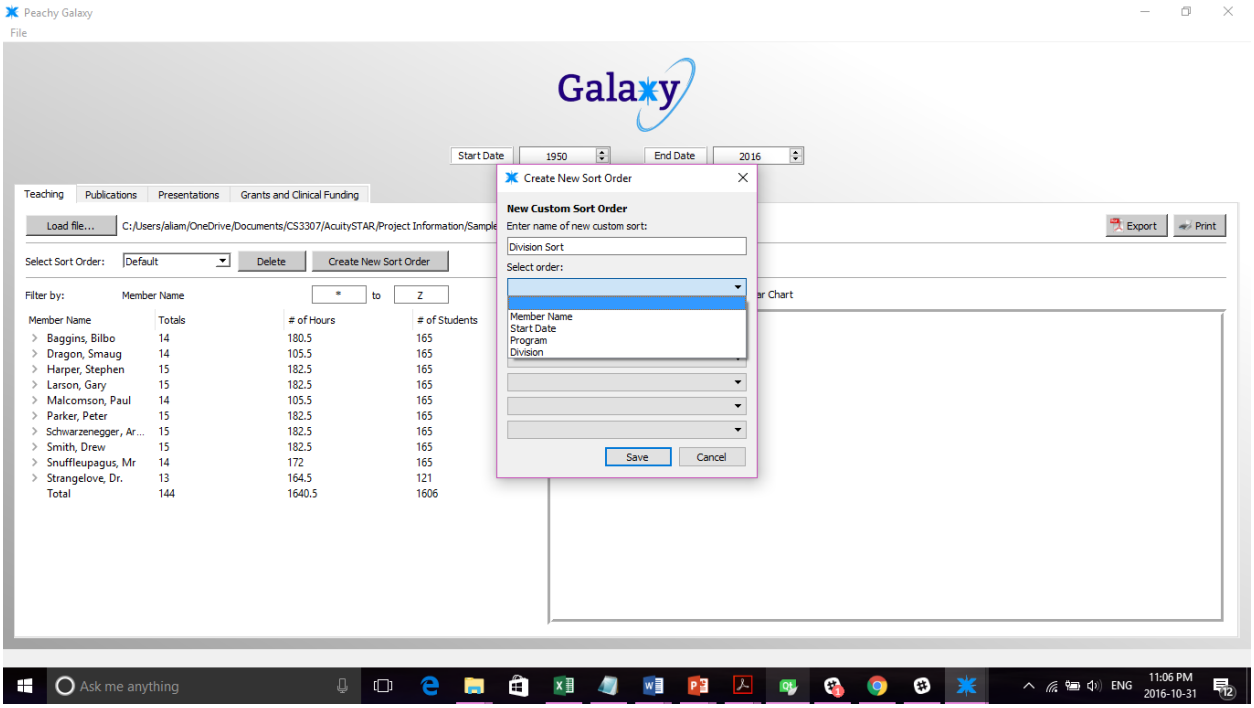
File IO is handled by Qt's serialization library, including QFile, QIODevice and QDataStream, for readability and maintainability purposes. The class interfaces solely with the MainWindow class to record and save each of the user's inputs. Future improvements will include the serialization of filters and current view states.

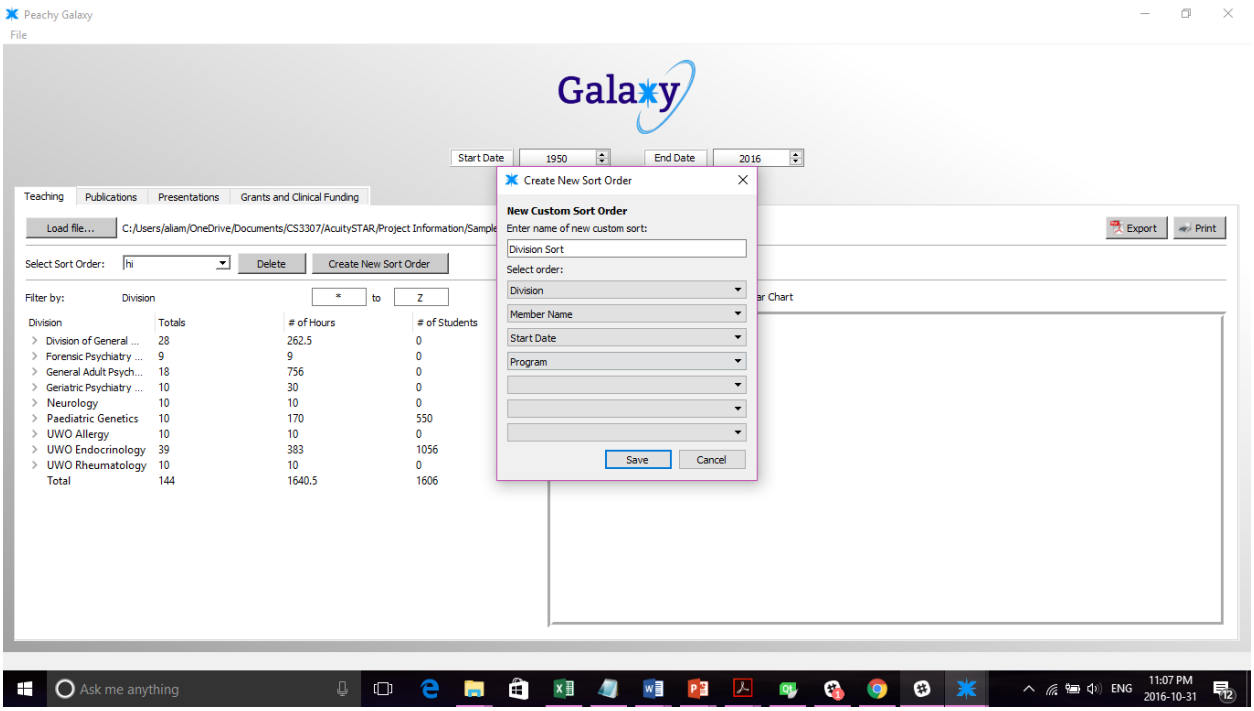


2.1.4 Ability to Sort by Member Division

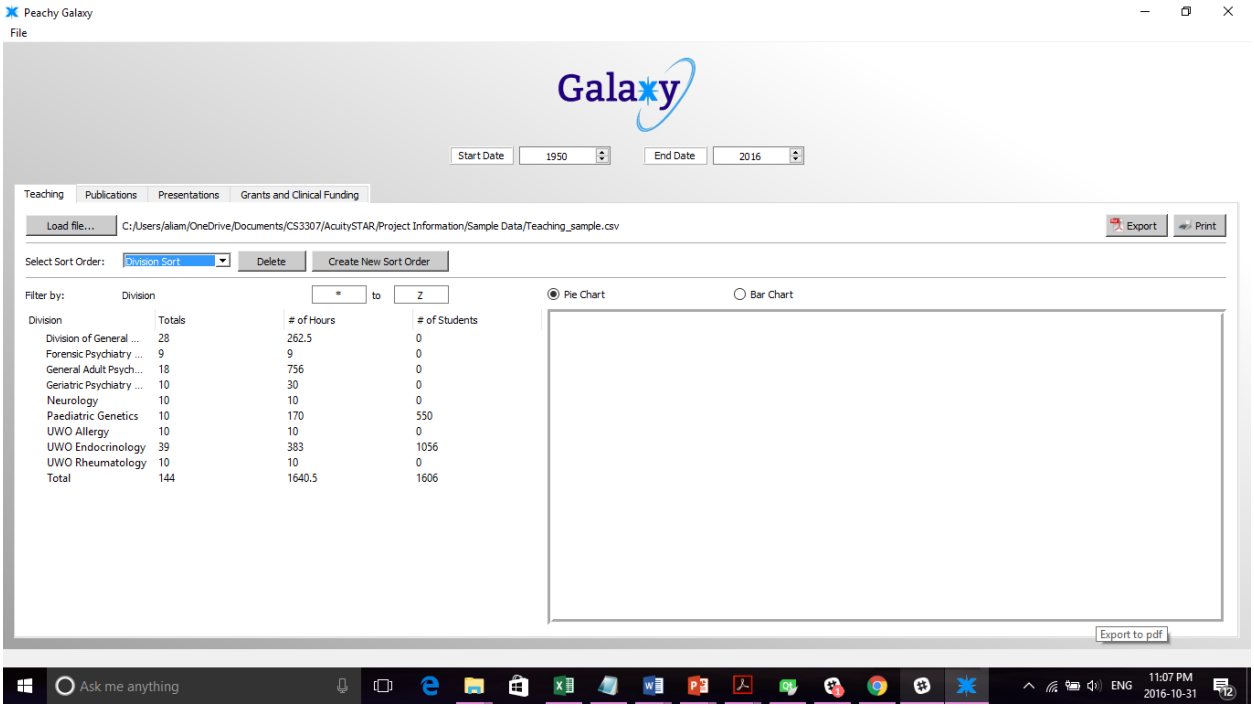
As part of this system, the user can now sort by “Member Division”. The user can create a new sort order under any of the available headings and sort by “Division”, as well as view data by “Division”. Furthermore, the new display also shows the Division distribution of users in the pie and bar graph view. Such a feature is integral when coupled with the new graph views as it gives the user a better perspective over their data.

