AcuitySTAR

Team Pisces

Computer Science 3307A - Object-Oriented Design & Analysis
Project: Phase II Implementation – Dashboard Project
Winter 2016
University of Western Ontario, London ON
Canada N6A 3K7

Ryan Zegray
Eric Lammers
Stepan Khomyn
Novia Wang
Haris Iftikhar
Leroi Yu

Professor: Nazim Madhavji

Table of Content

Table of Content 1

Requirements Implemented 3
- Requirement ID: Charts-1 3
- Requirement ID: Charts-2 3
- Requirement ID: Charts-3 4
- Requirement ID: ErrorEdit-1 5
- Requirement ID: ErrorEdit-2 5
- Requirement ID: Sort-1 6
- Requirement ID: Sort-2 6
- Requirement ID: Sort-2 7
- Requirement ID: Search-1 8
- Requirement ID: Drag&Drop-1 8
- Requirement ID: Save State-1 9
- Requirement ID: Keyboard Shortcut-1 9

Test Cases 10
- Test Matrix 10
- Test Cases and their results 11

System Design 13
- Use Case Diagram 13
- Class Diagram 20
  - Class Diagram Enhancements 20
  - DATABASES 21
    - CSVReader 21
    - QSortListIO 21
    - RecordsManager 21
  - DATAMODEL 21
    - TreeModel 21
    - TreeItem 21
  - GUI 21
    - MainWindow 21
    - Main 22
    - CustomSort 22
    - ErrorEditDialog 22
    - PieChartWidget 22
    - QCustomPlot 22
- Sequence Diagram 23
  - 1. Sequence Diagram - Displaying a Teaching Pie Chart: 23
    - 1.1 Program user toggles pie chart button 23
    - 1.2 Program User selects a teacher's name 23
    - 1.3 System creates a list from the selected name 23
    - 1.4 System creates the pie chart 24

2. Sequence diagram – loading data from file 24
2.1 User clicks load button 24
2.2 System displays file structure – user selects file 25
2.3 System verifies file is of proper type 25
2.4 System loads data from file 25
3.1 Program user clicks user select button 26
3.2 Program User selects the items to include in the filter 26
3.3 User clicks the apply button 26
3.4 The refresh function 27
Package Diagram 27
Graphical User Interface 28
Database Model 28
Data Structure Model 28
Standard QT and C++ Libraries 28
Tests 28

Design Patterns 29
Singleton Pattern 29
Prototype Pattern 30

Implementation in C++ 33
Main Window / Dashboard 33
Sort Orders 33
Visualizing Data 33
Loading Files 34
Save State 34

Development Plans 35
Timeline 35
Agent Task View 37
Modified Agent Task View (reflecting changes in task allocation): 38

Lesson Learnt 39
Coordinating group work 39
Problems with UML 39
Working with QT and C++ 39

Inspection Documents 41

Requirements Implemented

Requirement ID: Charts-1

Description:

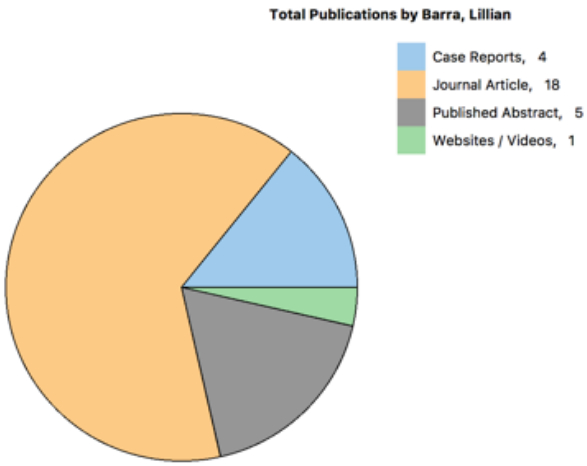
Improved overall appearance of the pie charts:

- Created a preset list of colors that work well together, improvement on the previous implementation of assigning colors at random
- Changed the outline of the pie chart from grey to black, looks sharper
- Moved the text which specifies amounts off the pie chart, now displayed after the label name

Origin:

Our group collectively was unhappy with the original appearance of the pie charts.

Requirement Type: Stretch



Requirement ID: Charts-2

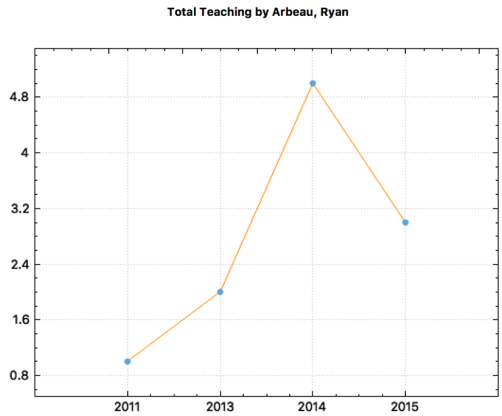
Description:

Added an option for displaying a simple line graph. Displays the data in line graph form.

Origin:

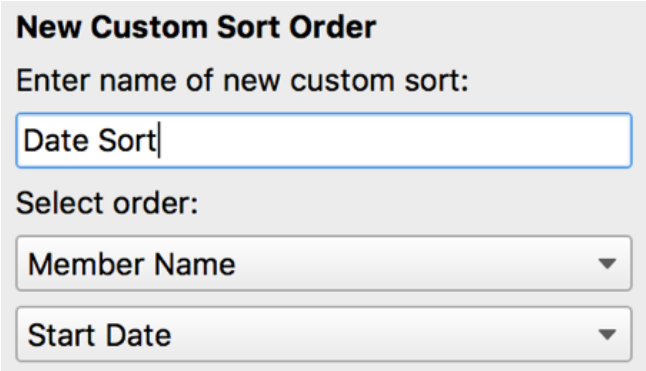
Customer Specifications

Requirement Type: Mandatory



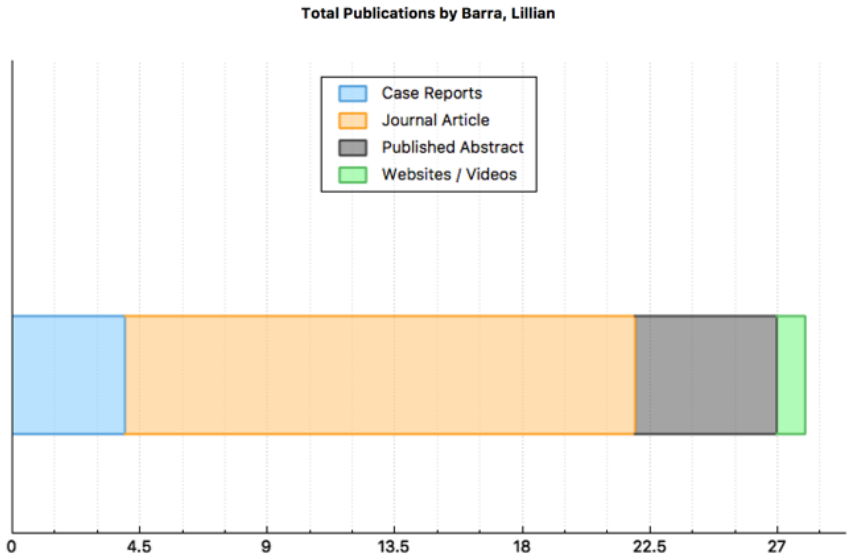
Quick side note about all charts: You can customize which data points are plotted on the graph by modifying the sort order.

So, for example if you wanted the line graph to plot teaching for a specific person by the date like in the graph above. You would create a new sort that sorts by Name then then date like so:



Requirement ID: Charts-3

Description:
Added an option for displaying a stacked bar char. Displays the data in a stacked bar chart form so one can easily compare the relative size of each data point.
Origin:
Customer Specifications
Requirement Type: Mandatory



Requirement ID: ErrorEdit-1

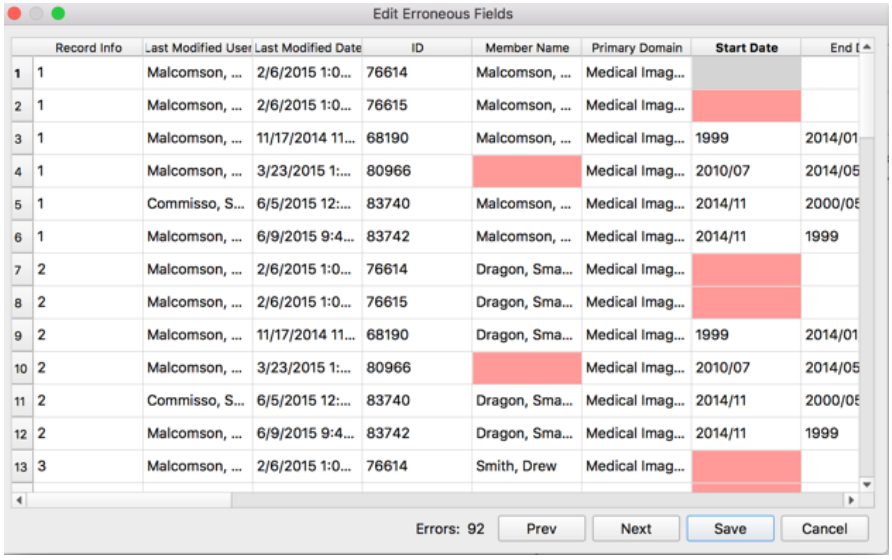
Description:

Added a find next and find previous button for the error edit dialog. It also displays a count of the errors left to fill out.

Origin:

Customer Specifications

Requirement Type: Mandatory



Edit Erroneous Fields

	Record Info	Last Modified User	Last Modified Date	ID	Member Name	Primary Domain	Start Date	End D
1	1	Malcomson, ...	2/6/2015 1:0...	76614	Malcomson, ...	Medical Imag...		
2	1	Malcomson, ...	2/6/2015 1:0...	76615	Malcomson, ...	Medical Imag...		
3	1	Malcomson, ...	11/17/2014 11...	68190	Malcomson, ...	Medical Imag...	1999	2014/01
4	1	Malcomson, ...	3/23/2015 1:...	80966		Medical Imag...	2010/07	2014/05
5	1	Commisso, S...	6/5/2015 12:...	83740	Malcomson, ...	Medical Imag...	2014/11	2000/05
6	1	Malcomson, ...	6/9/2015 9:4...	83742	Malcomson, ...	Medical Imag...	2014/11	1999
7	2	Malcomson, ...	2/6/2015 1:0...	76614	Dragon, Sma...	Medical Imag...		
8	2	Malcomson, ...	2/6/2015 1:0...	76615	Dragon, Sma...	Medical Imag...		
9	2	Malcomson, ...	11/17/2014 11...	68190	Dragon, Sma...	Medical Imag...	1999	2014/01
10	2	Malcomson, ...	3/23/2015 1:...	80966		Medical Imag...	2010/07	2014/05
11	2	Commisso, S...	6/5/2015 12:...	83740	Dragon, Sma...	Medical Imag...	2014/11	2000/05
12	2	Malcomson, ...	6/9/2015 9:4...	83742	Dragon, Sma...	Medical Imag...	2014/11	1999
13	3	Malcomson, ...	2/6/2015 1:0...	76614	Smith, Drew	Medical Imag...		

Errors: 92 Prev Next Save Cancel

Requirement ID: ErrorEdit-2

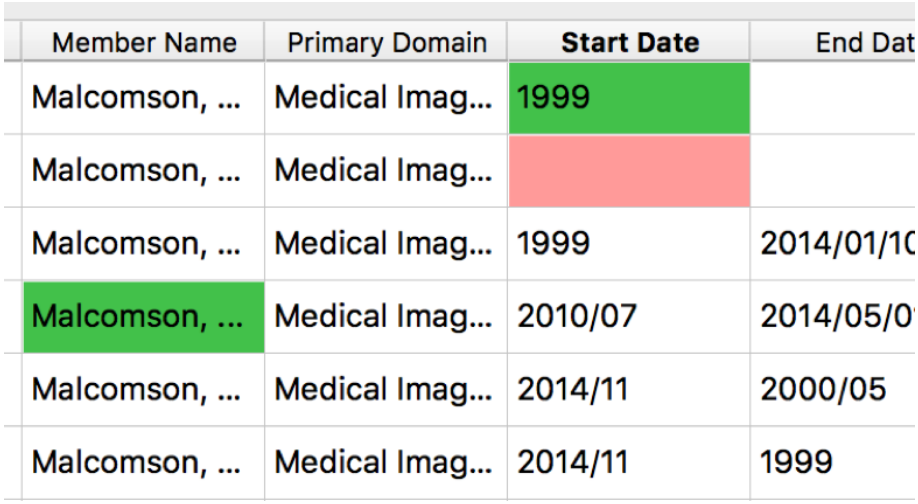
Description:

Added the feature that when a data item is filled in it turns green to show that the cell has been filled out.

Origin:

Our group thought that it was hard to tell which fields have been filled in and which fields still need to be filled out.

Requirement Type: Mandatory



Member Name	Primary Domain	Start Date	End Dat
Malcomson, ...	Medical Imag...	1999	
Malcomson, ...	Medical Imag...		
Malcomson, ...	Medical Imag...	1999	2014/01/1(
Malcomson, ...	Medical Imag...	2010/07	2014/05/0
Malcomson, ...	Medical Imag...	2014/11	2000/05
Malcomson, ...	Medical Imag...	2014/11	1999

Requirement ID: Sort-1

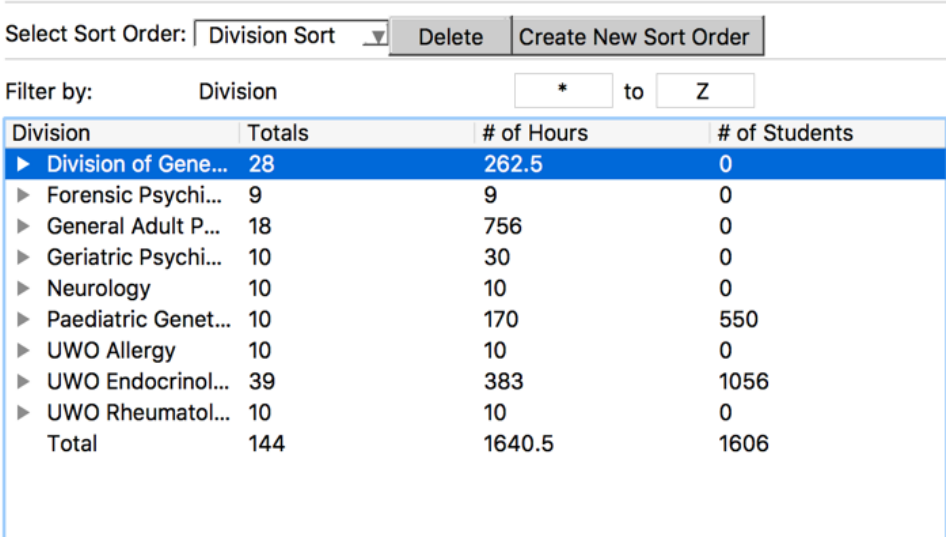
Description:

Added a sort by division.