Drop down menu includes “Division”

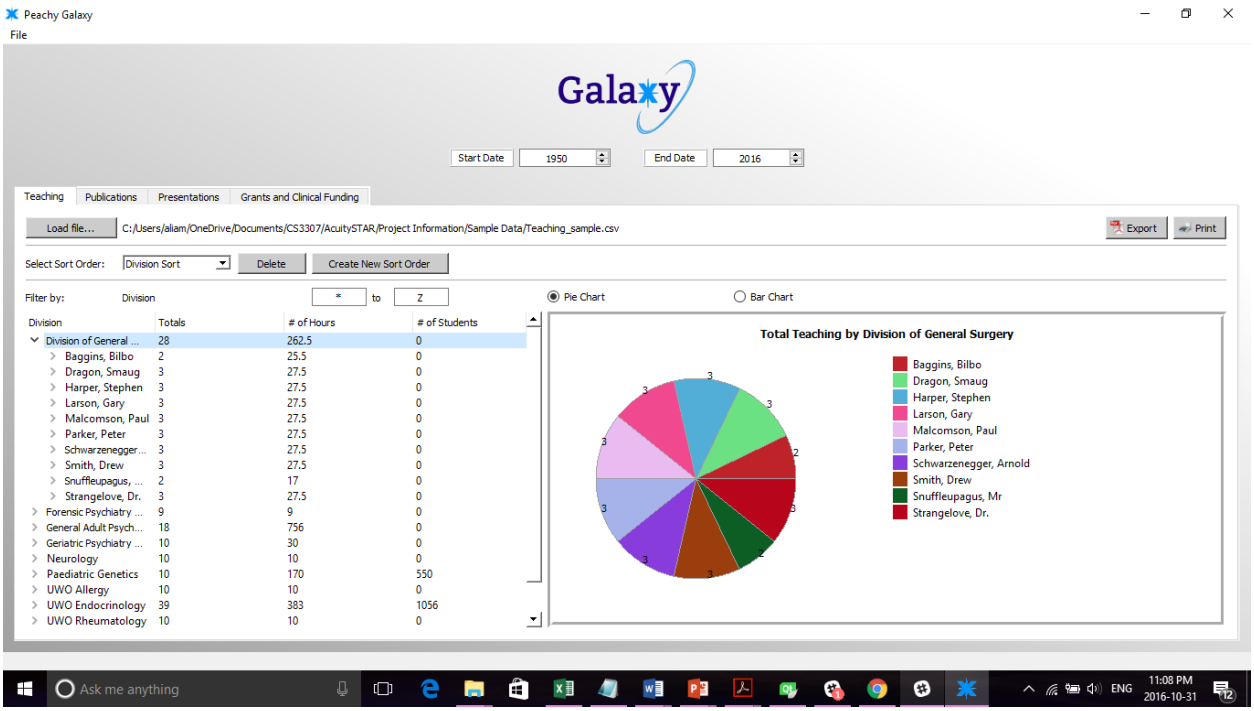




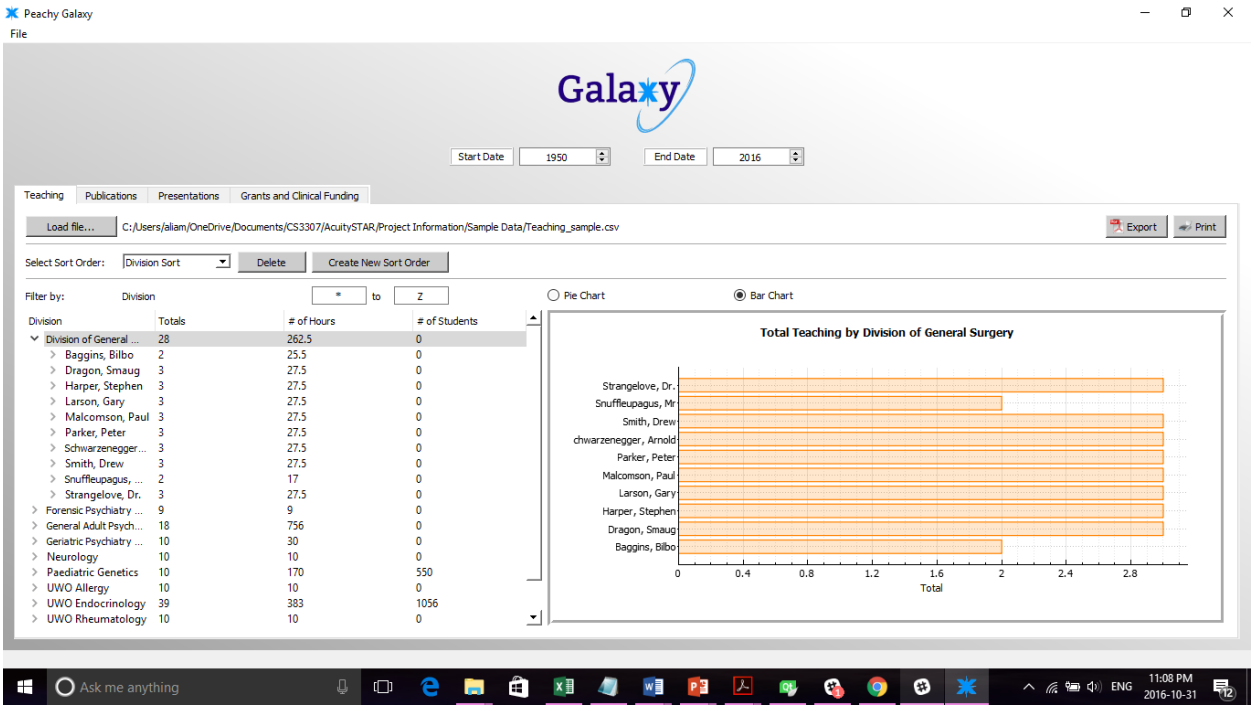
Display is now sorted by Division



Pie Graph that shows Division distribution



Bar Graph that shows Division distribution



2.1.5 Error Navigating

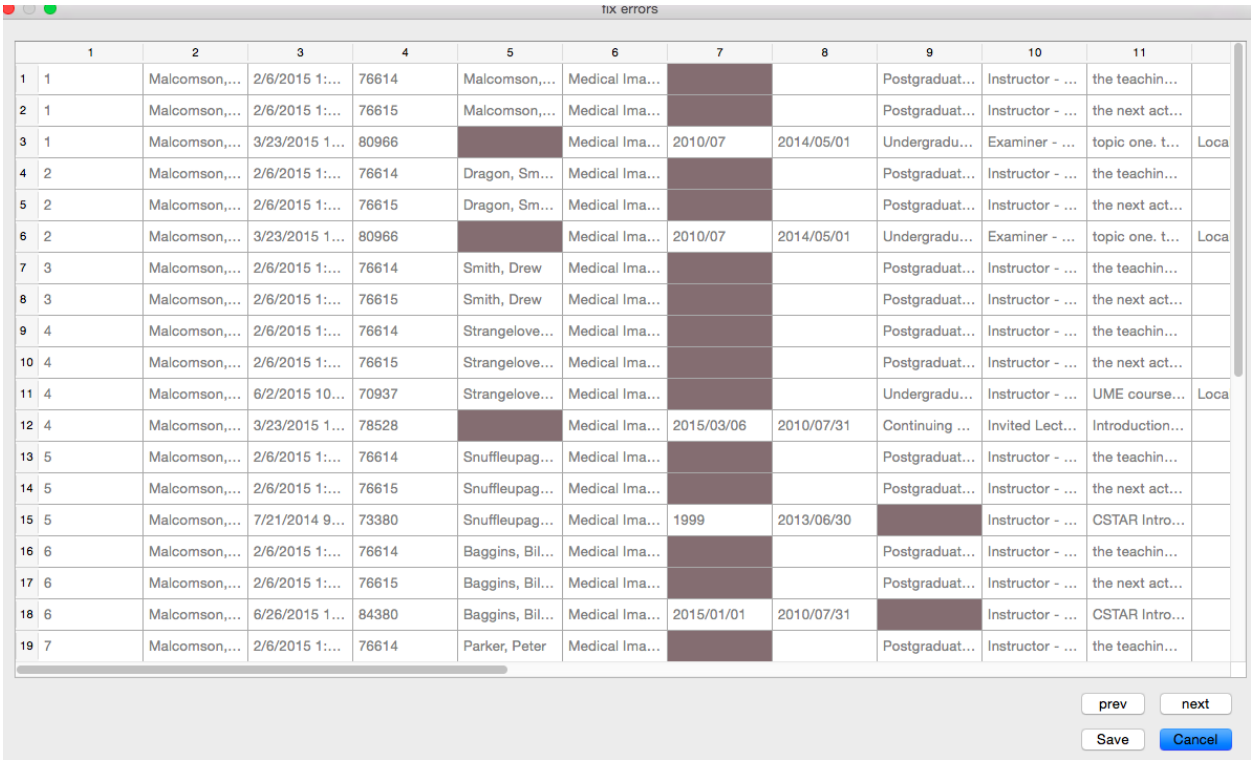
Previous and next buttons were implemented in the ErrorDialog class to allow the user to easily navigate through the errors present in the datasets. These buttons are meant to simplify the process of fixing through mistakes, which our team identified as one of the most significant problem points of using the application.

When the file is loaded, and the table is opened, there should be a Next/Prev button which will traverse through all the errors present in the table. This will help keep track of how many errors are still currently in the table and will also help the user see where the missing fields are located.

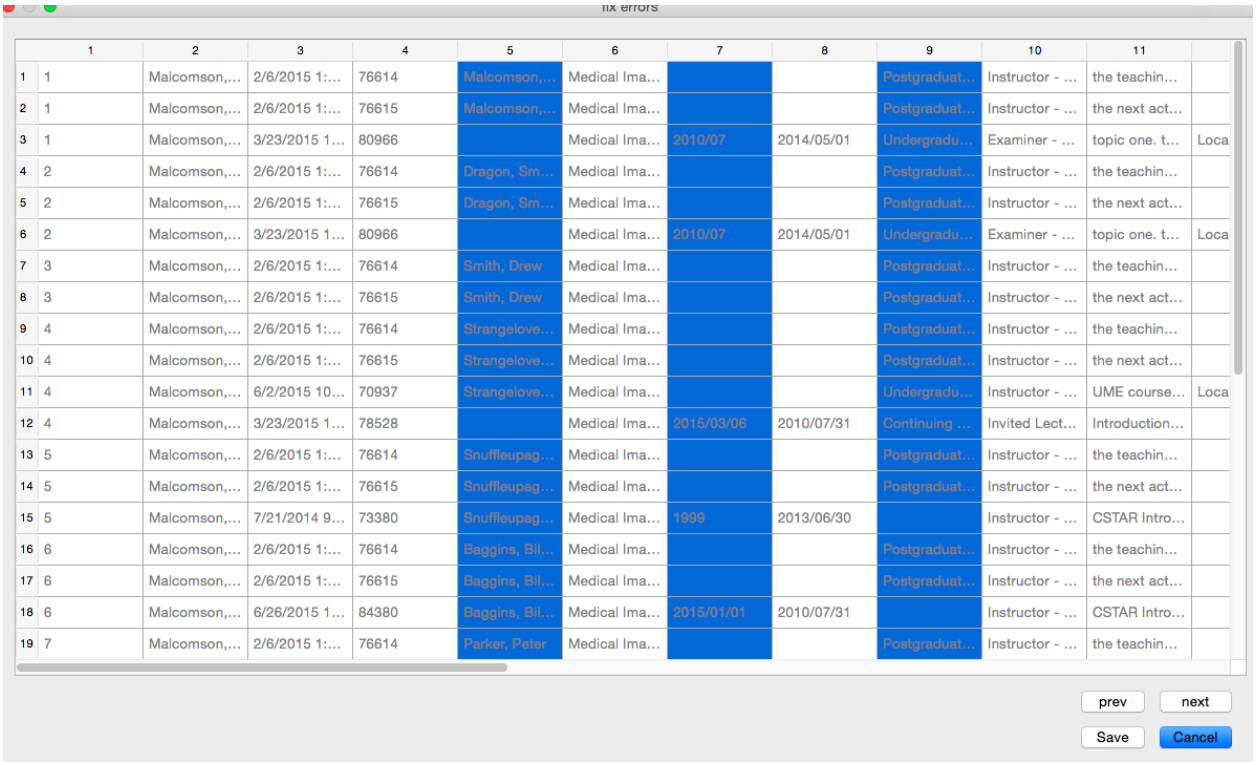
The origin comes from our group discussion of how we could make the errors easier to locate and read.

Supporting examples: First screenshot shows where the missing fields are located. The second screenshot shows the result after clicking on the next button, which highlights all the columns that contain those errors.

Graph showing the missing fields in the table:



Highlights the errors within each column in the table



	1	2	3	4	5	6	7	8	9	10	11	
1	1	Malcomson,...	2/6/2015 1:...	76614	Malcomson,...	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
2	1	Malcomson,...	2/6/2015 1:...	76615	Malcomson,...	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
3	1	Malcomson,...	3/23/2015 1...	80966		Medical Ima...	2010/07	2014/05/01	Undergradu...	Examiner - ...	topic one. t...	Loca
4	2	Malcomson,...	2/6/2015 1:...	76614	Dragon, Sm...	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
5	2	Malcomson,...	2/6/2015 1:...	76615	Dragon, Sm...	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
6	2	Malcomson,...	3/23/2015 1...	80966		Medical Ima...	2010/07	2014/05/01	Undergradu...	Examiner - ...	topic one. t...	Loca
7	3	Malcomson,...	2/6/2015 1:...	76614	Smith, Drew	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
8	3	Malcomson,...	2/6/2015 1:...	76615	Smith, Drew	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
9	4	Malcomson,...	2/6/2015 1:...	76614	Strangelove...	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
10	4	Malcomson,...	2/6/2015 1:...	76615	Strangelove...	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
11	4	Malcomson,...	6/2/2015 10...	70937	Strangelove...	Medical Ima...			Undergradu...	Instructor - ...	UME course...	Loca
12	4	Malcomson,...	3/23/2015 1...	78528		Medical Ima...	2015/03/06	2010/07/31	Continuing ...	Invited Lect...	Introduction...	
13	5	Malcomson,...	2/6/2015 1:...	76614	Snuffleupag...	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
14	5	Malcomson,...	2/6/2015 1:...	76615	Snuffleupag...	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
15	5	Malcomson,...	7/21/2014 9...	73380	Snuffleupag...	Medical Ima...	1999	2013/06/30		Instructor - ...	CSTAR Intro...	
16	6	Malcomson,...	2/6/2015 1:...	76614	Baggins, Bil...	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	
17	6	Malcomson,...	2/6/2015 1:...	76615	Baggins, Bil...	Medical Ima...			Postgraduat...	Instructor - ...	the next act...	
18	6	Malcomson,...	6/26/2015 1...	84380	Baggins, Bil...	Medical Ima...	2015/01/01	2010/07/31		Instructor - ...	CSTAR Intro...	
19	7	Malcomson,...	2/6/2015 1:...	76614	Parker, Peter	Medical Ima...			Postgraduat...	Instructor - ...	the teachin...	

2.2 STRETCH GOALS

2.2.1 Histogram

A histogram feature is planned to be implemented to illustrate the frequencies of the data to allow the user to better understand and diagnose the patterns within. Using this tool, the user would be able to identify the boundaries for outliers, and their likelihoods. They could use this data to identify the best and worst performers, and then follow-up with them to better understand what makes them exceptional. This data can then be used across the department to improve the performance and efficiencies within the department. An improvement to the user experience would be to randomize the colors of the graph or have them randomly selected from a pre-approved list to improve the visuals and the interactivity of the graph. Additionally, the axes should be reversed to better reflect the data.

2.2.2 Editable Text Fields

TextField is used to accept a line of text input. Input constraints can be placed on a TextField item. Editable text fields will be used to maintain acceptable input within each field. This takes place in the ErrorEditDialog class and window that pops up, where the user is prompted to edit present errors. Here, they are presented with the option of editing and inserting existing and missing data respectively, allowing the user to modify the data on an import. The client requested this feature, likely to add flexibility to the application, which we believe

2.2.2.1 Missing Fields

While debugging the Hercules Galaxy application, attempts were made to input missing fields into sample data after loading a file, and these attempts were successful. This can lead to a misrepresentation of data and should not be possible. This was fixed through the implementation of the Editable Text Fields feature.

2.2.3 Training Video

A training video will be provided for the user. It is an easy way to demonstrate how to use the application and navigate through Hercules Galaxy. This drastically lowers the learning curve for the user, providing a more approachable learning tool. The video's overview of the system's features, coupled with a closer analysis in the User Guide, should provide the user with the adequate tools to easily and effectively use Hercules Galaxy.

2.2.4 Code Deck for MAC Operating System functionality

To include Mac systems as users on this application, our team decided to prioritize delivering a dual-operating system product. This was done to avoid alienating the Mac OS users from using the system, and including them in the onboarding necessary when introducing a new system to employees. We believed that allowing the client to have fewer rounds of onboarding justified this prioritization, especially as it the client who requested this feature.

2.3 DEVELOPMENT PLANS

Along with the stretch requirements, these are additional improvements our team would have liked to implement, but lacked the time to do so.

2.3.1 Bar graphs and Pie charts are inconsistent

When records are single clicked, the corresponding pie or bar chart is not displayed in the window. The previously clicked graph is displayed. This occurrence is inconsistent; there are a few occurrences where the graph changes to the corresponding record. This functionality should be improved to ensure that data and charts are accurately displayed on either single or double click.

2.3.2 Chart Data is Not Displayed When Using a Different Sort Order

When attempting to display chart data after applying a sort order, no chart is displayed. This bug presents a missed opportunity to user the various charts to display summary data, and should be a priority fix. This is exclusive to the sort order, and has no effect on the User Selected List feature.

2.3.3 Input verification

When attempts to enter information into sample data succeeded, it was recognized that the entries were not verified. Characters and strings were entered into integer fields. This is a feature that should be implemented to preserve the integrity of the data.

2.3.4 Print preview functionality

When the user clicks the print button, there is no way to alter or visualize the information that will be printed until it is sent to the preferred printer. In other words, there is no way to tell what will be printed until it is already printed. A print preview function is recommended, where the user will be able to preview the representation of what will be printed.

2.4 AGENT TASK VIEW

	Member					
Tasks	**Amber Ali**	**Martin Kilonzo**	**Shanice Hanlon**	**Bowen Jiang**	**Michael Song**	**Michael Tassone**
Report	**x**	**x**	**x**			
Session State		**x**				
Graphs					**x**	**x**
Division Sorting	**x**		**x**			
Error Handling		**x**		**x**		
Training Video	**x**		**x**			
User Selected List		**x**			**x**	
Documentation	**x**	**x**	**x**			

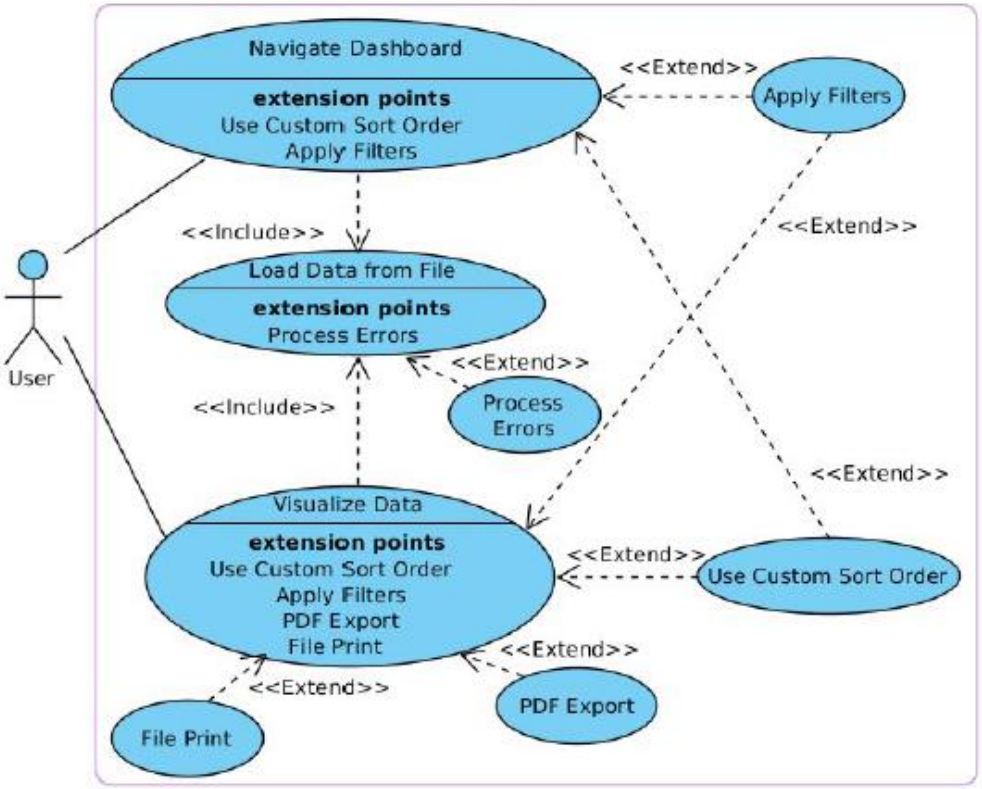
2.5 IV) TEST CASES:

See Test-case Matrix.xlsx

Our team created various test cases for both new and existing classes. Specifically, we have tested CSVReader, RecordsManager, the Graphs, and SessionState to test the various implicit features outlined above. Here test cases were described, with expected results implemented. In some cases, the expected results were correct, but the tests themselves failed, resulting in a false-negative, indicating a failure in the test case logic. These instances are outlined in the Test Case Matrix. Likewise, other failed cases are outlined in the Matrix.