Origin:

Customer Specifications

Requirement Type: Mandatory



Division	Totals	# of Hours	# of Students
Division of Gene...	28	262.5	0
Forensic Psychi...	9	9	0
General Adult P...	18	756	0
Geriatric Psychi...	10	30	0
Neurology	10	10	0
Paediatric Genet...	10	170	550
UWO Allergy	10	10	0
UWO Endocrinol...	39	383	1056
UWO Rheumatol...	10	10	0
Total	144	1640.5	1606

Requirement ID: Sort-2

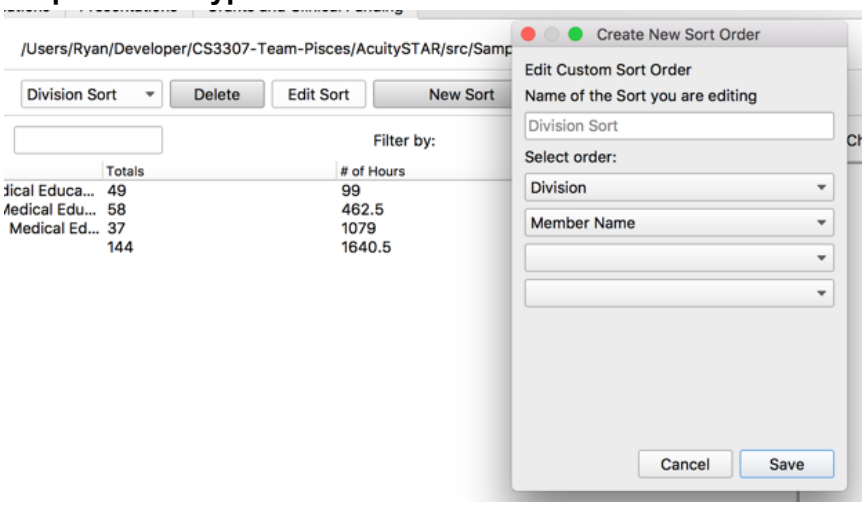
Description:

Added the ability to change/modify an existing sort order.

Origin:

Customer Specifications

Requirement Type: Stretch



Requirement ID: Sort-3

Description:

Added the ability to sort by a user selected list

Origin:

Customer Specifications

Requirement Type: Mandatory

Press “User Select”

Search Records: Filter by: User Select Apply Cancel Member Name * to Z

Member Name	Totals	# of Hours	# of Students
AAbel, Angra (Test Record)	4	4	35
Adams, Paul	31	653	0
Aguanno, Alaina	13	55.1	177
Al-Judaibi, Bandar	7	15	9
Althenayan, Eyad	22	108.1	589
Anderson, Scott	15	286	0
Arbeau, Ryan	11	103	110
Arntfield, Robert	56	106	0
Bagur, Rodrigo	19	514	2
Baig, Mirza	3	8	0

Select your custom list

Search Records: Filter by: User Select Apply Cancel Member Name * to Z

Member Name	Totals	# of Hours	# of Students
AAbel, Angra (Test Record)	4	4	35
Adams, Paul	31	653	0
Aguanno, Alaina	13	55.1	177
Al-Judaibi, Bandar	7	15	9
Althenayan, Eyad	22	108.1	589
Anderson, Scott	15	286	0
Arbeau, Ryan	11	103	110
Arntfield, Robert	56	106	0
Bagur, Rodrigo	19	514	2
Baig, Mirza	3	8	0
Ball, Ian	18	29.5	1469

Click “Apply” to filter the list by only the names you selected

Search Records: Filter by: User Select Apply Cancel Member Name * to Z

Member Name	Totals	# of Hours	# of Students
AAbel, Angra (Test Record)	4	4	35
Adams, Paul	31	653	0
Aguanno, Alaina	13	55.1	177
Al-Judaibi, Bandar	7	15	9
Althenayan, Eyad	22	108.1	589
Anderson, Scott	15	286	0
Arbeau, Ryan	11	103	110
Arntfield, Robert	56	106	0

Requirement ID: Search-1

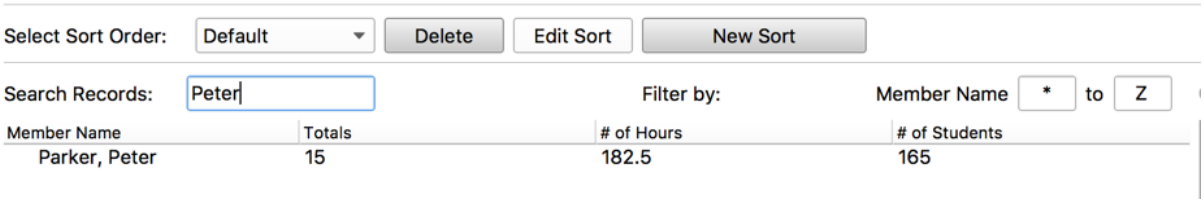
Description:

Added an advanced search feature to search through the data in the tree view.

Origin:

Customer Specifications

Requirement Type: Stretch



Requirement ID: Drag&Drop-1

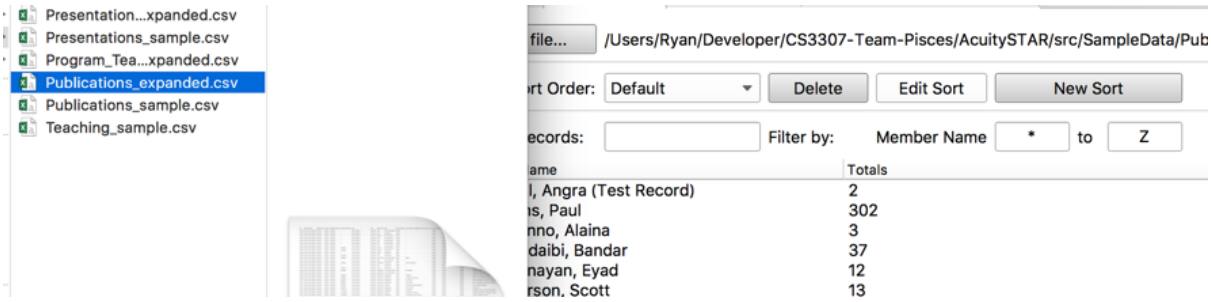
Description:

Added the ability for a user to simply drag and drop a csv file into the application to load it.

Origin:

Our group found it a pain to have to use the file explorer every time we wanted to load a file and wanted to find a way to make loading files easier.

Requirement Type: Stretch



Requirement ID: Save State-1

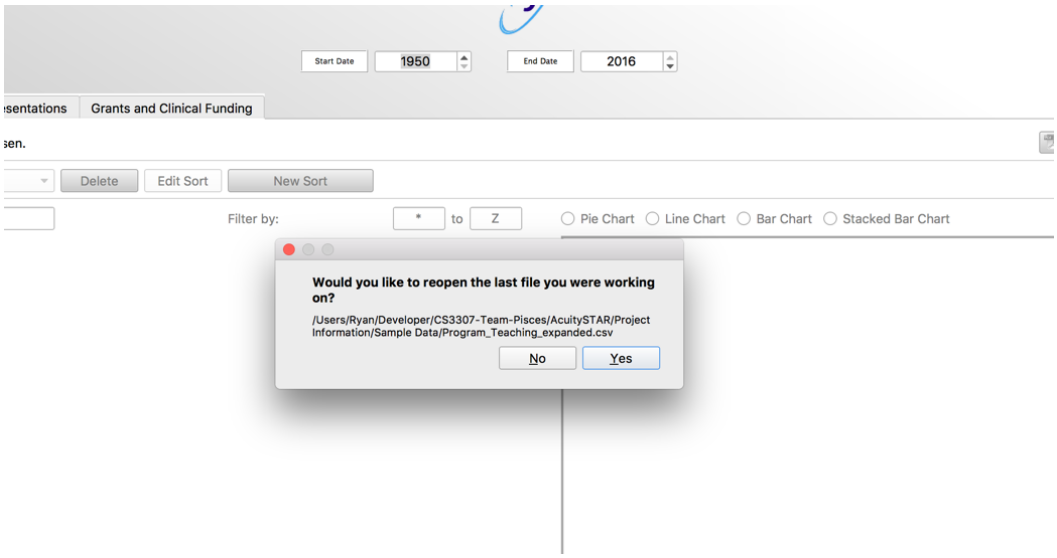
Description:

Added a prompt to re-open the file the user was last working on and switches to the appropriate tab.

Origin:

Customer Specifications

Requirement Type: Mandatory



Requirement ID: Keyboard Shortcut-1

Description:

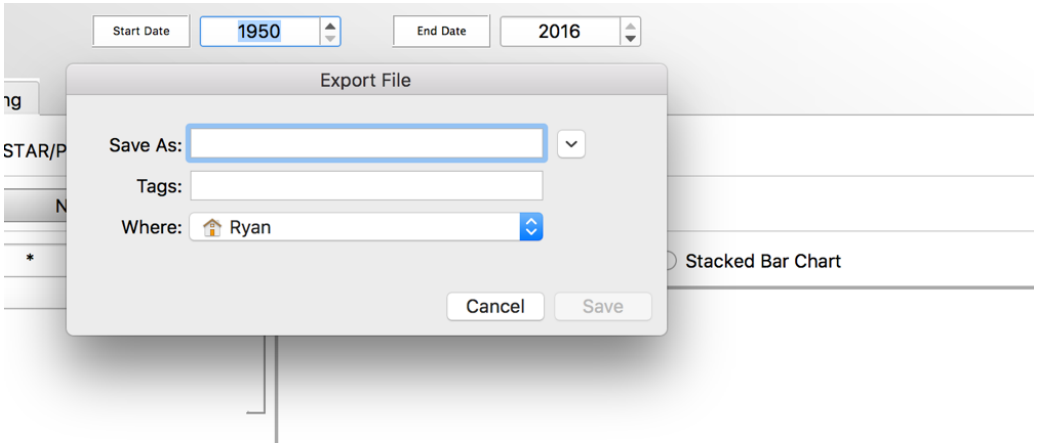
Added several keyboard shortcuts to make navigating the program easier.

- Control-P: Print Graphs
- Control-E: Export Graphs
- Control-L: Load a file
- Control-1/2/3/4 – Quick Switch between Teaching, Publications, Presentation and Grants tab

Origin:

Our group wanted to add a way for power users to be able to quickly navigate around our program.

Requirement Type: Stretch



Test Cases

Test Matrix

Implemented Requirements of Stage 2 System X Test Cases Involved in its verification

	Charts 1 (Charts Color)	Charts 2 (Line Chart)	Charts 3 (Stacked bar cahrt)	Error-Edit 1 (Next/ Prev)	Error-Edit 2 (Error color)	Sort-1 (Division Sort)	Sort-2 (Edit Sort Order)	Sort-3 (User selected list)	Search-1 (Search Data)	Drag & Drop-1	Save State-1	Keyboard Shortcuts -1
10-1	x											
10-2		x										
10-3				x								
10-4						x						
10-5		x										
11-5			x									
11-6				x								
11-7				x								
11-8										x		
11-9							x					
11-10							x					
11-11					x							
11-12									x			
12-1												x
12-2												x
12-3												x
12-4												x
12-5												x
12-6												x
12-7												x
12-8											x	
12-9											x	
12-10											x	
12-11											x	
12-12								x				