3 DIAGRAMS

3.1 USE CASE DIAGRAM



The use-case diagram outlines how the user and the system communicate, and how the former interfaces with the system. It is a high-level abstraction of the system from the user's perspective. The user has two main uses cases, "Navigate Dashboard" and "Visualize Data" which each correspond to a customer requirement. "Navigate Dashboard" allows the user to navigate the program and "Visualize Data" allows the user to visualize the loaded data file with 4 different types of graphs. The user can also apply a custom sort order to each view.

3.1.1 Loading data from file

Load Data from File (Sea Level)

3.1.1.1 Main Success Scenario:

1. The user clicks on a subject area tab (default is Teaching).
2. The user clicks the Load button.
3. The system displays a file structure screen.
4. The user selects a CSV file and clicks the Open button. [Alternate Course A: File is not CSV type] [Alternate Course B: User clicks Cancel button]
5. The system verifies if the records contain any missing fields. [Extension Point: 3.1.2 Error processing]

6. The system loads the records.

3.1.1.1.1 Alternate Course A: File is of invalid type

1. The system displays an error message.
2. The user accepts or closes the error message.

3.1.1.1.2 Alternate Course B: User clicks Cancel button

1. The system closes the file structure screen.

3.1.2 Error processing

Process Errors (Sea Level)

3.1.2.1 Main Success Scenario:

1. The system displays message showing number of invalid records and prompts user to edit or discard them.
2. The user clicks Edit button. [Alternate Course A: User clicks Discard button]
3. The system displays an error processing screen.
4. The user fills in all missing entries and clicks the Save button. [Alternate Course B: All entries not filled out] [Alternate Course C: User clicks Cancel button]
5. The system includes the newly modified records in the data to be loaded
6. The system closes the error processing screen.

3.1.2.1.1 Alternate Course A: User clicks Discard button

1. The system discards records with missing mandatory entries from the data to be loaded.
2. The system closes the error processing screen.

3.1.2.1.2 Alternate Course B: All entries are not filled out

1. The system displays an error message.
2. The user accepts or closes the error message. [Return to Main Success Scenario step 4]

3.1.2.1.3 Alternate Course C: User clicks Cancel button

1. The system discards records with missing mandatory entries.
2. The system closes the error processing screen.

3.1.3 Custom Sort

Apply a customer sort filter to the selected data (Sea Level)

Main success Scenario:

1. The user loads at least one record successfully following **Scenario 3.1.1: Load Data from File**
2. The user selects a previously created custom sort order, and the currently loaded record gets filtered

3.1.3.1 Session State

Restore system state (Sea Level)

Main success Scenario:

3. The user loads at least one record successfully following **Scenario 3.1.1: Load Data from File**

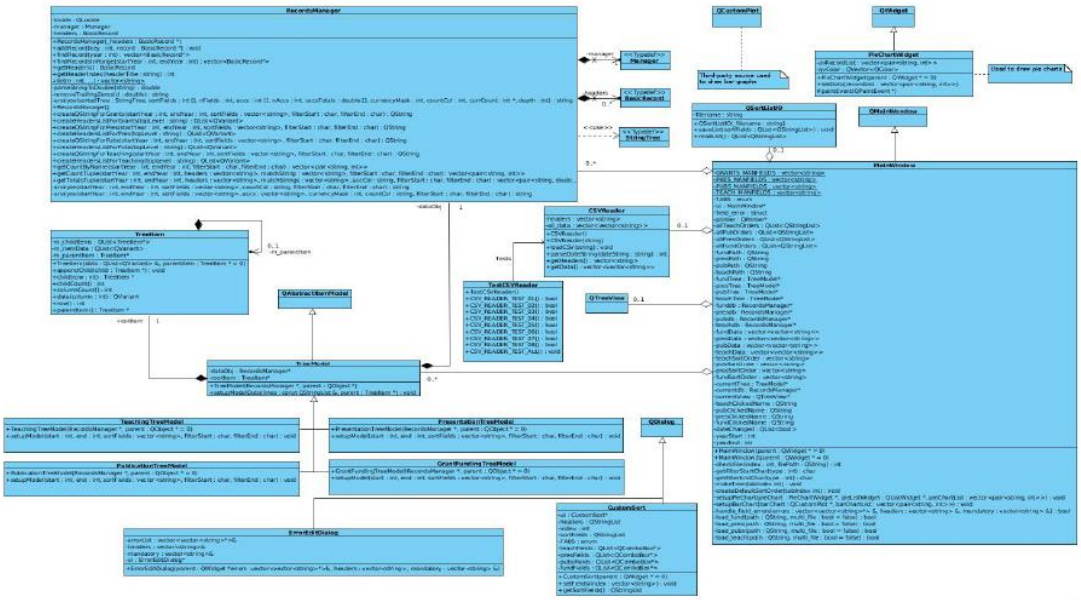
4. The user closes the program
5. The user re-opens the program, and the program automatically loads the records opened in the last session [Alternate Course A: No records were opened in the last section or The program failed to write to the session state file]

3.1.3.1.1 Alternate Course A:

1. The program opens with no files loaded

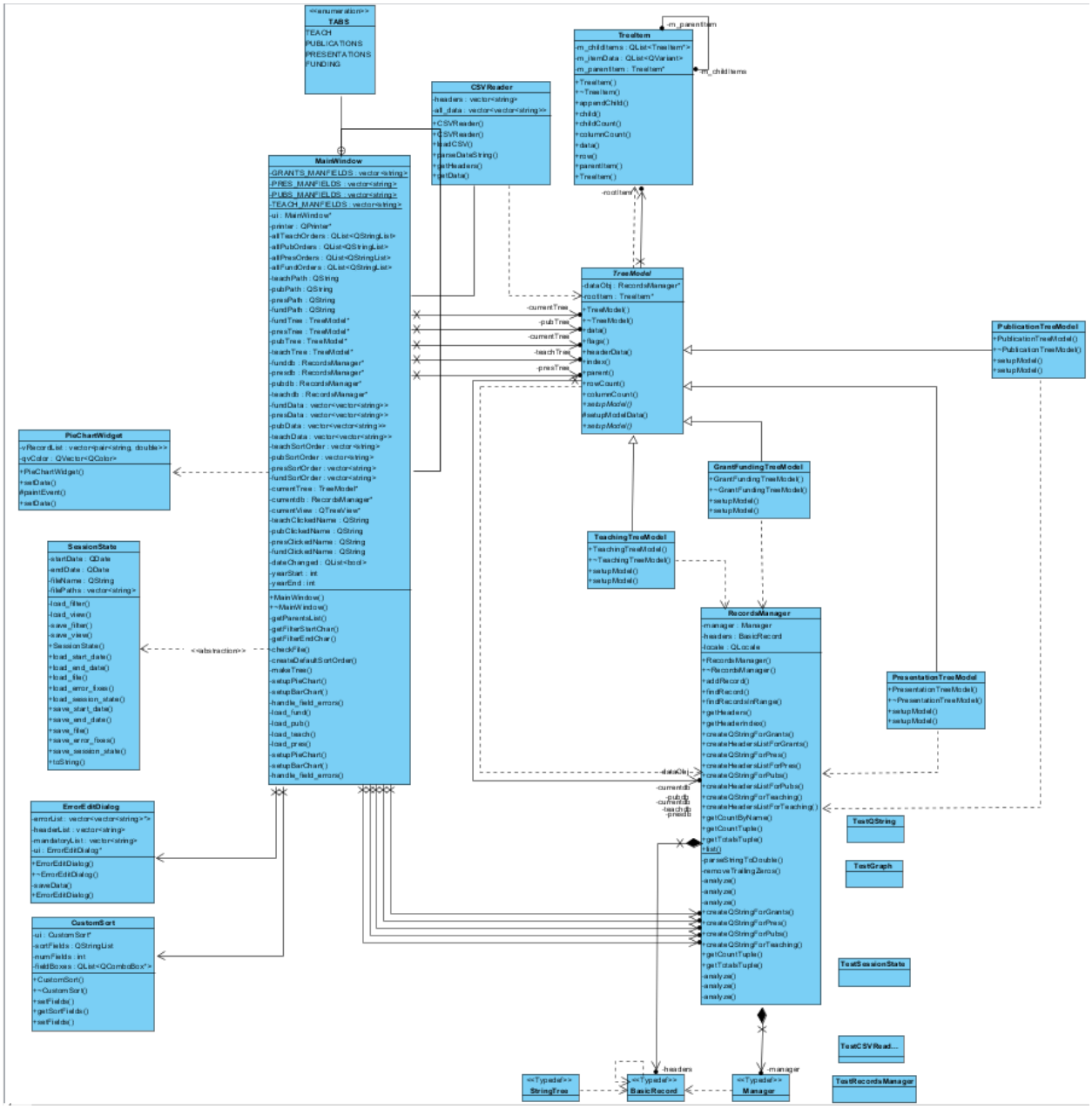
3.2 Class Diagram (UML)

Original Class diagram



This is the previous class diagram, detailing the types of system objects and their (inter)relationships. This model was modified to include the aforementioned features as optimally as possible, detailed, on the following page.

Enhanced Class diagram



This diagram outlines the relationships between classes and their structure, as well as the functions that are available amongst related classes. It also acts as an abstract overview of the programs design patterns. Much of this model was left intact over the previous design, with changes made as modifications to existing classes, or additions to the class structure.

3.2.1 CSVReader

CSVReader is called by MainWindow to load the comma separated value files. After doing so, CSVReader parses the data from the file, checking that the data contains the appropriate values to be loaded. Once this is complete, CSVReader passes the data back to MainWindow to forward it to RecordsManager.

3.2.2 RecordsManager

MainWindow calls RecordsManager to process the parsed data from the comma separated value file, delivered by CSVReader. Here, RecordsManager acts as the primary bridge between the front-end and the back-end, manipulating the data given the appropriate values passed into it from MainWindow to TreeModel, which then returns the data back to RecordsManager. These manipulations include sorting by date and by lexicographic-range, as well as by field. Further, these manipulations include filtering by the primary (first column) data field.

3.2.3 TreeModel

TreeModel acts as an abstract class which is implemented by classes which properly describe their respective data types: Teaching, Publications, Presentations, Grants and Funding as TeachingTreeModel, PublicationsTreeModel, PresentationTreeModel, and GrantFundingTreeModel, respectively. These classes take attributes passed to them by MainWindow, and pass them to RecordsManager, which applies those attributes, returning a data-set to the appropriate TreeModel, which builds the data structure to pass back into MainWindow to present to the user.

3.2.4 MainWindow

MainWIndow is a subclass of QMainWindow and is the central class that allows cross interaction between many classes. It also contains the user interface, which is the reason why this class is as central as it is—it takes in all of the user input and passes it to the appropriate classes. This UI is built using various QObjects, taken from the QT library for their easy implementation and reliability.

3.2.5 QSortListIO

QSortListIO is a class implemented to read and write the custom sort order data. It serializes the information which is then saved for later use.

3.2.6 SessionState

SessionState is a class implemented to read and write system state data, including the view, and filter data, and the files loaded. Upon loading, MainWindow attempts to load the serialized data from SessionState. Should the data exist, SessionState passes it into the MainWindow, which then loads the application as usual, prompting the user to fix any errors present, and modify the data. Should the data not exist, the system loads as normal, prompting the user for the required input to continue.

3.2.7 ErrorEditDialog

ErrorEditDialog is a subclass of QDialog, created whenever an error or missing data is found when attempting to load a file from MainWindow. This window has been modified to allow the user to input new data, modify erroneous data, and navigate between errors using the previous and next buttons to easily determine that need the most urgent attention.

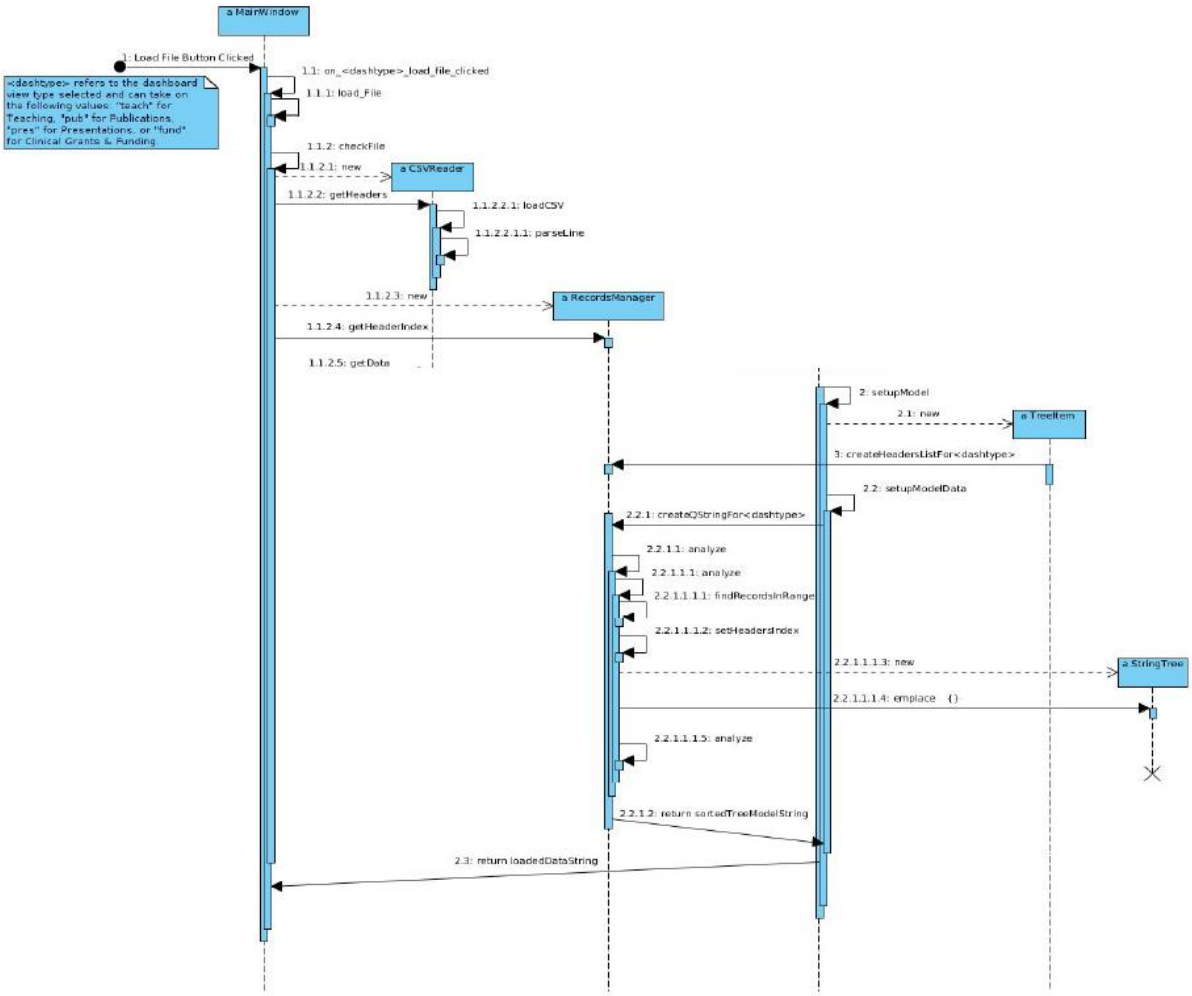
3.2.8 QCustomPlot

QCustomPlot is a third-party import used to plot bar, scatter, and line graphs for the various data-type views. As a stand-alone, third party library, built for QT, it has been well vetted and has excellent documentation. This class takes data piped to it from MainWindow, which then returns a graph diagram representative of the new data.

3.2.9 PieChartWidget

Similarly to QCustomPlot, PieChartWidget is an import, however, PieChartWidget is from a QT library class known as QWidget. This was implemented to visualize the data piped to it from MainWindow, where it returned a pie chart representative of that data.

3.3 SEQUENCE DIAGRAM



The sequence diagram illustrates the process flow of the system when it is provided with specific triggers. Each "vertical column" represents a key processes after a trigger, and the ensuing tasks that are performed to complete the task are illustrated on the horizontal axis. The sequence has not changed in a material way from the previous version of the application.

3.3.1 System Prompts for File Load

If this is either the user's first time loading a file for the specific tab, or the user wanted to change the file, the user would be required to specify which file to load. The user clicks on the *Load Data* button. MainWindow then calls its "loadFile" function, which brings up a file dialog window.At this point, the program prompts the user to select a valid comma separated value file containing the data to load into

the application, corresponding to one of the four data types: Teaching, Publications, Presentations, or Grants and Funding.

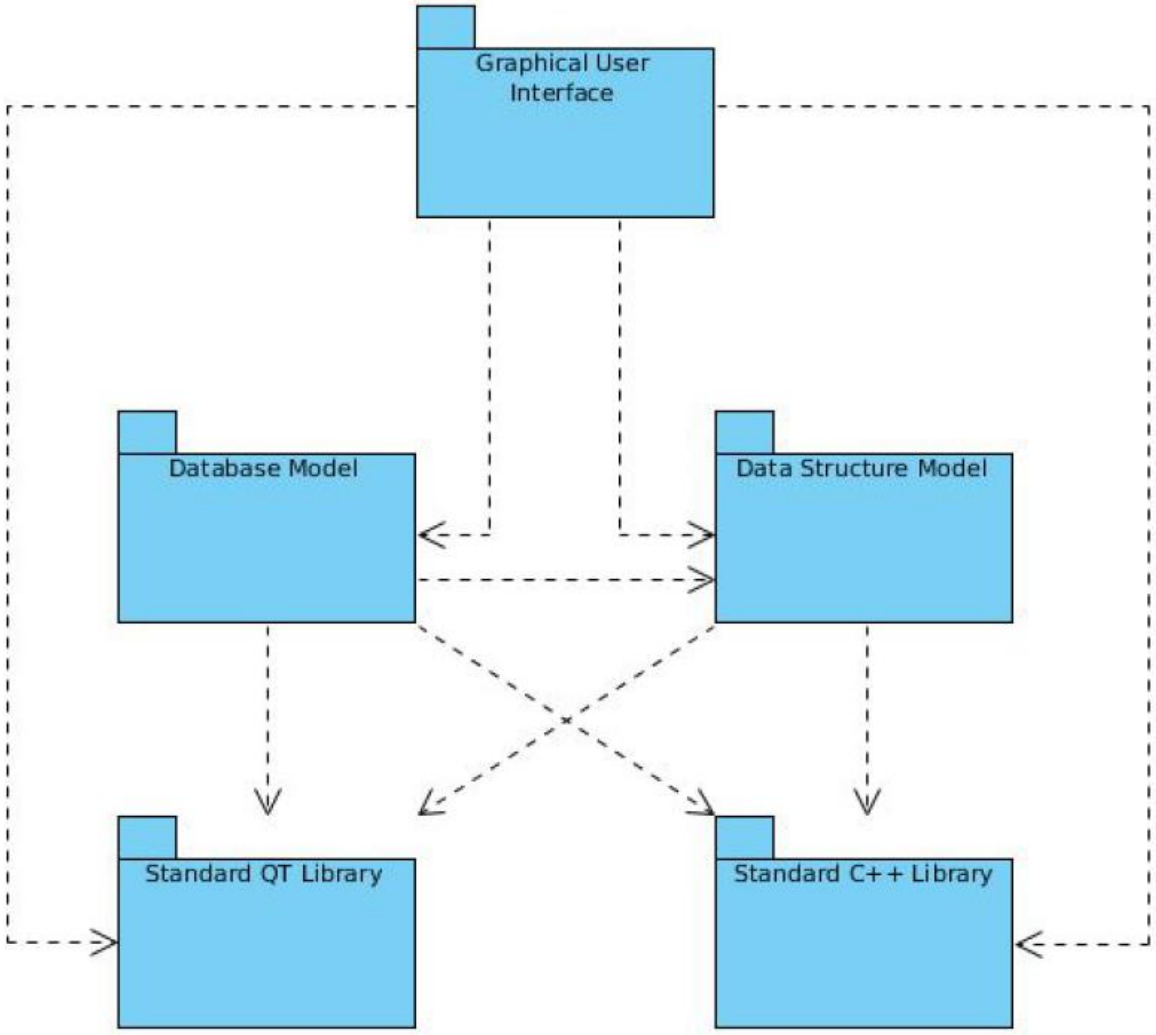
3.3.2 File Verfication

MainWindow takes the specified file path and determines if the file at the end of the path is appropriate and valid. In the event that it is not, the user is sent a warning dialog box. Otherwise, the file is sent to CSVReader to be parsed and to face further validation against the datatype view.