Test Cases and their results

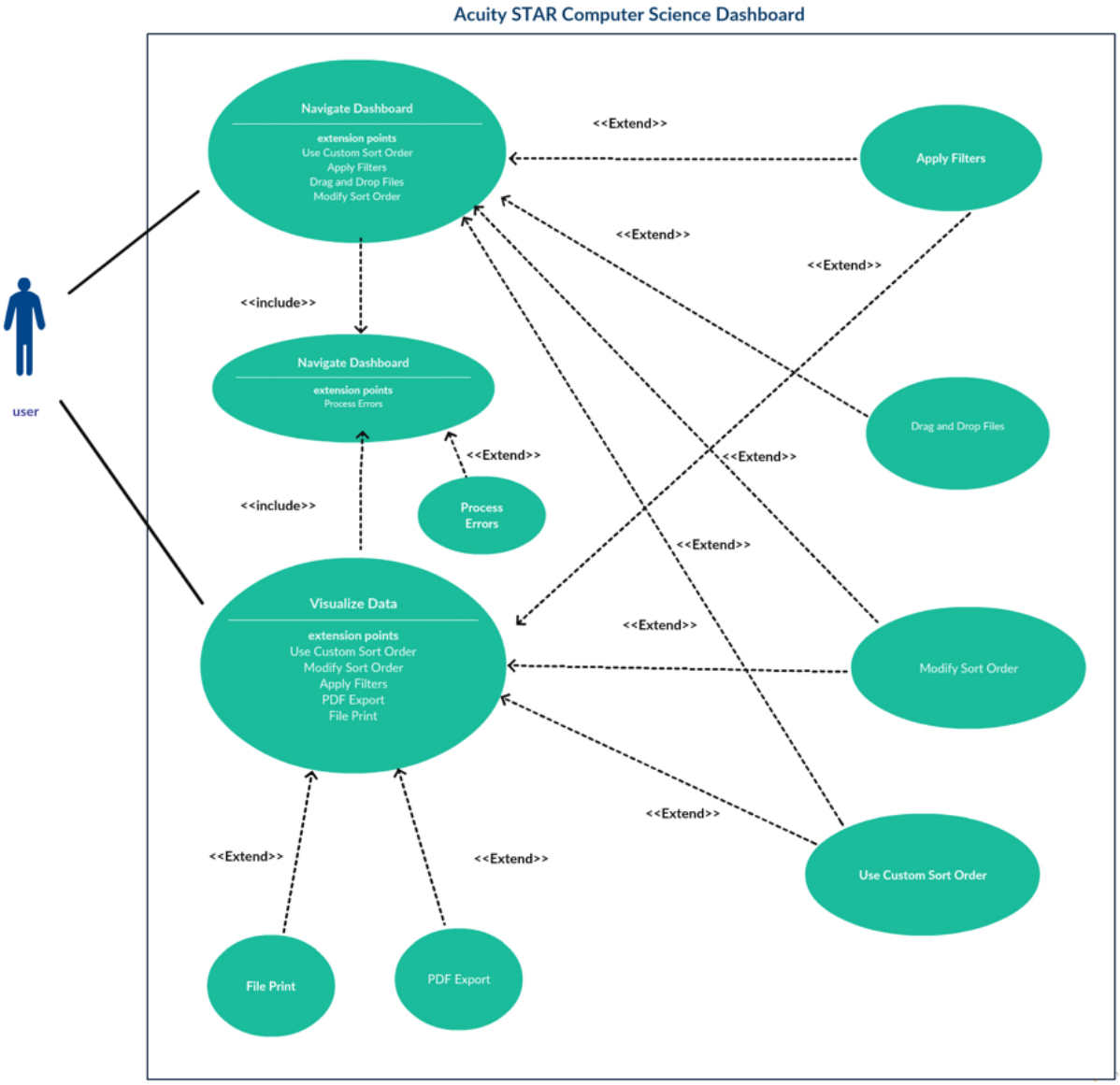
Test Case Id	Unit to Test	Requirements	Assumptions	Test data	Steps to be executed	Expected result	Actual result	Pass/Fail	Comments
1: Test Loading Data from a file									
1-1	CSVReader.cpp	Reads Grants and Clinical Funding CSV files successfully	The file header will not be empty	GrantsClinicalFunding_sample_sample.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
1-2	CSVReader.cpp	Reads Presentations CSV files succesfully	The file header will not be empty	Presentations_sample.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
1-3	CSVReader.cpp	Reads Publications CSV files successfuly	The file header will not be empty	Publications_sample.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
1-4	CSVReader.cpp	Reads Teaching CSV files successfuly	The file header will not be empty	Teaching_sample.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
1-5	CSVReader.cpp	Does not read in a file that does not exist	The file path we are attempting to load does not exist	"thisfiledoesnotexist.cvs" (this file does not exist in the working directory)	1. Read the file using CVSReader and 2. Check if the headers are still empty. If the headers are still empty we know that the file was not read.	The file was not read.	The file was not read.	Pass	
1-6	CSVReader.cpp	Does not read in a invalid file type		invaliddata.txt	1. Read the file using CVSReader and 2. Check if the headers are still empty. If the headers are still empty we know that the file was not read.	The file was not read.	libc++abi.dylib: terminating with uncaught exception of type std::out_of_range: basic_string The program has unexpectedly finished.	Fail	Fixed Below
1-6 BUGFIX	CSVReader.cpp	Does not read in a invalid file type		invaliddata.txt	1. Read the file using CVSReader and 2. Check if the headers are still empty. If the headers are still empty we know that the file was not read.	The file was not read.	The file was not read.	Pass	The previous code was not checking the file extension. Just if the file exists. This means selecting any file other then a .csv crashed the program. This was fixed by checking the file extension before loading the file
2: Test Displaying a summary of data									
2-1	mainwindow.cpp	Grants data is displayed on screen		GrantsClinicalFunding_sample_sample.csv	1. Go to the grants tab. 2. Press Load file and select a grants CVS. 3. Load the file and discard any errors.	The summarized data should be displayed for the user to look at	The summerized data was displayed	Pass	
2-2	mainwindow.cpp	Presentations data is displayed on screen		Presentations_sample.csv	1. Go to the presentations tab tab. 2. Press Load file and select a presentations CVS. 3. Load the file and discard any errors.	The summarized data should be displayed for the user to look at	The summerized data was displayed	Pass	
2-3	mainwindow.cpp	Publications data is displayed on screen		Publications_sample.csv	1. Go to the publications tab. 2. Press Load file and select a publications CVS. 3. Load the file and discard any errors.	The summarized data should be displayed for the user to look at	The summerized data was displayed	Pass	
2-4	mainwindow.cpp	Teaching Data is displayed on screen		Teaching_sample.csv	1. Go to the teaching tab. 2. Press Load file and select a teaching CVS. 3. Load the file and discard any errors.	The summarized data should be displayed for the user to look at	The summerized data was displayed	Pass	
3: Test visualizing/graphing of data									
3-1	mainwindow.ui / piechartwidgit.cpp	Check that the pie chart is displayed for all four subject areas	Pie charts will be displayed	PieChartWidgit	For each subject area select pie chart option, select a section, see if pie chart is displayed	The pie chart will be displayed	The pie chart was displayed	Pass	
3-2	qcustomplot.cpp	Check that the bar chart is displayed for all four subject areas			For each subject area select bar chart option, select a section, see if pie chart is displayed	The bar chart will be displayed	The bar chart was displayed	Pass	
4: Test dashboard navigation									
4-1	mainwindow.ui	Expand button expands sections and collapse button collapses sections		teachTreeView, pubTreeView, presTreeView, fundTreeView	1. Load a file. 2. Expand a section in the tree view. 3. Collapse a section on the tree view. Repeat for all 4 file types	The section will expands then collapse	The sections expanded and then collapsed	Pass	
4-2	mainwindow.ui	The user can use the tab bar to navigate to different tabs for each of the 4 subjects. Teaching, Publications, Presentations and Funding		categoryTab	1. Click on the Grants tab to navigate to it. 2. Click on the Presentations Tab to navigate to it. 3. Click on the presentations tab to navigate to it 4. Click on the teaching tab to navgation tab	You can navigate to all the tabs	You can navigate to all the tabs	Pass	
5: Test filtering and sorting data									
5-1	mainwindow.cpp	Changing a item in the filter_from box filters the data		teach_filter_from, pub_filter_from, pres_filter_from, fund_filter_from	1. Change text in the filter_from box. 2. Check if the data has been filtered	The change in the filter_from box will trigger a textChanged() event and cause the tree view to be update to reflect the filter	The data was filtered	Pass	
5-2	mainwindow.cpp	Chaning an item in the filter_to box filters the data		teach_filter_to, pub_filter_to, pres_filter_to, fund_filter_to	1. Change text in the filter_to box 2. Check if the data has been filtered	The change in the filter_to box will trigger a textChanged() event and cause the tree view to be update to reflect the filter	The data was filtered	Pass	
5-3	customsort.cpp	Allow the user to choose how to sort the data on screen		Teaching_sample.csv	1. Load a data file. 2. Create a custom sort. 3. Change the sort order to the custom sort	The data will be sorted	The data was sorted	Pass	If you attempt to create a sort with the Start Date first it will tell you you cannot sort by date first. While this is not a bug it is not very user friendly. This should be added to our "improvements" and we should prevent users from being able to sort by date first for a future deliverable.
6: Test PDF and Print Export									
6-1	mainwindow.cpp	You can export a pie chart as a pdf		Teaching_sample.csv	1. Load a data file. 2. Select a pie chart. 3. Export as a pdf	The chart will be exported as a pdf	The chart was exported	Pass	
6-2	mainwindow.cpp	You can export a bar chart as a pdf		GrantsClinicalFunding_sample.csv	1. Load a data file. 2. Select a bar chart. 3. Export as a pdf	The chart will be exported as a pdf	The char was exported	Pass	
6-3	mainwindow.cpp	You cannot export a blank pdf		Teaching_sample.csv	1. Load a data file. 2. Before selecting a chart press the export button.	The program will not let you export a empty pdf	The empty pdf was exported	Fail	If you export a PDF before selecting a data value you can export a empty PDF. Fixed Below
6-3: BUGFIX	mainwindow.cpp	You cannot export a blank pdf		Teaching_sample.csv	1. Load a data file. 2. Before selecting a chart press the export button.	The program will not let you export a empty pdf	The program will not let you export a empty pdf	Pass	Before the print buttons were being enabled as soon as data is imported to the program. This was fixed by only enabling the print buttons once they have selected data to print using "teachTreeView_clicked"
7: Testing Tree data structure									
7-1	treeitem.cpp void appendChild (TreeItem *child);	You can insert items into a tree	If the number of children is equal to the number of children appended to the TreeItem the Insert is working correctly	TreeItem parent item, child1, child2, child3	1. Create all 4 TreeItems using test data. 2. Set the child items to the parent item. 3. Check the number of children. Compare the number of children with the number of children appended to the tree.	The number of children inserted should be equal to the number of children	The number of children inserted was equal to the number of children	Pass	
7-2	treeitem.cpp TreeItem *parentItem();	You can get the parent from a tree		TreeItem parent item, child1, child2, child3	1. Create all 4 Tree items user test data. 2. Set the child items to the parent items. 3. Check if the child nodes parent is equal to the parent node	The child nodes parent should be equal to the parent node	The child nodes parent should be equal to the parents node	Pass	
7-3	treeitem.cpp TreeItem *child (int row);	You can get the children from a tree		TreeItem parent item, child1, child2, child3	1. Create all 4 Tree items user test data. 2. Set the child items to the parent items. 3. Check if the children node is equal to the parents children	The parent nodes child should be equal to the child node	The parent nodes child should be equal to the child node	Pass	
8: Test Error Processing									
8-1	EditErrorDialog.cpp	If a data file contains any invalid records the program will propt the user to edit or discard them	The datafile we are testing with contains at least 1 invalid record	Publications_sample.csv	1. Load a data file that contains an invalid record	The program showns the number of invalid records andpromps the user to edit or discard the invalid record	The program showns the number of invalid records andpromps the user to edit or discard the invalid record	Pass	

8-2	EditErrorDialog.cpp	If the user presses the button to edit invalid records the user will be brought to a screen to fill in the missing data in all invalid records.	The datafile we are testing with contains at least 1 invalid record	Publications_sample.csv	1. Load a data file that contains an invalid record 2. Press the edit button in the popup	The program displays the valid records along with the invalid that was corrected	The program displays the valid records along with the invalid that was corrected	Pass	
8-3	EditErrorDialog.cpp	If the user fills in the missing data and presses save. The program will display the valid data and the invalid data that has been corrected	The datafile we are testing with contains at least 1 invalid record	Publications_sample.csv	1. Load a data file that contains an invalid record 2. Press the edit button in the popup 3. Fill in the missing data. 4. Press the "save" button	The program displays the valid records along with the invalid that was corrected	The program displays the valid records along with the invalid that was corrected	Pass	
8-4	EditErrorDialog.cpp	If the user presses the button to discard invalid records. The program will discard these records and display the remaning data	The datafile we are testing with contains at least 1 invalid record	Publications_sample.csv	1. Load a data file that contains an invalid record 2. Press the "discard" button	The program discards the invalid records and only displays the valid records	The program discards the invalid records and only displays the valid records	Pass	
9: Testing QSortListIO									
9-1	QSortListIO class	The class should be able to save a QList of QStringLists to a file and then read the QList back out of the file.	The class will have the functionality to save a QList to a file and read it back out, the QList that we read from the file should be identical to our original QList. We will use QSortListIO::saveList (QList<QStrings>) to save our test QList to a file and then use QSortListIO::readList() to read it back out.	Made a test QList<QStringList>	1. Create a QList<QStringList> 2. Create a QSortListIO object 3. Use QSortListIO::saveList() to save to file 4. UseQSortListIO::readList() to retrieve the list 5. Compare the list to the original to make sure it did not get corrupted in the process	The List returned by readList() will be the same as the original	The Lists were identical	Pass	
10: Phase 1 System									
10-1	mainwindow.cpp / piechartwidgit.cpp	PieChart should utilize a preset list of colors that work well together instead of picking colors at random		Teaching_sample.csv	1. Load a datafile. 2. Select a dataitem to display its chart.	The chart will use our preset list of colours.	The chart was displayed using our preset list of colors	Pass	
10-2	mainwindow.cpp / qcustomplot.cpp	The UI should display a line graph for the user to look at.		Teaching_sample.csv	1. Load a datafile. 2. Select a data item to display. 3. Select the line chart button.	A line chart will be displayed summarizing the data	A line chart was displayed summarizing the data	Pass	
10-3	editerrordialog.cpp	The dialog shsould contain a find next button that finds the next data item missing information		Teaching_sample.csv	1. Load a datafile. 2. Select edit the errors. 3. Fix the first error. 4. Press the find find next button	The dialog should jump to the next error	The dialog jumped to the next error	Pass	
10-4	mainwindow.cpp	The user should be able to sort by division.		Teaching_sample.csv	1. Load a teaching data file. 2. Select create new custom sort. 3. Create a new custom sort by division. 4. Select that sort to be used	The data will be sorted by division	The data was sorted by division	Pass	
10-5	testcharts.cpp	The program should not be able to setup and empty line chart		nullptr	1. Call setupLineChart and pass in nullptr for the check. Check to make sure that it returns false	it should return false	it returns false	Pass	
10-6	testcharts.cpp	The program should not be able to setup and empty bar chart		nullptr	1. Call setupBarChart and pass in nullptr for the check. Check to make sure that it returns false	it should return false	it returns false	Pass	
10-7	testcharts.cpp	The program should not be able to setup and empty pie chart		nullptr	1. Call setupPieChart and pass in nullptr for the check. Check to make sure that it returns false	it should return false	it returns false	Pass	
10-8	editerrordialog.cpp	All text feilds in the edit error dialog are editable		Teaching_sample.csv	1. Load a datafile 2. Select to edit the invalid data. 3. Attempt to edit a cell that contains valid data and attemp to edit a cell that contains invalid data	Both cells should be editable	Both cells are editable	Pass	
10-9	editerrordialog.cpp	When someone makes changes in the edit error data field to already valid data those changes are maintained when the data is loaded into mainwindow		Teaching_sample.csv	1. Load a datafile 2. Select to edit the invalid data. 3. Edit a cell that contains valid data and fill in all missing data 4. Click save	The change made should persist into the mainwindow	The change did not persist into the mainwindow	Fail	**This feature is still being worked on! It will be implemented for a future delivarable.**
11: Phase 2 System									
11-1	CSVReader.cpp	The program should be able to read the new teaching files that include division and department data items	The file header will not be empty	Program_Teaching_expanded.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
11-2	CSVReader.cpp	The program should be able to read the new publication files that include division and department data items	The file header will not be empty	Publications_expanded.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
11-3	CSVReader.cpp	The program should be able to read the new presentation files that include division and department data items	The file header will not be empty	Presentations_expanded.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
11-4	CSVReader.cpp	The program should be able to read the new grants files that include division and department data items	The file header will not be empty	Grants_expanded.csv	1. Read the file using CVSReader and 2. Check if the header is not empty. If the header is not empty we know that it has been read correctly	The header will not be empty	The header was not empty	Pass	
11-5	testcharts.cpp	The program should not be able to setup and empty stacked bar chart		nullptr	1. Call setupStackedLineChart and pass in nullptr for the chart. Check to make sure that it returns false	it should return false	it returns false	Pass	
11-6	editerrordialog.cpp	The dialog should contain a find next button that finds the next data item missing information		Teaching_sample.csv	1. Load a datafile. 2. Select edit the errors. 3. Fix the first error. 4. Press the find find next button	The dialog should jump to the next error	The dialog jumped to the next error	Pass	
11-7	editerrordialog.cpp	The dialog should contain a find previous button that finds the previous data item missing information		Teaching_sample.csv	1. Load a datafile. 2. Select edit the errors. 3. Fix an error. 4. Press the find previous button	The dialog should jump to the previous error	The dialog jumped to the previous error	Pass	
11-8	mainwindow.cpp	You should be able to drag and drop a file to load it		Teaching_sample.csv	1. Drag the file into the application window and drop it	The file should be loaded	The file was loaded	Pass	
11-9	mainwindow.cpp / erroreditdialog.cpp	You should be able to edit an existing sort			1. Load a data file. 2. Select a custom sort. 3. Press the edit sort button. 4. Change the sort 5. Press save	The sort should be modified	The sort was modified	Pass	
11-10	mainwindow.cpp / erroreditdialog.cpp	You should not be able to edit the default sort order			1. Load a data file. 2. Press the edit sort button. 3. Change the sort 4. Press save	The application should display that you cannot edit the default sort	The application displayed that you cannot edit the default sort order	Pass	
11-11	erroreditdialog.cpp	When an error is fixed its cell should switch to green signaling that the cell has been fixed.		Teaching_sample.csv	1. Load a file. 2. Click the edit error button. 3. Fix several errors.	The errors should switch from red to green	The errors switched from red to green	Pass	
11-12	mainwindow.cpp	You can search through the data in the tree view		Teaching_sample.csv	1. Load a file 2. Type text in the search bar.	The data should be filtered to only show data that contains the search string	The data was filtered to only show data that contains the search string	Pass	