3.3.3 System Loads File

Once a file path was validated and sent to CSVReader, CSVReader is tasked with parsing the data from the file, producing headers, and data. Throughout this process, it can reject the file if it the data in the file does not match the specified data type. Once parsed, the data is passed through MainWindow to be passed into RecordsManager. Here, RecordsManager is responsible for sorting and filtering the data by using the appropriate TreeModel classes relevant to the datatype. To accomplish this, RecordsManager uses various analyze functions to specify which attributes in the data are to be sorted, kept, and/or filtered.

3.4 PACKAGE DIAGRAM



This diagram shows the package dependencies between the package model diagrams in the Galaxy application. Our team decided to keep the original package design as we acknowledged that the system functioned well in its current state, and would not benefit much from redesigning the package model. As none of the features implemented drastically redesigned the system, much remains the same.

3.4.1 Database Model

Here, CSVReader and RecordsManager use data sent through the various data structures in the Data Structure Model to load, parse, save, and retrieve the data. It is the responsibility of CSVReader to load and parse the comma separated value files. RecordsManager relies on the Database Models directly, which interface with the UI, to provide the appropriate data set for each of the specific views (Teaching, Publications, Presentations, and Grants and Clinical Funding). Each pipe specific fields from the user interface in MainWindow, to RecordsManager, which then sorts and filters the raw data for the user.

3.4.2 Data Structure Model

Using the TreeModel abstract class as their basis, various classes were modeled to specify the required data fields for RecordsManager. Here, TeachingTreeModel, PublicationTreeModel, PresentationTreeModel, and GrantFundingTreeModel are each defined to pipe the required data from MainWindow into RecrodsManager, respectively, such that the user can see relevant data. As an example, the user can provide a list of comma separated names in the top right input field. Then, the application will update the data to filter the primary key (typically the entries' name fields) by passing the string from the input field, through MainWindow, into the appropriate TreeModel class, through to RecordsManager, which will update the database to reflect the user's input. Given this verbosity, it contradicts the Acyclic Dependency Principle, although this was done in an isolated matter, mitigating the drawbacks this contradiction brings.

3.4.3 Standard QT Library

This application relies on the QT library for many functions, classes, objects, and dependencies. These are well vetted, well documented, and are considered reliable as dependencies.

3.4.4 Standard C++ Library

This application relies on the C++ library for many functions, classes, objects, and dependencies. These are well vetted, well documented, and are considered reliable as dependencies.

Design Patterns

In an effort to ensure the codebase is as maintainable as possible, many of the existing design patterns have been maintained.

4 SINGLETON

The Singleton approach was used for classes that are intended to run passively, or are needed on demand. Here, access to each sole instance is encapsulated such that it controls the user's interaction with the class. In doing so, the Singleton classes are restricted in number of instances, which is convenient for single-instance classes, such as MainWindow. If restricting the

Typically, such classes are difficult to delete, yet, in practice, because these are monolithic classes, they are meant to be deconstructed infrequently, negating this drawback. Alternatively, the Singleton approach allows for easier and more consistent modifications. Below are the various reasons this model was selected:

- Controlled access to sole instance; The Singleton model encapsulates its sole instance, giving it complete control over how, where, and when the client can access it.
- Reduced name space: The Singleton pattern clears up the global variable pollution that would otherwise store sole instances.
- Permits refinement of operations and representation: the Singleton approach to classes allows for subclasses, with simple configuration within the instance of the extended class. This can even be accomplished at run-time.
- Permits a variable number of instances: The pattern both constricts the number of possible instances, but also allows this value to be easily modified should the design require that more (or less) instances of the Singleton class be instantiated.
- More flexible than class operations: Given its enhanced scope, the Singleton model allows its Singleton classes greater functionality than typical class operations.

4.1 C++ IMPLEMENTATION

Although this model was upheld to ensure consistency with the existing codebase, it has its flaws in that it is cumbersome to expand. For example, each of the graphs are defined within the MainWindow.cpp Singleton. This meant that adding additional graphs had to be done within the existent Singleton. This is exacerbated by the fact that each tab implements its own unique Singleton of the graph. Alternatively, had a Prototype-or-similar model been pursued more consistently within the Singleton, it would increase its modularity and reduce code-duplication. As such, in the future, this code is expected to be refactored to reflect the Prototype model to gain these benefits.

Further, the SessionState feature maintains a hybrid, Singleton-Prototype model, where it exists within the MainWindow class, but is also broken down into subclasses. In so doing, the SessionState is able to use many of the benefits of the Singleton approach--such as limited instances, and greater scope--with the added feature of being more modular and easier to expand, borrowing from the Prototype model.

5 PROTOTYPE

The Prototype pattern allows for consistent implementation of template classes, whilst allowing sufficient customization for classes needed for more unique roles. Moreover, the Prototype model allows the client to quickly and easily plug and unplug classes as they are needed and unneeded respectively. This is done as classes are recursively defined by their components, where additional levels of recursive definitions allows for more adaptability and flexibility. Because this is the case, classes which implement Prototypes only interact with the main class, hiding the abstractions from the client. Below are some specific benefits of the Prototype pattern:

- Specifying new objects by varying values: Very dynamic systems allow for the definition of new functionality by providing object composition. Here, values are specified for an object's variables. This is in an effort to reduce the number of classes by avoided the redundant definition of classes. This is done by allowing the user to define new classes as inheritents of a base class. This has two strong features: it allows for the instantiation of classes with minimal programming; and it greatly reduces the number of classes a system requires.
- Specifying new objects by varying structure: In contrast to instantiating many individual, complex, laborious structures, the Prototype pattern defines elements by their components, improving the code's modularity and reducing its redundancies.
- Adding and remove products at run-time: the Prototype model is flexible enough to allow a system to easily integrate new classes into a system by registering prototypical inheritance with the client.
- Configuring an application with classes dynamically: In an environment where classes are loaded dynamically, the Prototype model facilitates this process by reducing the number of necessary classes to load.

However, this model does come with limitations: each subclass must implement a clone function, which may be difficult depending on the implementation. This can be exacerbated if the implementation includes circular references, or the internals cannot be copied themselves.

5.1 C++ IMPLEMENTATION

As mentioned before, this model was retained in the definition of the SessionState class. Here, the model allows the SessionState class to be defined by subclasses, such as the FileState class. This allows the SessionState to be easily modified, which is especially important in a program that is still in development, especially as the user requires a different feature set. Likewise, when adding the necessary features for navigating the error fields in the ErrorDialog class, this expandability was leveraged to easily implement the necessary changes. Had the ErrorDialog class been implemented differently, it might have been more difficult to expand.

6 FAÇADE

A façade design pattern allows the client to have an interface to access the system, without seeing the system complexities. A façade object is one single, simplified interface for all the rest of the subsystem. When you divide a system into multiple subsystems, you need to ensure you can minimize the dependencies between subsystems. Usually, a single wrapper class contains a set of members that can access the system for the façade client. It is useful for complex or large systems. The pattern is very easy to create, and has no additional cost. It is also easy to maintain and remove if the system requires it.

Below are some additional benefits of having a façade design pattern:

- Having convenient methods for common items. This makes software libraries easier to use, test and makes them more easily readable.
- There is more flexibility when creating the system, as interface implementations are not in multiple classes.

6.1 C++ IMPLEMENTATION

Although C++ does not have an interface class like java, the façade model is still applicable in this case. For example, the MainWindow class contains most of the UI elements of the system. Through this, modifications to the system can be made through this class, without having to go through any specific system details. Modifications to user interface, icons, layout, loading features etc. can be made through MainWindow, and having to modify other classes can be avoided.

7 C++ IMPLEMENTATION

7.1 SINGLETON

Although this model was upheld to ensure consistency with the existing codebase, it has its flaws in that it is cumbersome to expand. For example, each of the graphs are defined within the MainWindow.cpp Singleton. This meant that adding additional graphs had to be done within the existent Singleton. This is exacerbated by the fact that each tab implements its own unique Singleton of the graph. Alternatively, had a Prototype-or-similar model been pursued more consistently within the Singleton, it would increase its modularity and reduce code-duplication. As such, in the future, this code is expected to be refactored to reflect the Prototype model to gain these benefits.

Further, the SessionState feature maintains a hybrid, Singleton-Prototype model, where it exists within the MainWindow class, but is also broken down into subclasses. In so doing, the SessionState is able to use many of the benefits of the Singleton approach--such as limited instances, and greater scope--with the added feature of being more modular and easier to expand, borrowing from the Prototype model.

7.2 PROTOTYPE

As mentioned before, this model was retained in the definition of the SessionState class. Here, the model allows the SessionState class to be defined by subclasses, such as the FileState class. This allows the SessionState to be easily modified, which is especially important in a program that is still in development, especially as the user requires a different feature set. Likewise, when adding the necessary features for navigating the error fields in the ErrorDialog class, this expandability was leveraged to easily implement the necessary changes. Had the ErrorDialog class been implemented differently, it might have been more difficult to expand.

7.3 FAÇADE

Although C++ does not have an interface class like java, the façade model is still applicable in this case. For example, the MainWindow class contains most of the UI elements of the system. Through this, modifications to the system can be made through this class, without having to go through any specific system details. Modifications to user interface, icons, layout, loading features etc. can be made through MainWindow, and having to modify other classes can be avoided.