11: Phase 3 System									
12-1	mainwindow.cpp	The user should be able to press control + p to print a graph		Teaching_sample.csv	1. Load a data file 2. Click on a data point to create the graph 3. Press control P to print	The program brings up the print menu	The program brought up the print menu	Pass	
12-2	mainwindow.cpp	The user should be able to press control + e to export a graph		Teaching_sample.csv	1. Load a data file 2. Click on a data point to create the graph 3. Press control E to export	The program brings up the export menu	The program brought up the eport menu	Pass	
12-3	mainwindow.cpp	The user should be able to press control + l to load a file		Teaching_sample.csv	1. Open the program 2. Press Control L to load a data file	The program brings up the load file menu	The program brought up the load file menu	Pass	
12-4	mainwindow.cpp	The user should be able to press control + 1 to switch to the teaching tab			1. Open the program 2. Switch to a tab other than teaching 3. Press Control 1 to switch to the teaching tab	The program switches to the teaching tab	The program switched to the teaching tab	Pass	
12-5	mainwindow.cpp	The user should be able to press control + 2 to switch to the publications tab			2. Open the program 2. Press control 2 to switch to the publications tab	The program switches to the publications tab	The program switched to the publications tab	Pass	
12-6	mainwindow.cpp	The user should be able to press control + 3 to switch to the presentations tab			2. Open the program 2. Press control 3 to switch to the presentations tab	The program switches to the presentations tab	The program switched to the presentations tab	Pass	
12-7	mainwindow.cpp	The user should be able to press control + 4 to switch to the grants tab			2. Open the program 2. Press control 4 to switch to the grants tab	The program switches to the grants	The program switched to the grants tab	Pass	
12-8	mainwindow.cpp	The program should save the last file opened if the last file is teaching		Program_Teaching_expanded.csv	1. Load a teaching file. 2. Close the program. 3. Reopen the program	The program promps the user to reopen the last file they had open	The program promped the user to reopen the last file they had open	Pass	
12-9	mainwindow.cpp	The program should save the last file opened if the last file is publications		Publications_expanded.csv	1. Load a publications file. 2. Close the program. 3. Reopen the program	The program promps the user to reopen the last file they had open	The program promped the user to reopen the last file they had open	Pass	
12-10	mainwindow.cpp	The program should save the last file opened if the last file is presentations		Presentations_expanded.csv	1. Load a presentations file. 2. Close the program. 3. Reopen the program	The program promps the user to reopen the last file they had open	The program promped the user to reopen the last file they had open	Pass	
12-11	mainwindow.cpp	The program should save the last file opened if the last file is grants		Grants_expanded.csv	1. Load a teaching file. 2. Close the program. 3. Reopen the program	The program promps the user to reopen the last file they had open	The program promped the user to reopen the last file they had open	Pass	
12-12	mainwindow.cpp	The program should be able to sort by a user selected list		Program_Teaching_expanded.csv	1. Load a file 2. Press the "User Select" Button 3. Select a group of names 4. Press Apply	The tree view should only list the selected items	The tree view should only list the selected items	Pass	

System Design

Use Case Diagram



Above is the Acuity STAR use case diagram. A user refers to anyone running the program, be it a faculty member, department manager, or anyone authorized to use the program. Each user has two use cases, namely “Navigate Dashboard” and “Visualize Data”. Each of these use cases corresponds to a customer requirement: Navigate Dashboard addresses the dashboard screen requirement, while Visualize Data addresses the visualizations requirement. Note that both of these use cases include another use case, “Load Data from File”, which addresses the CSV file data processing requirement. The other use cases are extensions going on beyond the user requirements and reflect our stretch goals. Below are the texts for each use case:

1.1 Loading data from file

Load Data from File (Sea Level)

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.

Alternate Course A: File is of invalid type

1. The system displays an error message.

2. The user accepts or closes the error message.

Alternate Course B: User clicks Cancel button

1. The system closes the file structure screen.

1.2 Error processing

Process Errors (Sea Level)

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.

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.

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]

Alternate Course C: User clicks Cancel button

1. The system discards records with missing mandatory entries.

2. The system closes the error processing screen.

1.3 Navigating the dashboard

Navigate Dashboard (Sea Level)

Main Success Scenario:

1. The user loads the data from file via the use case 3*.1.1 Loading data from file.* [Extension Point: 3*.1.4 Applying filters*. Applicable to step 3.] [Extension Point: 3*.1.5 Dragging and Dropping Files*. Applicable to step 3.] [Extension Point: *3.1.6 Using a custom sort order*. Applicable to step 3.]

2. The system displays the updated dashboard summary view.

3. The user expands/collapses elements of the dashboard summary view.

4. The system displays the expanded/collapsed elements of the dashboard summary view.

1.4 Applying Filters

Apply Filters (Sea Level)

Main Success Scenario:

1. The user modifies the values in the start and end date boxes.

2. The system sets its date range according to the values in the start and end date boxes.

3. The user modifies the values in the first and last letter of member last name boxes.

The system sets its member name range according to the values in the first and last letter of member last name boxes.

1.5 Drag and Drop Files

Drag and Drop Files (Sea Level)

Main Success Scenario:

1. The user drags a file from a specific finder.
2. The user drops the file into the dashboard. [Alternate Course A: File is not CSV type]
3. The system verifies if the records contain any missing fields. [Extension Point: 3.1.2 Error Processing]
4. The system loads the records

Alternate Course A: File is of invalid type

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

1.6 Using a custom sort order

Use Custom Sort Order (Sea Level)

Main Success Scenario:

1. The user clicks Create New Sort Order button. [Alternate Course A: User selects existing sort order] [Alternate Course D: The user clicks the Edit Sort Order button]

2. The system displays a new sort order screen.

3. The user enters the name of the new sort order, selects the hierarchy of filters to order the sort by, and clicks the Save button. [Alternate Course B: User does not enter name] [Alternate Course C: User clicks Cancel button]

4. The system closes the new sort order screen.

5. The system adds the new sort order to the list of existing sort orders.

6. The user selects the sort order from the list of existing sort orders.

7. The system sets its sort order to the one selected in the list of existing sort orders.

Alternate Course A: User selects existing sort order

1. [Return to Main Success Scenario step 6]

Alternate Course B: User does not enter name

1. The system displays an error message.

2. The user accepts or closes the error message. [Return to Main Success Scenario step 3]

Alternate Course C: User clicks Cancel button

1. The system closes the new sort order screen.

Alternate Course D: The user clicks the Edit Sort Order button

1. The system displays the edit sort order screen. With the current hierarchy of filters to sort by.

2. The user can then modify the current hierarchy of filters and click the save button.

3. The system closes the edit sort order screen.

4. The system replaces the existing sort order with the modified sort order in the list of existing sort orders.

5. The user selects the modified sort order from the list of existing sort orders.

6. The system sets its sort order to the one selected in the list of existing sort orders.

1.7 Visualizing data

Visualize Data (Sea Level)

Main Success Scenario:

1. The user loads the data from file via the use case 3*.1.1 Loading data from file.* [Extension Point: 3*.1.4 Applying filters*. Applicable to step 3] [Extension Point: *3.1.6 Using a custom sort order*. Applicable to step 3]

2. The system displays the dashboard summary view.

3. The user clicks on an element in the dashboard summary view. [Extension Point: *3.1.8 Exporting to PDF*. Applicable to step 5] [Extension Point: *3.1.9 Printing to file*. Applicable to step 5]

4. The system displays a visualization (default is Pie Chart) of the selected element.

5. The user clicks on the Bar Graph radio button.

6. The system displays a bar graph of the selected element.

1.8 Exporting to PDF

PDF Export (Sea Level) Main Success Scenario:

1. The user clicks the Export button.

2. The system displays a file structure screen.

3. The user selects a file path, enters a file name and clicks the Save button. [Alternate Course A: No file name entered] [Alternate Course B: User clicks Cancel button]

4. The system exports the selected visualization type to a PDF with the entered file name at the selected file path.

Alternate Course A: No file name entered

1. The system displays an error message.

2. The user accepts or closes the error message. [Return to Main Success Scenario step 3]

Alternate Course B: User clicks Cancel button

1. The system closes the file structure screen.

1.9 Printing to File

File Print (Sea Level)

Main Success Scenario:

5. The user clicks the Print button.

6. The system displays a file structure screen.

7. The user selects a file path (default is current working directory), enters a file name (default is "print") and clicks the Print button. [Alternate Course A: No file name entered] [Alternate Course B: User clicks Cancel button]

8. The system exports the selected visualization type to a PDF with the entered file name at the selected file path.

Alternate Course A: No file name entered

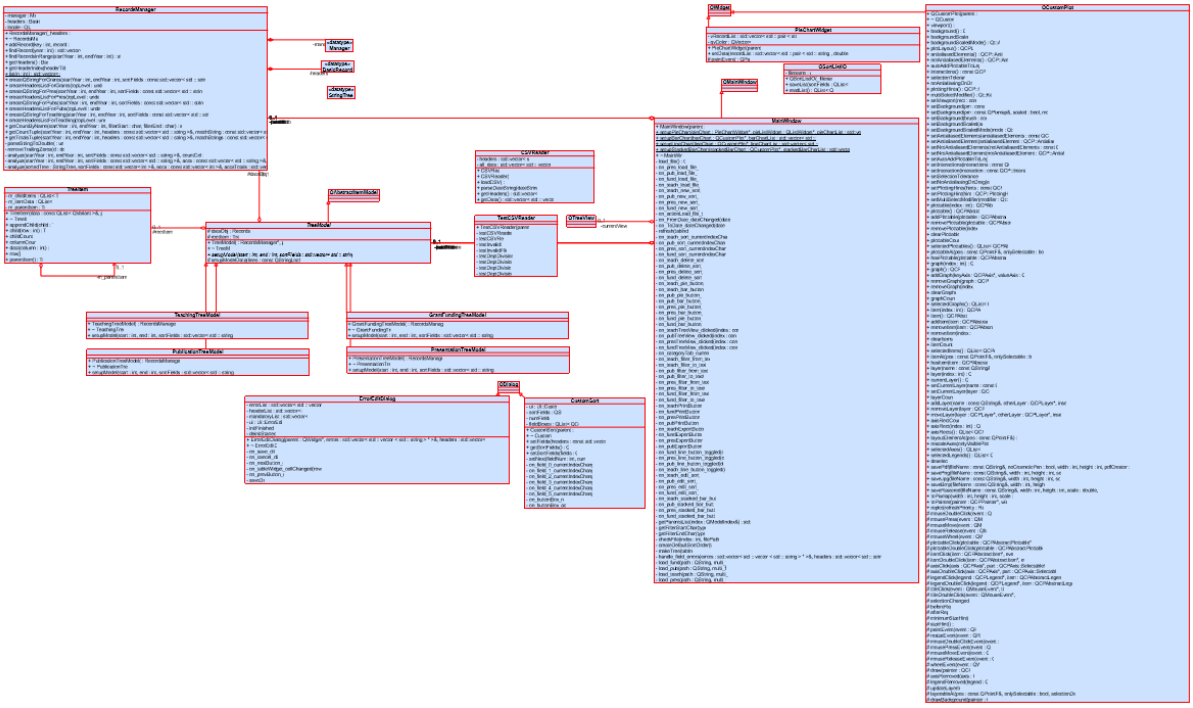
1. The system displays an error message.

2. The user accepts or closes the error message. [Return to Main Success Scenario step 3]

Alternate Course B: User clicks Cancel button

1. The system closes the file structure screen.

Class Diagram



Please see a pdf version in the uploaded package

Class Diagram Enhancements

The class diagram demonstrates the relationships between the different classes. In addition, each class shows their respective operations and attributes. The interdependency among the classes are intended to provide consistency in the user experience and correspondence between the customer's requirements and use case.

Rationale

Please see below regarding specific decisions made in the classes. The current design allows for data encapsulation, cohesion, and maintainability. Specifically,, the modular nature of the structure means that while classes depend on each other, the overall design is simplified so that each module contains all the methods needed to handle one aspect of the application. Specifically, MainWindow calls methods from other classes and brings all the classes together. This design was used because it allows for implementation, consistency, and maintenance. In addition, certain classes implement others as a way to simplify and allow for consistent design and functionality. Specifically, TreeItem has four subclasses - the different types of dashboards. These implement TreeItem and are used by TreeModel to create a data structure to organize the information. Overall, the design and organizational structure was considered when deciding how these classes should communicate with each other.

Changes from Phase I to II include the addition of operations and classes such as MySortProxyFilterModel and saveProgramState(). These were added to allow for new features that would fit the user requirements.

DATABASES

CSVReader

CSVReader is used to read and parse through the data from a CSV file. It is used primarily in the MainWindow.

QSortListIO

QSortList is used to read and write data from and into the file. It is used primarily in the MainWindow.

RecordsManager

The RecordsManager is responsible for creating records using data from the CSV files. These records are used by the system to create various types of representations of the information. It is also used by the TreeModel.

DATAMODEL

TreeModel

TreeModel uses RecordsManager to construct a linked list given the data that's to be in it. This is then passed to the GUI. Within TreeModel, TreeItem is used as a pointer to the first record. MainWindow uses the TreeModel to determine the correct information to display. In addition to being a subclass of QAbstractItemModel, it is used by GranFundinTreeModel, PresentationTreeModel, PublicationTreeModel, and TeachingTreeModel.

TreeItem

TreeItem is used by TreeModel to store information for the records. A linked list is created which stores the data provided in an efficient manner.

GUI

MainWindow

As a subclass of QMainWindow, MainWindow inherits the properties of the superclass. This allows it to access the QT library features, which allows for consistency in the interface. MainWindow's purpose is to set up the user interface. Specifically, this means that it controls the window's core functionality, including refreshing, loading files, handling errors, filtering, and displaying information through various different mediums.

Main
This class serves as the driver for the program, where it runs the application and MainWindow. If the test variable is true, it runs the tests and outputs the results to the log.

CustomSort
As a subclass of QDialog, CustomSort inherits the properties of the superclass. This allows it to take on properties which allow for integration with other GUI components. In addition, the CustomSort class allows the user to customize the data they want for representation. This is called in MainWindow when the method new_sort_clicked() is called for each Tree item.

ErrorEditDialog
As a subclass of QDialog, ErrorEditDialog inherits the properties of the superclass. This allows it to take on properties which allow for integration with other GUI components. In addition, the ErrorEditDialog class is intended to provide an opportunity to discard or edit mandatory fields which are missing.

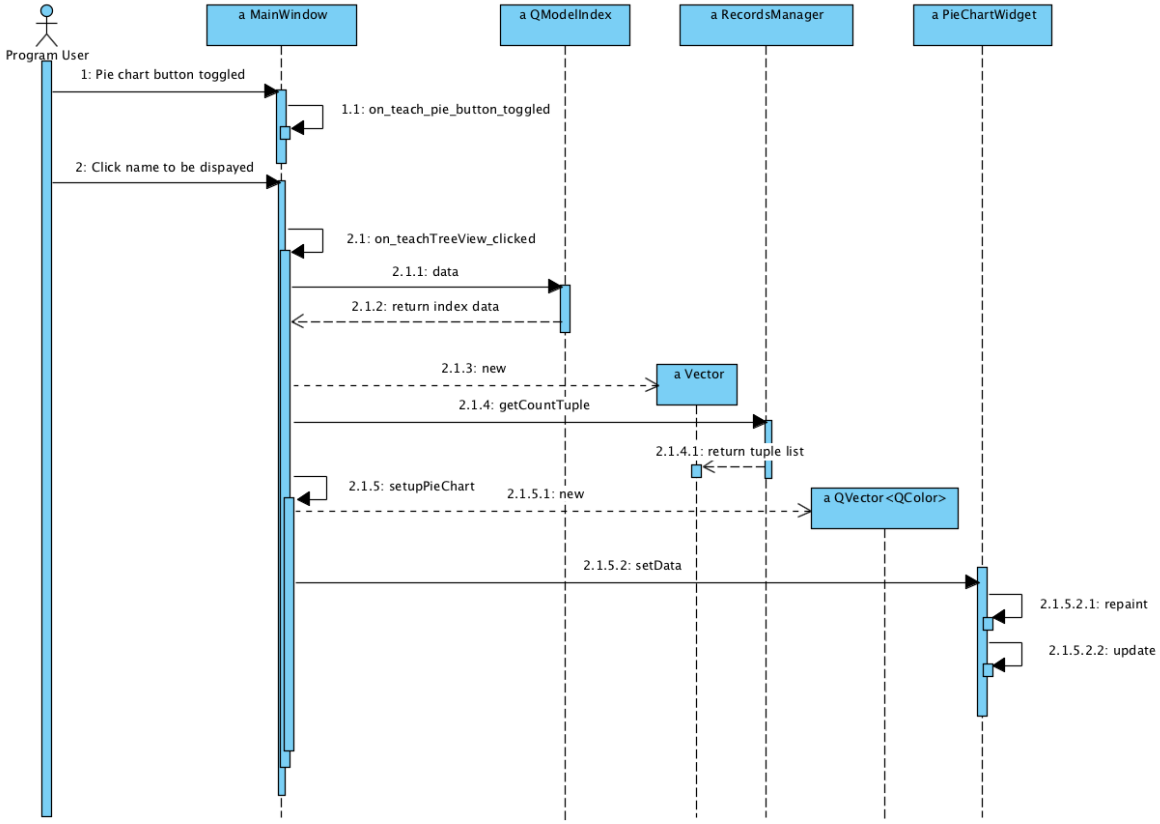
PieChartWidget
As a subclass of QWidget, PieChartWidget inherits the properties of the superclass. This allows it to access the QT library features, which allows for consistency in the interface. Specifically, PieChartWidget helps visualize the data stored in the TreeModel.

QCustomPlot
Defines the properties of the interface, with details such as the pen and plotting. These are then used to plot the data onto graphs, which are used in MainWindow.

Sequence Diagram

1. Sequence Diagram - Displaying a Teaching Pie Chart:

Below is the Sequence Diagram for the use case “Display Teaching Pie Chart”, including the main scenario and extensions. This use case assumes that you have already loaded a teaching excel file. Presented below you will also find explanations corresponding to the each steps in the scenario.



1.1 Program user toggles pie chart button

This step is represented by the program users first arrow, found in the top left corner of the diagram and entitled “Pie Chart Button Toggled”. It activates the MainWindow’s self-call to “on_teach_pie_button_toggled”. This function tells the ui to display the pie chart item from the teach_graph_stackWidget.

1.2 Program User selects a teacher's name

This step is represented by the program users second arrow entitled “Click name to be displayed”. It activates the MainWindow’s self-call “on_teachTreeView_clicked”, catalyzing the rest of the sequence.

1.3 System creates a list from the selected name

This step is accounted for in the first half of MainWindow “on_teachTreeView_clicked”. A QModelIndex is passed as a parameter to the function and from this parameter it extracts the data. It then proceeds to make a

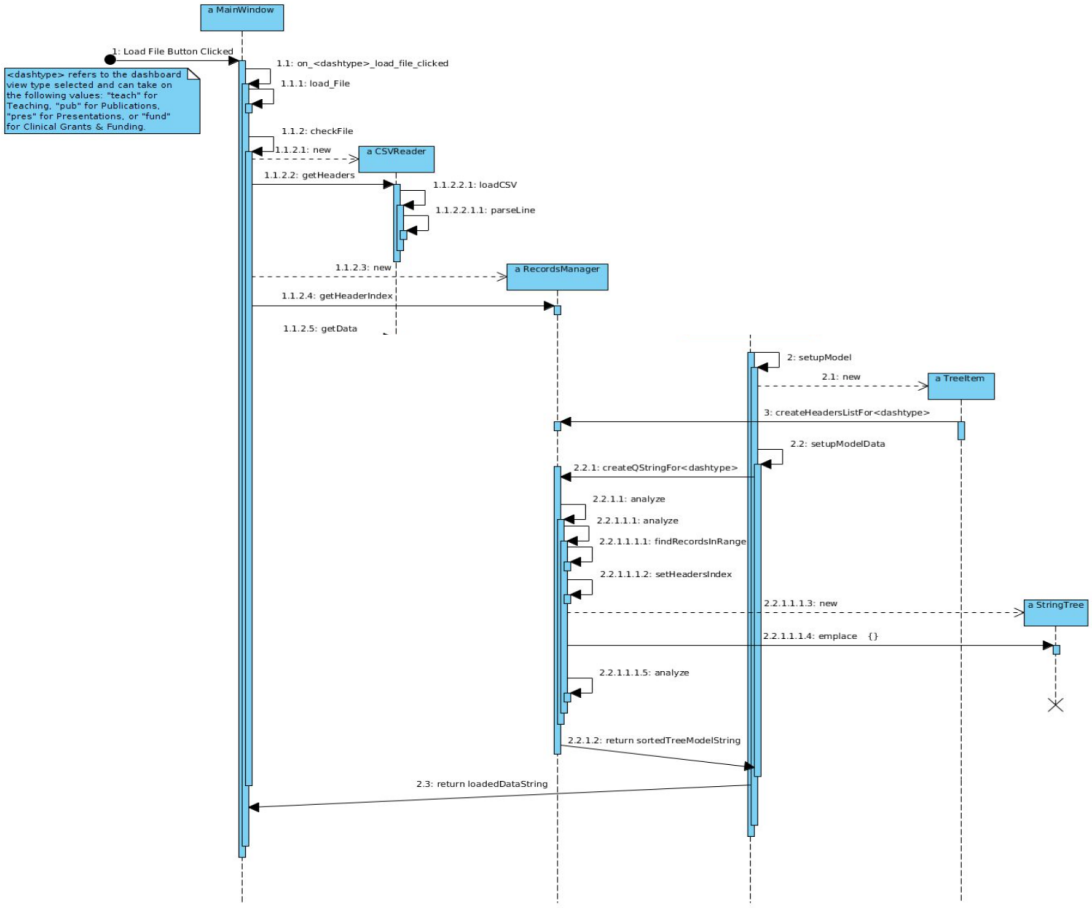
vector list of tuples, the data from these tuples are used in the creation of the pie chart.

1.4 System creates the pie chart

The function “setupPieChart” found within MainWindow “on_teachTreeView_clicked” handles setting up the pie chart. It creates a QVector of colors that it then passes to the function “pieChart->setData” which is a function from the class PieChartWidget. This function repaints the pie chart and updates it. The users sees this updated pie chart displayed.

2. Sequence diagram – loading data from file

Below is the sequence diagram for the use case scenario “Load Data from File”, including the main scenario and extensions. Presented below you will also find



explanations corresponding to the each steps in the scenario.

2.1 User clicks load button

This step is represented by the found message, an arrow with a dotted end in the top left of the diagram entitled “Load Data Button Click”. It activates the MainWindow self-call “on_<dashtype>_load_file_clicked”, catalyzing the entire sequence.

2.2 System displays file structure – user selects file

These two steps are contained in a single activation. MainWindow self-calls the "loadFile" method, which opens a dialog box for the user to select the desired CSV file. It then returns the file path and, if successful, self-calls the "checkFile" to make sure the CSV file type is compatible with the dashboard view type. It is natural to combine these steps in such a way because the user will typically click the "Load Data" button and select an appropriate CSV file. If this is not the case (i.e. the user cannot find the desired file or accidentally closes the dialog box) then it does not make sense to try and load data.