Revised Timeline

See next three pages

First Iteration

LEGEND	
(Projected) Start	●
(Projected) Complete	★
Start & Complete Same Week	▲
Deliverable Due in this Week	#*

PEOPLE	
A	Amber Ali
B	Bowen Jiang
M	Martin Kilonzo
MT	Michael Tassone
MS	Michael Song
S	Shanice Hanlon

Stage 1

Task (Milestones)	Person	Due Date	Due Time	October 1	October 2	October 3*	October 4	November 1	November 2	November 3*	November 4	December 1	December 2	December 3	December 4*
Documentation and Code Base Understood		18-Oct	11:55 PM												
Loading Files	M			▲											
Parsing Data	M, B			●	★										
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			●	★										
Error Processing	MT			●	★										
Test Cases															
Loading Files	M			▲	▲										
Parsing Data	M, B														
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A				▲										
Error Processing	MT				▲										
Improvement and Enhancement Identification															
Loading Files	M			▲											
Parsing Data	M, B			▲											
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			▲											
Error Processing	MT			▲											
Development Infrastructure				▲											
Product Design				▲											

Stage 2

Task (Milestones)	Person	Due Date	Due Time	September 1	September 2	September 3*	September 4	October 1	October 2	October 3*	October 4	November 1	November 2	November 3	November 4*
		31-Oct	11:55 PM												
Error Handling Buttons	B									●	★	★			
Custom Sorting	A, S									●	★	★			
Additional Graphs	MT, MS, M									●	★	★			
Save Session State	M									●	★				
Stage 1 Test Cases															
Loading Files	M									▲					
Parsing Data	M, B									▲					
Sorting & Filters	MS, S									▲					
Graphs	MT									▲					
GUI	A									▲					
Error Processing	MT									▲					
Project Planning	MT									▲	▲	▲			

Stage 3

Task (Milestones)	Person	Due Date	Due Time	September 1	September 2	September 3*	September 4	October 1	October 2	October 3*	October 4	November 1	November 2	November 3	November 4*
		23-Nov	11:05 AM												
User Selected List	A, S											●		★	
Improvements on Stage 2												●		★	
Error Handling Buttons	B											●		★	
Custom Sorting	MS											●		★	
Additional Graphs	MT												●	★	
Save Error Session State	M											●		★	
Training Video	All													▲	
Revamping Filter System	M, MS												▲		
Fixing Graph Bugs	MT, MS										●			★	
Print Preview	B												▲		
Error Handling Verification	B												▲		
Fixing .csv Crash Bug	A, S													▲	
Fixing Failed Tests	All										●			★	
Create Executable	M											▲			
Redesigning Graphs	B												▲		

Second Iteration

LEGEND

(Projected) Start	●
(Projected) Complete	★
Start & Complete Same Week	▲
Deliverable Due in this Week	#*

PEOPLE

A	Amber Ali
B	Bowen Jiang
M	Martin Kilonzo
MT	Michael Tassone
MS	Michael Song
S	Shanice Hanlon

Stage 1

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**September** 1	2	3*	4	**October** 1	2	3*	4	**November** 1	2	3	4*
Documentation and Code Base Understood		18-Oct	11:55 PM												
Loading Files	M			▲											
Parsing Data	M, B			●	★										
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			●	★										
Error Processing	MT			●	★										
Test Cases															
Loading Files	M			▲	▲										
Parsing Data	M, B														
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A				▲										
Error Processing	MT				▲										
Improvement and Enhancement Identification															
Loading Files	M			▲											
Parsing Data	M, B			▲											
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			▲											
Error Processing	MT			▲											
Development Infrastructure				▲											
Product Design				▲											

Stage 2

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**October** 1	2	3*	4	**November** 1	2	3*	4	**December** 1	2	3	4*
		31-Oct	11:55 PM												
Error Handling Buttons	B									●	★				
Custom Sorting	A, S										●	★			
Additional Graphs	MT, MS, M									●	★				
Save Session State	M									●	★				
Stage 1 Test Cases															
Loading Files	M											▲			
Parsing Data	M, B											▲			
Sorting & Filters	MS, S											▲			
Graphs	MT											▲			
GUI	A											▲			
Error Processing	MT											▲			
Project Planning	MT									▲	▲	▲			

Stage 3

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**October** 1	2	3*	4	**November** 1	2	3*	4	**December** 1	2	3	4*
		23-Nov	11:05 AM												
User Selected List	A, S											●		★	
Improvements on Stage 2															
Error Handling Buttons	B											●		★	
Custom Sorting	MS											●		★	
Additional Graphs	MT												●	★	
Save Error Session State	M											●		★	
Training Video	All													▲	
Revamping Filter System	M, MS												▲		
Fixing Graph Bugs	MT, MS														
Print Preview	B												▲		
Error Handling Verification	B												▲		
Fixing .csv Crash Bug	A, S													▲	
Fixing Failed Tests	All										●			★	
Create Executable	M										▲				
Redesigning Graphs	B											▲			

Final Iteration

LEGEND

(Projected) Start	●
(Projected) Complete	★
Start & Complete Same Week	▲
Deliverable Due in this Week	#*

PEOPLE

A	Amber Ali
B	Bowen Jiang
M	Martin Kilonzo
MT	Michael Tassone
MS	Michael Song
S	Shanice Hanlon

Stage 1

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**September** 1	2	3*	4	**October** 1	2	3*	4	**November** 1	2	3	4*
Documentation and Code Base Understood		18-Oct	11:55 PM												
Loading Files	M			▲											
Parsing Data	M, B			●	★										
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			●	★										
Error Processing	MT			●	★										
Test Cases															
Loading Files	M			▲	▲										
Parsing Data	M, B														
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A				▲										
Error Processing	MT				▲										
Improvement and Enhancement Identification															
Loading Files	M			▲											
Parsing Data	M, B			▲											
Sorting & Filters	MS, S			▲											
Graphs	MT			▲											
GUI	A			▲											
Error Processing	MT			▲											
Development Infrastructure				▲											
Product Design				▲											

Stage 2

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**October** 1	2	3*	4	**November** 1	2	3*	4	**December** 1	2	3	4*
		31-Oct	11:55 PM												
Error Handling Buttons	B									●	★				
Custom Sorting	A, S										●	★			
Additional Graphs	MT, MS, M									●	★				
Save Session State	M									●	★				
Stage 1 Test Cases															
Loading Files	M											▲			
Parsing Data	M, B											▲			
Sorting & Filters	MS, S											▲			
Graphs	MT											▲			
GUI	A											▲			
Error Processing	MT											▲			
Project Planning	MT									▲	▲	▲			
Program Documentation	M												▲		

Final Stage

Task (Milestones)	**Person**	**Due Date**	**Due Time**	**October** 1	2	3*	4	**November** 1	2	3*	4	**December** 1	2	3	4*
		23-Nov	11:05 AM												
User Selected List	A, S											●	★		
Improvements on Stage 2															
Error Handling Buttons	B											●	★		
Custom Sorting	MS											●	★		
Additional Graphs	MT											●	★		
Save Error Session State	M														
Training Video	All												▲		
Revamping Filter System	M, MS												▲		
Fixing Graph Bugs	MT, MS														
Print Preview	B												▲		
Error Handling Verification	B												▲		
Fixing .csv Crash Bug	A, S												▲		
Fixing Failed Tests	All									●	★				
Create Executable	M										▲				
Redesigning Graphs	B												▲		
Documentation															
User Documentation	A, S												▲		
Windows Install Documentation	M												▲		
Mac Install Documentation	MT												▲		
System Inspections	B, MT, MS, S												▲		

8 LESSONS LEARNED & RETROSPECTIVE ANALYSIS

8.1 LESSONS LEARNED

There were many lessons learnt throughout this project. Understanding the code with limited documentation and comments presented a challenge. It was tough to understand what parts of the code were created through Qt libraries, rather than having been implemented from scratch. Additionally, it was difficult to follow what functions were defined in Qt, where variables were declared and initialized in the code, and which classes variables were from. It was also not clear as to where information was stored in the program, and how to retrieve data from the dataModel classes. The lack of familiarity with Qt and C++ was challenging. There is limited documentation and examples related with Qt online, and it was tough to implement our changes using the inherent system design used by the previous group, without a clear understanding of how they used Qt. Understanding which libraries would be the best for the modifications we intended for the system, and learning about using signals and slots effectively was also challenging.

Moreover, testing the original system was also challenging. Without a strong foundation in Qt, it was tough to analyze the code, and use Qt's testing infrastructure to create test cases for the existing code base. Understanding how to test features using signals and slots was also difficult. It was challenging because it was the first time we interacted with Qt, and used Qt's testing model.

Another hurdle was how to edit the graphical user interface, yet allow the same familiarity and functionality from the original application. Combining our modifications to the system, and having them fit into the design of the original system presented a challenge. It was decided that the original user interface worked for the intended use of the application, and didn't need major modifications. We decided to combine our modifications with the original design, so that the system functioned similarly to the original design.

As a group, we would have looked more closely at the code before trying to implement changes. We would have used more resources (such as contacting the TA's) to understand Qt. Furthermore, we would have looked more at implementing Qt libraries, rather than modifying code through objects.

8.2 RETROSPECTIVE ANALYSIS

As a team, we worked well and had defined roles and goals. Management was organized, and our team leader frequently contacted the instructor and TA's to clarify requirements and project goals. We had a schedule implemented, and the team frequently communicated and evaluated progress towards each deliverable.

Everyone had assigned tasks, and contributed towards the project. As a team, we could have met more in-person, which would have been helpful with code-related issues, and communication in general. By meeting more as a team, we could have brainstormed solutions or helped other members of the team on their assigned tasks. It would have also been useful to see everyone's progress on their specific assigned task. Furthermore, we could have worked better around everyone's schedule. There were multiple members of the team who had jobs and outside commitments, and it was hard for them to

come to campus on a daily basis. In hindsight, we should have set up meeting locations that were more convenient for everyone.

8.2.1 Value of this Project

- Learning Qt testing
- Learning about Qt libraries
- Learning to work with an existing code base, and modifying someone else's code
- Learning C++
- Continued learning to work in a group dynamic
- Learning how to schedule big projects and deliverables
- Understanding how industry work deals with deadlines
- Understanding how to design for specific clients and their needs
- Learning how to create a tailored application
- Learning about signals and slots
- Learning how to create different Use Case, UML, Sequence and Package diagrams
- Learning how to take instruction
- Learning how to take constructive criticism from other members of the team
- Collaborating on ideas with other group members
- Learning about what design patterns, and how to utilize them, and what situations they should be implemented

System Inspections

See next pages

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)
  ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question: (i) your analysis, and (ii) your finding (in the form of a comment). The analysis would typically show how you arrived at the finding.
- Add new lines as necessary for your analysis or findings.

Scope and process:

- Choose a subset of the system's features.
- Create some scenarios in which these features will be used.
- Keep the inspection process down to, say, two hours. This gives you some preliminary experience with inspection.
- Log improvement suggestions.
- Make improvements as appropriate.

++++++++++++++++++

System Feature: Division Sorting
Class affected: Mainwindow.cpp
Performed by: Shanice Hanlon

Scenario 1:
User opens TeamHercules. User then chooses the teaching heading and clicks on the load file button, then chooses the corresponding csv file. After loading the file successfully, the user has the option to create a sort. Under the create a sort drop down, the user is able to select the name of the sort to perform

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: Class diagrams were compared to header files.
Comment on your findings: Classes are accurately represented in the class diagram of the system.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: Mainwindow was tested in accordance with the specified requirements and stretch goals.
Comment on your findings: Mainwindow is a simple class that handles the dashboard display in Team Hercules. The class works as intended.