2.3 System verifies file is of proper type

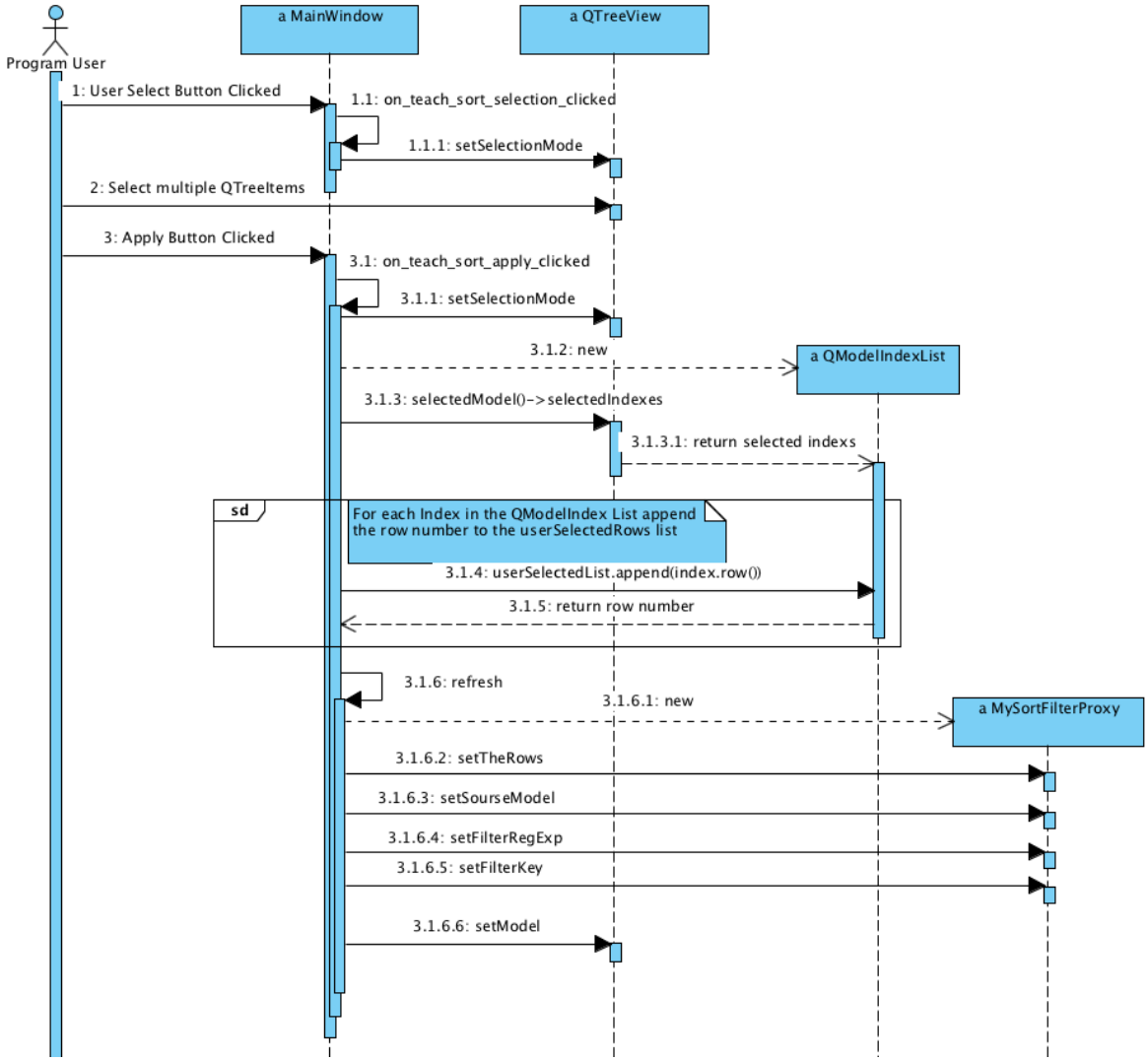
This step is accounted for in the MainWindow "checkFile" method self-call, which first checks that the filepath is valid and the filetype matches the appropriate dashboard view. If either of these conditions fail, MainWindow displays the following message: "Not a valid <dashtype> file". If however the conditions are met, MainWindow proceeds with loading the data. This approach was chosen in order to allow the program to terminate gracefully and give the user the option to select a new (proper) CSV file. This design decision stemmed from the conclusion that a simple file path error should not crash the entire application.

2.4 System loads data from file

This last step is the most resource-consuming of all and is thus described by the bulk of the sequence diagram. The first activation creates a new CSVReader object which parses the file, while the next call returns the headers. Then MainWindow creates a new RecordsManager object and gets the sorted header indices, which in general are different for each of the four file types. Next the data is retrieved from CSVReader and stored into RecordsManager. However the data is not sorted; to do this MainWindow creates a new <dashtype>TreeModel, which when activated self-calls its "setupModel" and creates a new TreeItem (that is, the root node) and its appropriate header list. The data is now ready to be sorted and so "setupModel" calls the RecordsManager "analyze" function to accomplish this task. The "analyze" function has many self-calls and is internally overridden; it also calls "findRecordsInRange" to filter by date and creates an instance of the StringTree helper class. It builds the data string to return as well as any necessary accumulators for the dashboard view. The result is a TreeModel filled with the sorted data, ready to be used for visualization. The classes are designed to maximize cohesion and minimize coupling to support the general goals of readability and maintainability. CSVReader and StringTree's destructor methods are called as they are no longer of us.

3. Sequence Diagram - Filter by User Selected List:

Below is the Sequence Diagram for the use case "Filter by User Selected List", including the main scenario and extensions. This use case assumes that you have already loaded a teaching excel file. Presented below you will also find explanations corresponding to the each steps in the scenario.



3.1 Program user clicks user select button

This step is represented by the program users first arrow, found in the top left corner of the diagram and entitled “User Select Button Clicked”. It activates the MainWindow’s self-call to “on_teach_sort_selection_clicked”. This function changes the teachTreeViews selection mode to allow for multiple selections.

3.2 Program User selects the items to include in the filter

The user selects the items they wish to include in the filter.

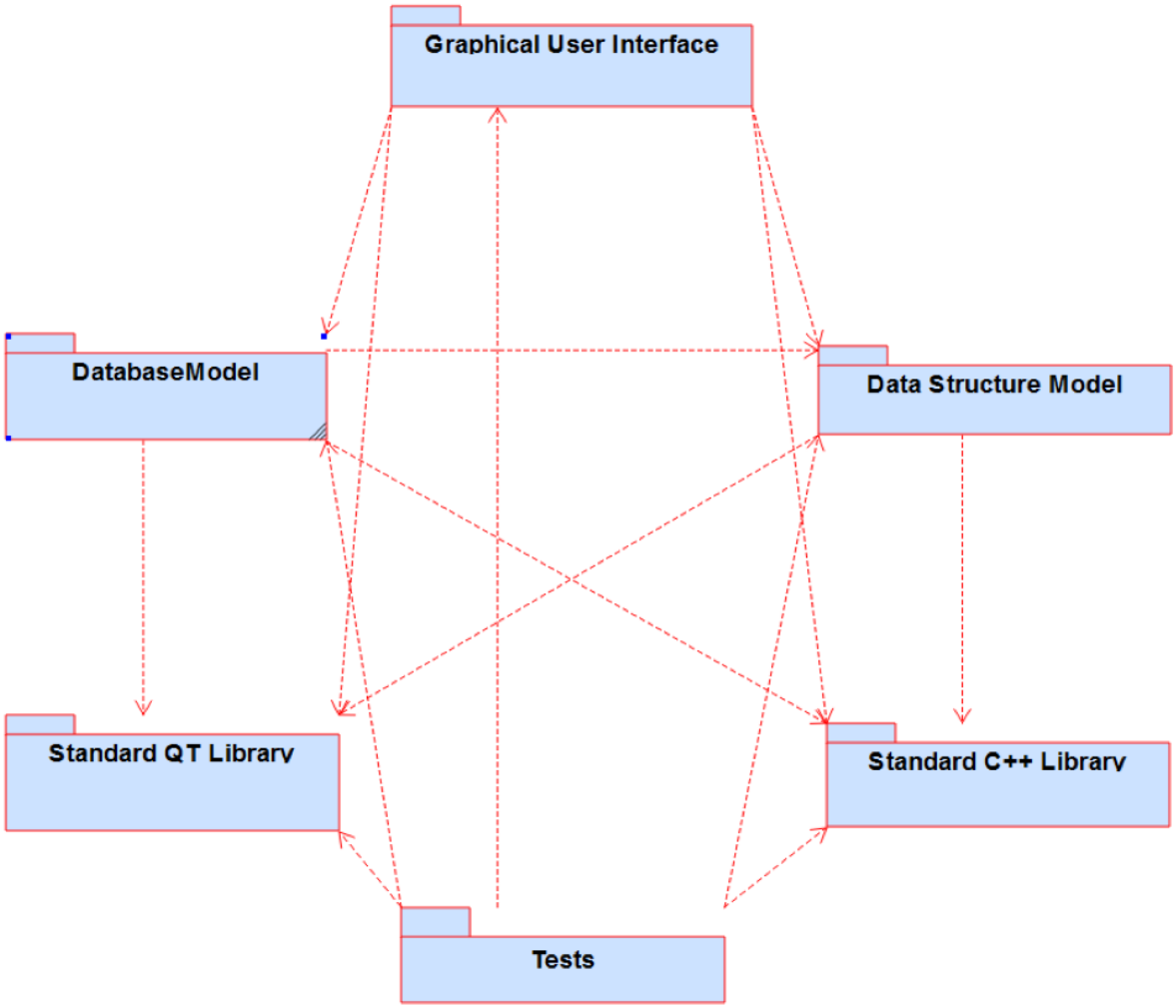
3.3 User clicks the apply button

The user clicks the apply button which activates the MainWindow’s self-call to “on_teach_sort_apply_clicked”. From within “On_teach_sort_apply_clicked” the rest of the events to filter by user selection are handled. First it calls “selectedIndexes” to get the data for the items that the user selected. It then stores the row numbers in the MainWindow’s QList<int> userSelectedRow object. Once the data is stored in the list the MainWindow’s refresh function is called.

3.4 The refresh function

From within the refresh function a new MySortFilterProxy object is created. We pass the MySortFilterProxy object the MainWindow's currentTree and the list of row numbers the user selected. We then call "setFilterRegExp" which uses our overwritten function "filterAcceptsRow" to filter out all items except the ones in the list that we provided. We then set the currentView to this filtered model. This is then displayed in the UI.

Package Diagram



In the above package diagram, the dotted arrows denote the dependencies. These can be read as <source package> depends on <target package>.

Graphical User Interface

This package is responsible for containing the classes that are used by the user interface. Specifically, these are the MainWindow, PieChartWidget, and CustomSort classes. During user interaction, these classes are responsible for controlling the graphical components of the program. In addition, MainWindow creates new databases and models every time a new

CSV file is loaded. Given the nature of these classes, they rely on the standard QT and C++ libraries.

Database Model

This package handles the CSVReader and RecordsManager claase which are used to crate and use the databases storing the loaded CSV files. CSVReader parses through the file and relies on the standard C++ library for strings and vectors. RecordsManager depends on the data structure model and the standard QT and C++ libraries. This is because the RecordsManager must create the appropriate database for each dashboard view type.

Data Structure Model

Within this package are the TreeModel, TreeItem, and implementation classes. The TreeModel class uses the TreeItem class to create a data structure and the implementations for each of the dashboard views. These include: GrantFundingTreeModel, PresentationTreeMode, PublicationTreeModel, and TeachingTreeModel. TreeModel uses RecordsManager when building itself, so the database and data structure model packages depend on each other. However, this interdepency is localized.

Standard QT and C++ Libraries

Stand-alone packages are already well-documented, and not dependant on other packages in the application in any way.

Tests

Tests on operations are dependent on the respective packages they are testing. In addition, they are dependent on the C++ and standard QT libraries.

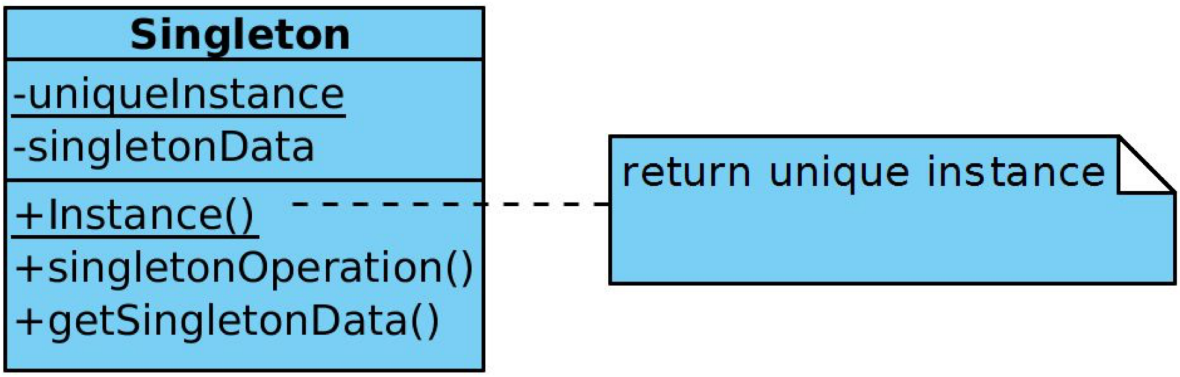
Design Patterns

Our teams improved Peachy Galaxy Application makes use of two important design patterns. The Singleton pattern is used for the MainWindow class and the Prototype pattern is used for the TreeModel class.

Singleton Pattern

The goal of the Singleton pattern is to ensure that a class only ever has one instance created and to provide a global point of access to it. While a singleton can be created by using a single global variable, it does not prevent someone from creating several new instances of the class. A better way to implement the singleton pattern is to have the class itself be responsible for keeping track of its single instance. This way, the class itself can be used to ensure that no more than once instance of the class can ever be created and the class itself can act as a way to access the single instance.

The structure of a singleton pattern is illustrated below:



The main benefits of using the singleton pattern in a system are that it provides:

- **Controlled Access to the sole instance:** because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
- **Reduced name space:** the Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
- **Permits refinement of operations and representation:** the Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. One can configure the application with an instance of the class you need at run-time.
- **Permits a variable number of instances:** Permits a variable number of instances: this pattern makes it easy to change one's mind and allow more than one instance of the Singleton class. Moreover, one can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- **More Flexible the class operations:** another way to package a Singleton's functionality is to use class operations such as a static member function in C++. However this technique makes it hard to change a design to allow more than one

instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

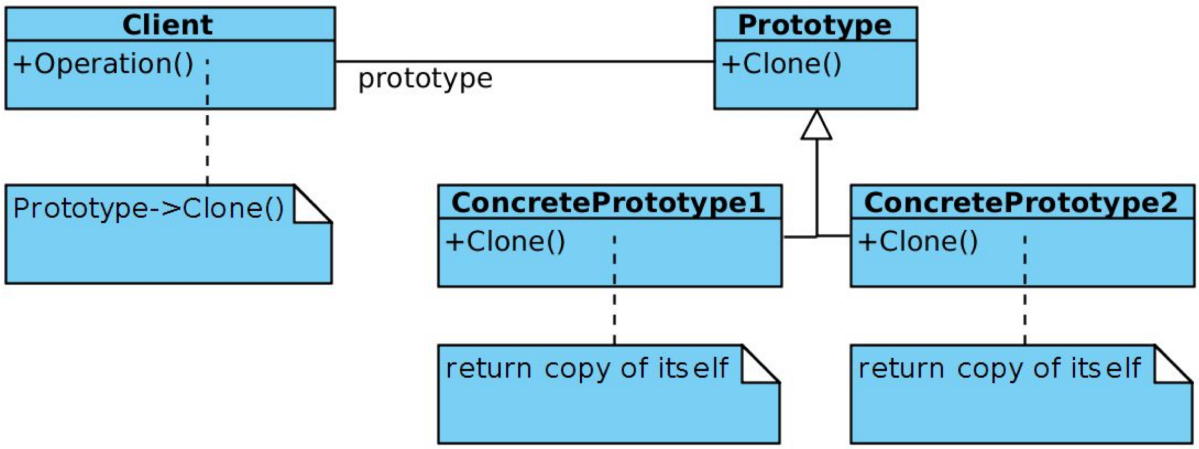
The most important issue with the Singleton pattern we consider when implementing it is ensuring a unique instance. The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation (that is, either a static member function or a class method) that guarantees only one instance is created. This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a singleton is created and initialized before its first use. One can define the class operation in C++ with a static member function Instance of the Singleton class. Singleton also defines a static member variable uniqueInstance that contains a pointer to its unique instance.

Although we do not require a variable number of instances, we acknowledge our flexibility in modifying our instance should the need arise. We use the Singleton design pattern in the implementation of MainWindow, which controls the runtime behaviour of the graphical user interface. This design decision is a result of our belief that there must be exactly one instance of this class and it must be accessible to clients from a well-known access point.

In our implementation of the Singleton we just used a single global variable to make our object accessible. While this solution is not as ideal as making the class itself responsible for keeping track of its sole instance, it is simple to implement and easier for people with a less technical background to understand.

Prototype Pattern

The goal of using a Prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. Suppose our system has many objects which, although differ slightly from each other, exhibit almost identical behaviour. We know object composition is a flexible alternative to subclassing. We would like our framework to take advantage of this to parameterize instances based on the type of class it will create. The solution to this dilemna lies in making the framework create a new instance by copying or "cloning" an instance of the desired class. We call this instance a prototype and depict its structure below.



The Prototype pattern shares many of the same consequences with other creational design patterns: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification. However, their are additional benifits unique to Prototype:

- **Adding and removing products at run-time:** prototypes allow the incorporation of a new concrete product class into a system simply by registering a prototypical instance with the client. This is slightly more flexible than other creational patterns because a client can install and remove prototypes at run-time.
    - **Specifying new objects by varying values:** highly dynamic systems permit new behaviour through object composition—by specifying values for an object's variables, for example—and not by defining new classes. One effectively defines new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype. This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs.
- **Specifying new objects by varying structure:** many applications build objects from parts and subparts. Our graphical user interface, for example, is built from widgets, dialog boxes, radio buttons, etc. For convenience, such applications often let one instantiate complex, user-defined structures, say, to use a specific widget again and again. The Prototype pattern supports this as well. We simply add this widget as a prototype to the palette of available graphical user interface elements.
- **Reduced subclassing:** other alternatives to the Prototype pattern, such as the Factory Method, often produce a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern allows one to clone a prototype instead of asking a factory method to make a new object. Hence one des not need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects.
- **Configure an application with classes dynamically:** some run-time environments, like that of our application, enables one to load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++. An application that wants to create instances of a dynamically

loaded class will not be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager. Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally.

Our decision to use the Prototype pattern follows from the conclusion that our system should be independent of how its products are created, composed, and represented. We consider it the best choice for the TreeModel class, the instantiation of which is specified at run-time by dynamic loading. In general we like to avoid building a class hierarchy of factories that parallels the class hierarchy of products, which is why we do not opt for the factory method. Furthermore, our MainWindow, PieChartWidget, and CustomSort classe instances can have one of only a few different combinations of state. It is thus more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. For example, adding Clone is difficult when the classes under consideration already exist. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references.

For our TreeModel class, we use what is referred to as a prototype manager. When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), we keep a registry of available prototypes. Clients will not manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. Unfortunately our prototype classes do not define operations for (re)setting key pieces of state. If not, then you may have to introduce an initialization operation that takes initialization parameters as arguments and sets the clone's internal state accordingly. An example of this is the setupModel() operation in TreeItem.

Implementation in C++

Main Window / Dashboard

The MainWindow class was designed as one large singleton class and oversees all runtime behaviour of the applications GUI. Our application uses a single global variable to make the object accessible and only ever has one sole instance of the class initialized.

Along with overseeing behaviour caused by user interaction with the GUI, the MainWindow class overrides some Qt methods to allow for the Drag and Drop of files and Keyboard shortcuts. Specifically, for Drag and Drop the MainWindow class implementes the “dragEnterEvent” and the “dropEvent” methods to get file path for a dropped file and calls the correct load file method depending on which category is selected (load_teach, load_pub etc.). For the keyboard shortcuts the MainWindow class implemented Qt’s “keyPressEvent” and “keyReleaseEvent” methods to keep track of which keys have been pressed and which action to trigger.

Sort Orders

The CustomSort class is responsible for the creation and editing of sort orders in the application. When the user presses the “New Sort” or “Edit Sort” button, the MainWindow class creates a brand-new instance of the CustomSort class and passes in a list of Fields that the user should be able to select from. The user can then select which Fields they want to sort by and when the save button is clicked the sort order returned back to the MainWindow and saved to a text file using an instance of the QSortListIO class and its “saveList” method.

QSortListIO simply takes a list of all the sort orders and uses QT’s built in QDataSteam to write the sort orders into a text file. Then on relaunch of the program QSortListIO’s “readList” method is called to read back the sort orders from the text file.

Visualizing Data

Three out of the four different types of charts in our program are implemented using the QCustomPlot Library. While the remaining chart, the pie chart is implemented using the PieChartWidget class.

When a user clicks on a datapoint in the MainWindow’s treeView, the corresponding on_treeView_clicked method is called. This method gets the data item that was clicked and its child data items and creates a vector of string/double pairs to be plotted on the graph. The on_treeView_clicked method then calls the correct setup method for each of the 4 different chart types.

For the each of the LineChart, BarChart and Stacked BarChart the setup method takes the vector of string/double pairs and graphs them using an instance of QCustomPlot. For the PieChart the setup method takes the vector of string/double pairs and graphs them using an instance of the PieChartWidget class.

Loading Files

To read in the CSV file the system MainWindow uses an instance of the CSV reader class. Firstly, the MainWindow creates a new instance of CSVReader and passes in the file path as parameter. The constructor for CSVReader then calls the loadCSV method to read in all the data from the file.

The MainWindow class then uses the CSVReaders getHeaders() and getData() methods to get the headers and data from the csv file and passes it into RecordManager instance we saves the data in a Tree to be displayed in the MainWindows TreeView.

Save State

In order to save the last file opened by our program, when the application closes the MainWindows “closeEvent” method is called. This method simply calls the MainWindow’s “saveProgramState” method and accepts the close event. The “saveProgramState” method uses an instance of Qt’s QDataStream to save the file path of the most recently opened file and the index of tab of the category that is currently open.

In order to load the state back into the program, the MainWindow classes constructor creates a different instance of Qt’s QDataStream and reads back in the file path and category tab index and calls the load method for that category to load in the csv file.

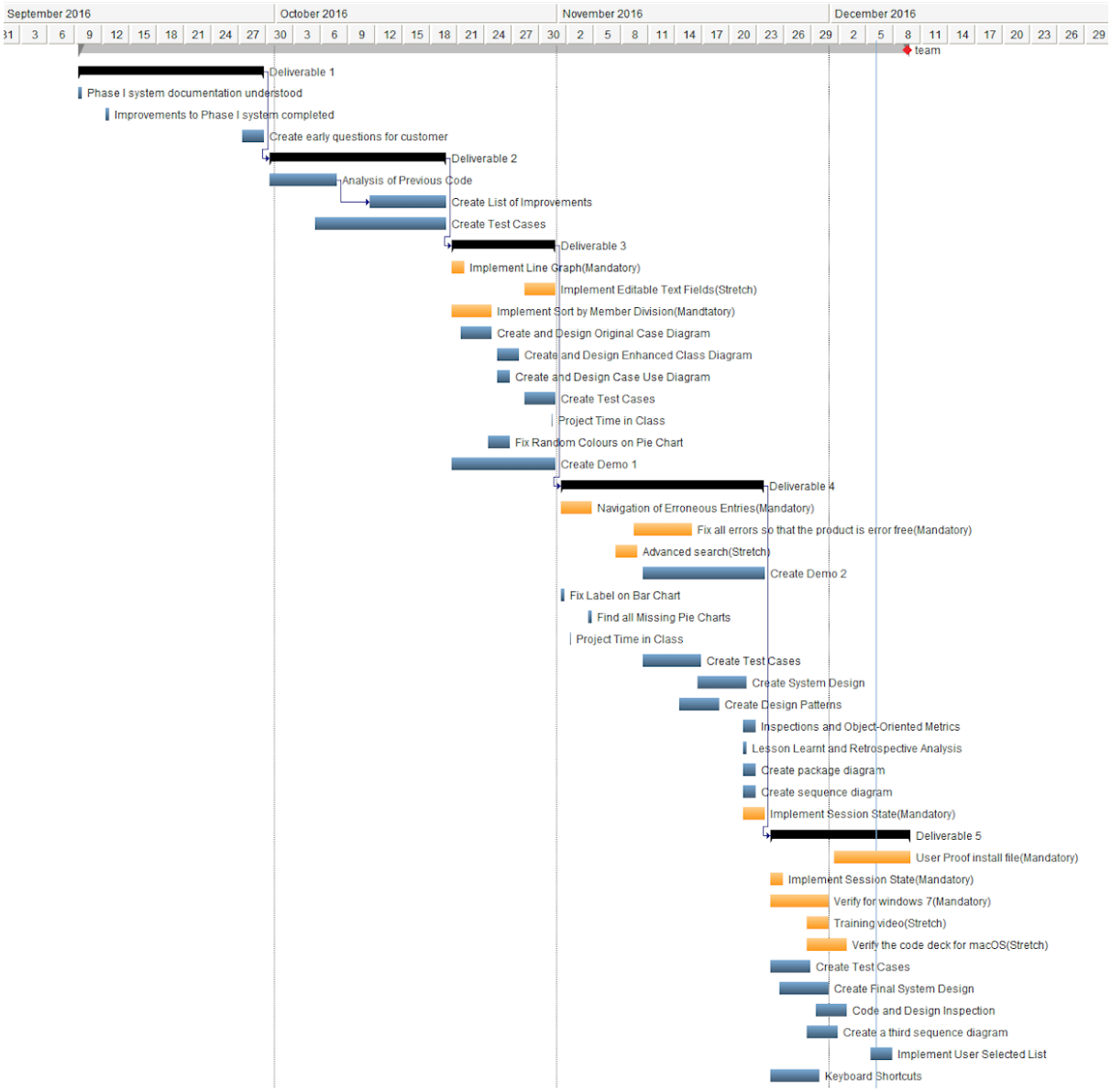
Development Plans

Timeline

List of deliverables with their respective tasks accompanied with duration of the task.

		Name	Duration	Start	Finish	Predecessors
0		**team**	**66d?**	**09/09/2016**	**12/09/2016**	
1		**Deliverable 1**	**15d?**	**09/09/2016**	**09/29/2016**	
2		Phase I system documentation understood	1d?	09/09/2016	09/09/2016	
3		Improvements to Phase I system completed	1d?	09/12/2016	09/12/2016	
4		Create early questions for customer	3d?	09/27/2016	09/29/2016	
5		**Deliverable 2**	**14d?**	**09/30/2016**	**10/19/2016**	**1**
6		Analysis of Previous Code	6d?	09/30/2016	10/07/2016	
7		Create List of Improvements	7d?	10/11/2016	10/19/2016	6
8		Create Test Cases	11d?	10/05/2016	10/19/2016	
9		**Deliverable 3**	**8d?**	**10/20/2016**	**10/31/2016**	**5**
10		Implement Line Graph(Mandatory)	2d?	10/20/2016	10/21/2016	
11		Implement Editable Text Fields(Stretch)	2d?	10/28/2016	10/31/2016	
12		Implement Sort by Member Division(Mandtatory)	3d?	10/20/2016	10/24/2016	
13		Create and Design Original Case Diagram	2d?	10/21/2016	10/24/2016	
14		Create and Design Enhanced Class Diagram	3d?	10/25/2016	10/27/2016	
15		Create and Design Case Use Diagram	2d?	10/25/2016	10/26/2016	
16		Create Test Cases	2d?	10/28/2016	10/31/2016	
17		Project Time in Class	2h?	10/31/2016	10/31/2016	
18		Fix Random Colours on Pie Chart	3d?	10/24/2016	10/26/2016	
19		Create Demo 1	8d?	10/20/2016	10/31/2016	
20		**Deliverable 4**	**17d?**	**11/01/2016**	**11/23/2016**	**9**
21		Navigation of Erroneous Entries(Mandatory)	4d?	11/01/2016	11/04/2016	
22		Fix all errors so that the product is error free(Mandatory)	5d?	11/09/2016	11/15/2016	
23		Advanced search(Stretch)	3d?	11/07/2016	11/09/2016	
24		Create Demo 2	10d?	11/10/2016	11/23/2016	
25		Fix Label on Bar Chart	1d?	11/01/2016	11/01/2016	
26		Find all Missing Pie Charts	1d?	11/04/2016	11/04/2016	
27		Project Time in Class	1h?	11/02/2016	11/02/2016	
28		Create Test Cases	5d?	11/10/2016	11/16/2016	
29		Create System Design	4d?	11/16/2016	11/21/2016	
30		Create Design Patterns	5d?	11/14/2016	11/18/2016	
31		Inspections and Object-Oriented Metrics	2d?	11/21/2016	11/22/2016	
32		Lesson Learnt and Retrospective Analysis	1d?	11/21/2016	11/21/2016	
33		Create package diagram	2d?	11/21/2016	11/22/2016	
34		Create sequence diagram	2d?	11/21/2016	11/22/2016	
35		Implement Session State(Mandatory)	3d?	11/21/2016	11/23/2016	
36		**Deliverable 5**	**12d?**	**11/24/2016**	**12/09/2016**	**20**
37		User Proof install file(Mandatory)	7d?	12/01/2016	12/09/2016	
38		Implement Session State(Mandatory)	2d?	11/24/2016	11/25/2016	
39		Verify for windows 7(Mandatory)	5d?	11/24/2016	11/30/2016	
40		Training video(Stretch)	3d?	11/28/2016	11/30/2016	
41		Verify the code deck for macOS(Stretch)	5d?	11/28/2016	12/02/2016	
42		Create Test Cases	3d?	11/24/2016	11/28/2016	
43		Create Final System Design	4d?	11/25/2016	11/30/2016	
44		Code and Design Inspection	4d?	11/29/2016	12/02/2016	
45		Create a third sequence diagram	4d?	11/28/2016	12/01/2016	
46		Implement User Selected List	3d?	12/05/2016	12/07/2016	
47		Keyboard Shortcuts	4d?	11/24/2016	11/29/2016	

Gantt Chart which show the different tasks within each deliverable. The colours of deliverable, general tasks, mandatory/stretch tasks are black, blue, and orange respectively.