Cohesion:
Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion (good): 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: The mainwindow class was inspected prior to submission.
Comment on your findings: All the methods within each class access common data.

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

☐ Yes ■No ☐Partly (Can be reduced)

Comment on your analysis: Mainwindow was examined in relation to the other classes prior to submission.
Comment on your findings: Accessor methods were created to eliminate the possibility of different classes sharing common variables.

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: Mainwindow along with the other classes were inspected.
Comment on your findings: Each class is a well-defined interface with cohesive functions designed to target each concern or implementation. There are minimal connections between classes.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: The mainwindow header was examined.
Comment on your findings: There is proper utuilization of private and public methods to manage complexity and dependency on other code.

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: Careful examination of all classes within Team Hercules.
Comment on your findings: Most of the classes such as TreeItem can be reused. For classes such as the mainwindow, reusability is not recommended unless the application will follow a similar format to TeamHercules.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: Examination of the mainwindow header.
Comment on your findings: Methods are easily identifiable and method names are straightforward to demonstrate each task.

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

☐ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: _____
Comment on your findings: _____

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: The mainwindow class was examined.

Comment on your findings: Division sorting was a very uncomplicated implementation and therefore the conclusion is that the class is easily enhanced with other minor modifications.

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: Examination of the mainwindow class.
Comment on your findings: There are no nested loops present and no further delays in concurrent processing, thus no inefficiency in code.

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: Examination of Team Hercules headers, paying specific attention to Mainwindow.h
Comment on your findings: There is no inheritance present in the code. There are no inheritances involving the maindwindow class.

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: Careful examination of the headers, with specific attention on the mainwindow header.
Comment on your findings: Classes are not extended and therefore there are no children classes.

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:

i) Title of scenario

ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) 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: Creating and Viewing a customer
Anticipated frequency: Normal
Trigger: Clicking the "Create new sort order" button
Expected type of outputs: Members will be sorted by numerous headings including by Division.
End-user inputs:

- The user clicks on "Load File", and the program opens the file picker, allowing the information to be loaded into the application
- The user clicks on "Create new sort order" and is prompted by a dialog box with options
- User inputs name of the sort order and select first sorting header and subsequently all further sub sorting headers.
- User saves the custom sort created.
- User selects their custom sort order from the "Select sort order" drop down menu

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)
  ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question: (i) your analysis, and (ii) your finding (in the form of a comment). The analysis would typically show how you arrived at the finding.
- Add new lines as necessary for your analysis or findings.

Scope and process:

- Choose a subset of the system's features.
- Create some scenarios in which these features will be used.
- Keep the inspection process down to, say, two hours. This gives you some preliminary experience with inspection.
- Log improvement suggestions.
- Make improvements as appropriate.

++++++++++++++++++++

Class inspected: ErrorEditDialog, mainwindow, PieChartWidget
Performed by: Bowen Jiang

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:
The structure of the code in the mainwindow is directly used in ErrorEditDialog in the constructor in order to set up the class being able to reference mainwindow.

Comment on your findings:
Each class is well defined with methods that performs a certain function.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
After testing the previous and next buttons, they work excellently while traversing through the errors in the error fixing table.

Comment on your findings:
The method works very well, and the transition from one blank to the next goes very smoothly.

Cohesion:
Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion (good): 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:
Every single method performed an important function in each class

Comment on your findings:
Each class contained its own specific data

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

☐ Yes ■No ☐Partly (Can be reduced)

Comment on your analysis:
The classes are separate from one another.

Comment on your findings:
Each class has it's own variables, and do not rely on another class.

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:
Each concern is encapsulated in a class.

Comment on your findings:
All the classes are well defined with few connections with other classes

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
private and public variables are set appropriately

Comment on your findings:
There are public and private methods in each class

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:
You can you the classes for other applications.

Comment on your findings:
The two for loops for the on_prev_clicked and on_next_clicked method iterate through all of the errors in the error table and then performs an action everytime the button is clicked.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
The classes are easily understandable by friendly class names and universal variable conventions.

Comment on your findings:
Method's name are clear and concise.

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
There are comments available for complicated portions of the code

Comment on your findings:
The comments are easy to understand for all users

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:
Classes provide for easy addition of new features

Comment on your findings:
There can be minor editions of new features and code

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:
Code is efficient for each purpose.

Comment on your findings:
Uses for loops, does not delay processing.

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:
There is no inheritance relationship for the classes

Comment on your findings:
There is no inheritance for the classes

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:
They are separate classes.

Comment on your findings:
Classes do not have children.

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:

i) Title of scenario
ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) 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.:
Error box will appear when there are mandatory fields missing from the imported file, which will trigger a pop up error edit table. The user can then choose to use the previous and next button to navigate through the errors and fix them accordingly.

(Note: expand here as necessary for each scenario)

END.

Scenario 1
Title: Loading and viewing a graph
Frequency: High
Trigger: Clicking the "Load File" button
Expected outputs: A graph displays in the graph panel, with 4 different possible graphs
End-user inputs:
- The user clicks on "Load File", and the program opens the file picker, allowing the user to select the appropriate data to load in the graph
- After the file is selected, the "setupPieChart" function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different "setup___" function is called to load that graph

Scenario 2
Title: Printing a graph
Frequency: Low
Trigger: Clicking the "Print" button
Expected outputs: A physical sheet of paper with the selected graph on it
End-user inputs:
- The user clicks on "Load File", and the program opens the file picker, allowing the user to select the appropriate data to load in the graph

- After the file is selected, the “setupPieChart” function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different “setup___” function is called to load that graph
- The user clicks on “Print”, opening the print system dialogue and sending the picture of the currently open graph to it

Scenario 3
Title: Exporting as PDF
Frequency: Medium
Trigger: Clicking the “Export” button
Expected outputs: A pdf file containing a picture of the graph
End-user inputs:
- The user clicks on “Load File”, and the program opens the file picker, allowing the user to select the appropriate data to load in the graph
- After the file is selected, the “setupPieChart” function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different “setup___” function is called to load that graph
- The user clicks on “Export”, opening the system file browser, allowing the user to select where they will like to save the PDF and what to name it

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)
  ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question: (i) your analysis, and (ii) your finding (in the form of a comment). The analysis would typically show how you arrived at the finding.
- Add new lines as necessary for your analysis or findings.

Scope and process:

- Choose a subset of the system's features.
- Create some scenarios in which these features will be used.
- Keep the inspection process down to, say, two hours. This gives you some preliminary experience with inspection.
- Log improvement suggestions.
- Make improvements as appropriate.

++++++++++++++++++++

Class inspected: mainwindow, ErrorEditDialog, SessionState, RecordsManager
Performed by: Michael Song

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:
The system class diagram accurately represents the structure of the code.

Comment on your findings:
Every class is well defined with methods which follow encapsulation

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
All new requirements work well according to the project specifications, and the program is error free.

Comment on your findings:
All methods work well together, and using and navigating the program is a smooth experience.

Cohesion:
Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion (good): 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:
Methods are encapsulated in classes with their own private variables, performing functions when they are called upon and not just simply returning a value.

Comment on your findings:
Overall, encapsulation is done well throughout the program with no methods exposing public variables unnecessarily.

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

☐ Yes ■No ☐Partly (Can be reduced)

Comment on your analysis:
In the same vein as above, methods are encapsulated and do not share variables.

Comment on your findings:
Overall, methods perform their own separate functions and do not rely on other methods to perform them, resulting in low coupling.

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:
Each methods addresses a problem separately and keeps it encapsulated within the method

Comment on your findings:
Methods do not overly rely on other methods/classes in order to perform key functions

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis:
Classes have public methods to access key functions

Comment on your findings:
Internal functions remain unexposed with private methods

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:
Classes are easily re-usable for additional graphs, but do not have many use cases beyond those

Comment on your findings:
Classes and methods are rather specific to the application for the most part

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

☐Yes ☐No ■Partly (Can be improved)

Comment on your analysis:
Classes and methods are well named and documented

Comment on your findings:
Some methods could be split up into additional classes to make navigating around for them easier

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

☐Yes ☐No ■Partly (Can be improved)

Comment on your analysis:
Most parts of the code are well documented

Comment on your findings:
A few methods remain undocumented

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:
Loose coupling, well documented code and well named methods/classes allow for easy maintenance of the code base

Comment on your findings:
New features can easily be added with few changes to the existing code base

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:
Methods are straightforward and simple.

Comment on your findings:
There are no unnecessary loops or delays.

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:
Not applicable, there is no inheritance

Comment on your findings:
N/A

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:
N/A. there is no inheritence

Comment on your findings:
N/A

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:

i) Title of scenario
ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) 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.

1. Edit Error Table
2. High
3. Click on previous or next button
4. Empty boxes will be highlighted from one box to another
5. User clicks on load file, select file, edit mandatory fields, click previous or next

Comment on your findings, with specific references to the design/code elements/file names/etc.:
Error box will appear when there are mandatory fields missing from the imported file, which will trigger a pop up error edit table. The user can then choose to use the previous and next button to navigate through the errors and fix them accordingly.

(Note: expand here as necessary for each scenario)

Scenario 1
Title: Loading and viewing a graph
Frequency: High
Trigger: Clicking the "Load File" button
Expected outputs: A graph displays in the graph panel, with 4 different possible graphs
End-user inputs:

- The user clicks on "Load File", and the program opens the file picker, allowing the user to select the appropriate data to load in the graph
- After the file is selected, the "setupPieChart" function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different "setup___" function is called to load that graph

Scenario 2
Title: Printing a graph

Frequency: Low
Trigger: Clicking the “Print” button
Expected outputs: A physical sheet of paper with the selected graph on it
End-user inputs:

- The user clicks on “Load File”, and the program opens the file picker, allowing the user to select the appropriate data to load in the graph
- After the file is selected, the “setupPieChart” function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different “setup___” function is called to load that graph
- The user clicks on “Print”, opening the print system dialogue and sending the picture of the currently open graph to it

Scenario 3
Title: Exporting as PDF
Frequency: Medium
Trigger: Clicking the “Export” button
Expected outputs: A pdf file containing a picture of the graph
End-user inputs:

- The user clicks on “Load File”, and the program opens the file picker, allowing the user to select the appropriate data to load in the graph
- After the file is selected, the “setupPieChart” function is called and loads a Pie Chart by default. If the user uses a radio button to select another graph, a different “setup___” function is called to load that graph
- The user clicks on “Export”, opening the system file browser, allowing the user to select where they will like to save the PDF and what to name it

END.