Agent Task View

Tasks/Agents	Task Type:	Haris	Leroi	Ryan	Stepan	Novia	Eric	Status of Task:
Deliverable 1								
Phase 1 system documents understood		X	X	X	X	X	X	Complete
Improvements to Phase 1 system complete		X						Complete
Create early questions for customers		X	X	X	X	X	X	Complete
Deliverable 2								
Analysis of Previous code		X	X	X	X	X	X	Complete
Create list of improvements		X	X	X	X	X	X	Complete
Create test cases	C++			X			X	Complete
Deliverable 3								
Implement line graph (mandatory)	C++			X			X	Complete
Create agent task view							X	Complete
Implement Editable text fields (stretch)	C++				X			Complete
Implement sort by member division (mandatory)	C++			X				Complete
Create and design original case diagram			X			X		Complete
Create and design enhanced class diagram			X					Complete
Create and design case use diagram						X		Complete
Create test cases	C++			X				Complete
Fix Random colours on Pie Chart	C++	X					X	Complete
Create timeline		X						Complete
Create demo 1	C++			X				Complete
Deliverable 4								
Navigation of erroneous entries (mandatory)	C++			X				Complete
Create demo 2	C++	X		X				Complete
Fix label on bar chart	C++			X				Complete
Create test cases	C++			X				Complete
Create system design					X			Complete
Create sequence diagram							X	Complete
Create timeline		X						Complete
Create package diagram			X			X		Complete
Verify for windows 7 (mandatory)				X				Complete
Create agent task view		X					X	Complete
User proof install file (mandatory)				X				Complete
Deliverable 5								
Inspection and object-oriented metrics			X					Complete
Advanced search (stretch)	C++		X	X				Complete
Fix all errors so that the product is error free (mandatory)	C++	X	X	X	X	X	X	Complete
Lesson learn and retrospective analysis				X		X		Complete
Implement Session State (mandatory)	C++	X		X			X	Complete
User proof install file (mandatory)		X	X	X				Complete
Verify for windows 7 (mandatory)			X		X			Complete
Training video (stretch)		X				X		Complete
Verify the code deck for macOS (stretch)		X	X	X				Complete
Create test cases	C++			X	X		X	Complete
Create final system design			X			X		Complete
Create a third sequence diagram							X	Complete
Implement User Selected List	C++						X	Complete
Keyboard Shortcuts	C++			X				Complete

Modified Agent Task View (reflecting changes in task allocation):

Below is an agent task view that shows the changes in task allocation throughout the project's development. Some planned task allocation remain unchanged throughout the development process while many other tasks needed reassignment and allocation to accommodate unforeseeable changes/challenges.

**********	**********	**********	**********	**********	**********	**********	**********	**********
Legend:								
Originally Assigned to the Task								
Actually Implemented the Task								
**********	**********	**********	**********	**********	**********	**********	**********	**********
Tasks/Agents	**Task Type:**	Haris	Leroi	Ryan	Stepan	Novia	Eric	**Status of Task:**
Deliverable 1	----------	----------	----------	----------	----------	----------	----------	
Phase 1 system documents understood		X	X	X	X	X	X	Complete
Improvements to Phase 1 system complete		X						Complete
Create early questions for customers		X	X	X	X	X	X	Complete
Deliverable 2	----------	----------	----------	----------	----------	----------	----------	
Analysis of Previous code		X	X	X	X	X	X	Complete
Create list of improvements		X	X	X	X	X	X	Complete
Create test cases	C++			X			X	Complete
Deliverable 3	----------	----------	----------	----------	----------	----------	----------	
Implement line graph (mandatory)	C++			X			X	Complete
Create agent task view							X	Complete
Implement Editable text fields (stretch)	C++				X		X	Complete
Implement sort by member division (mandatory)	C++			X	X			Complete
Create and design original case diagram			X			X		Complete
Create and design enhanced class diagram			X					Complete
Create and design case use diagram						X		Complete
Create test cases	C++			X				Complete
Fix Random colours on Pie Chart	C++	X					X	Complete
Create timeline		X						Complete
Create demo 1	C++			X				Complete
Deliverable 4	----------	----------	----------	----------	----------	----------	----------	
Navigation of erroneous entries (mandatory)	C++			X				Complete
Create demo 2	C++	X	X	X		X	X	Complete
Fix label on bar chart	C++			X				Complete
Create test cases	C++			X				Complete
Create system design					X			Complete
Create sequence diagram		X					X	Complete
Create timeline		X						Complete
Create package diagram			X			X		Complete
Verify for windows 7 (mandatory)				X				Complete
Create agent task view		X					X	Complete
User proof install file (mandatory)				X				Complete
Deliverable 5	----------	----------	----------	----------	----------	----------	----------	
Inspection and object-oriented metrics	C++		X				X	Complete
Advanced search (stretch)	C++		X	X	X			Complete
Fix all errors so that the product is error free (mandatory)	C++	X	X	X	X	X	X	In Progress
Lesson learn and retrospective analysis		X	X	X		X		Complete
Implement Session State (mandatory)	C++	X		X			X	Complete
User proof install file (mandatory)			X	X				Complete
Verify for windows 7 (mandatory)			X	X	X		X	Complete
Training video (stretch)		X				X		Complete
Verify the code deck for macOS (stretch)		X	X	X				Complete
Create test cases	C++			X	X		X	Complete
Create final system design			X			X		Complete
Code and design inspection		X					X	Complete
Implement User Selected List	C++						X	Complete
User Short Cuts	C++			X				Complete
User Guide							X	Complete

Lesson Learned

Coordinating group work

It is difficult to coordinate group work, especially to share documents with each other. However, we learned that by using modern tools such as GitHub, and cloud services to share document it helps to improve team efficiency. Additionally, we found that messaging services and VoIP programs such as Facebook Messenger and Skype help to make the coordination and communication between such a large group much easier. At any time, we can easily check the progress of our project, and share information with everyone in the group. Also, we can make the code repository and group chats be private; therefore, these tools also keep all files and ideas in a relatively safe place.

Problems with UML

UML provides many kinds of diagrams to help plan and design a project. We used use-case diagrams and class diagrams in our project to show the various relations between different components. A use case diagram illustrates a unit of functionality provided by the system. The main purpose of the use-case diagram is to help development teams visualize the functional requirements of a system. Use Case Diagrams make it easy to describe and show the implemented features to the customers even if they are lacking technical knowledge. However, use-case diagrams just can describe the functional requirements of a user, but cannot do anything to describe non-functional requirements such as reliability, performance, and so on. In addition, the class diagram shows how to connect different entities (people, things, and data); in other words, it shows the static structures of the system. Therefore, when a programmer wants to change the code, it is figure out which file should be modified. However, for developers who do not understand UML the design can be difficult to understand at first glance. Because of this, when our group tried to make our UML diagrams, we need to search for a lot of information from the web, and the Fowler's book on UML.

Working with QT and C++

While some of our group already had experience with C++ going into the project learning the ins and outs of the Qt API and Qt Standard Library took a while. But in learning how to use Qt it also helped us better understand some of the fundamental characteristics of C++ such as inheritance and OOP. Throughout this semester of using Qt for the first time we learned that:

- Qt is a language extension based on C ++

Qt itself has inherited the C++ fast, simple, object-oriented design and builds on that to provide many other advantages

- Qt has a very good portability

Qt not only can run in Linux, also can run on Windows, Mac OS X and many other different platforms. This means that applications written with Qt can run on multiple platforms with very little or no modification.

- Qt support cross-platform build (Cross-Platform Builds)

Writing software for multiple platforms can tedious and buggy at any time. This is especially when different compilers and platform combinations require several files to be compiled. With the built in Qmake tool, Qt is well positioned to meet this challenge. This tool can be used to generate the correct documentation for each different target platform.

- Qt to use simple and efficient

Qt's has a unified, cross-platform APIs which allows programmers to focus on the core implementation of a software project without having to worry about maintaining and managing multiple versions of the existing application's infrastructure. So Qt developers only need to learn a single API to write an application that can be run on several different systems.

Inspection Documents

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.

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

Classes inspected: TreeModel, CSVReader, MainWindow

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 classes CSVReader, TreeModel, and MainWindow classes can be found in the attached class diagram found in this document.*
Comment on your findings: *All methods and attributes for CSVReader, TreeModel, and MainWindow can be found in the class diagram.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Compare functionality and performance with respect to their intended purpose according to the dashboard summary requirements.*
Comment on your findings: *Implementations of the TreeModel class could be simplified. Rather than copying the different functions for each of the four implementations, the code*

could be simplified to allow for just one occurrence of each function that could be used by each of the entities. This would simplify the current complexity.

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: *Determine the functionality of each method and compare it with other methods that serve the class. Specifically, the use of the return values and presence of member variables will be observed.*

Comment on your findings: *TreeModel implements the TreeItem Class. The setupBarChart(QCustomPlot *barChart, std::vector<std::pair<std::string, double>>barChartList) method in MainWindow is encapsulated in MainWindow.*

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: *Determine the number of includes, global variables, and function calls*
Comment on your findings: *TreeModel depends on the TreeItem class. publicationTree, teachingTree, presentationTree, fundingTree implement the TreeModel class. TestCSVReader depends on the CSVReader 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: *Determine how the different cases are separated.*
Comment on your findings: *Distinct separation of the groups in the tree, items in the tree, data, and index. Represented in the MainWindow class: case FUNDING, case PRESENTATIONS, case PUBLICATIONS, case TEACH*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Determine if there are private, protected, and public methods and if they make sense.*
Comment on your findings: *No methods contain public or private designations in front them in CSVReader, TreeModel, or MainWindow.*

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: *Determining what the class could be used for with slight modifications*
Comment on your findings: *TreeModel can be constantly reused, especially since the switch cases differentiate the different dashboard types. CSVReader can be constantly reused.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Determine the size and ease of understanding for the classes.*
Comment on your findings: *There are difficulties with the ease of understanding of the MainWindow class from the size of it. It has ~2300 line of code given the amount of repetition need*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Verify the presence and clarity of comments in the classes*
Comment on your findings: *CustomSort is light on comments, but it is easy enough to understand. MainWindow lacks appropriate commenting. As an example, on_fundTreeView_click() is fairly difficult to understand and lacks the appropriate level of commenting.*

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: *Determining what would be need to add a new dashboard type*
Comment on your findings: *Changing the GUI, writing a few more methods, and adding different cases would be easy to perform given the structure of the application.*

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: *Looking through the code for unnecessary nested loops or inefficient data structures.*
Comment on your findings: *CustomSort does not use nested loops and has limited calls to other functions. TreeModel is also efficient given the nature of the data structure. The setupModelData() function has a nested loop, but does not impact efficiency to a large extent.*

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: *Determining the depth of inheritance.*
Comment on your findings: *CustomSort inherits QDialog, which is a Qt class. TreeModel implementations inherits from TreeModel which inherits from QAbstractItemModel. Inheritance does not run very deep.*

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: *Determine the number of children classes*
Comment on your findings: *TreeModel is inherited by four subclasses for each dashboard type.*

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.
_____-
_____

(Note: expand here as necessary for each scenario)

TestCSVReader is not used by the end user.

Title: Read a CSV file with editable text fields to create the appropriate tree
Anticipated frequency of us: High
Starting Point: User clicks "Load File" and selects the appropriate CSV
Expected Outputs: A properly formatted TreeModel

Inputs:

- Start and end year of a date range: passed to RecordsManager
- A list of the fields (headers) to sort by: passed to RecordsManager
- A char range to filter the first header by: passed to RecordsManager

Walkthrough:

- A new TreeModel is created in MainWindow's method makeTree(int tabIndex)
- TreeModel.setupModel() is called by MainWindow, which passes the respective inputs
- RecordsManager.createQStringFor*() is called by setupModel(). This calls CSVReader that reads the CSV file and loads the data into vectors. These vectors are then passed to RecordsManager.createHeadersFor*(). This passes the user's inputs
- The returned QString and QList are passed to TreeModel.setupModelData()
- The QString is parsed through using setupModelData(), wich populates the QList of TreeItems
- Entries are stored relative to the parent TreeItem
- RecordManager calls the CSVReader to read and load the CSV file
- CustomSort is called when making changing to the sorting method
- The tree is displayed to the user using QTreeView.setModel() in the MainWindow

END.

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.

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

Classes inspected: TreeModel.cpp, CustomSort.cpp, TestCSVReader

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 classes CustomSort, TreeModel, and MainWindow classes can be found in the attached class diagram found in this document.*
Comment on your findings: *All methods and attributes for CustomSort, TreeModel, and TestCSVReader can be found in the class diagram.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Compare functionality and performance with respect to their intended purpose according to the dashboard summary requirements.*
Comment on your findings: *Classes function correctly and are accurately represented in the class diagram.*

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: *Determine the functionality of each method and compare it with other methods that serve the class. Specifically, the use of the return values and presence of member variables will be observed.*
Comment on your findings: *TreeModel and TestCSVReader are cohesive. While CustomSort has cohesion, the presence of the switch statements could be improved.*

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: *Determine the number of includes, global variables, and function calls*
Comment on your findings: *TreeModel depends on the TreeItem class. publicationTree, teachingTree, presentationTree, fundingTree implement the TreeModel class. TestCSVReader depends on the CSVReader 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: *Determine how the different cases are separated.*
Comment on your findings: *Cases are separated in TestCSVReader through test cases. The method setupModel() implements in four separate classes for each dashboard type.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Determine if there are private, protected, and public methods and if they make sense.*
Comment on your findings: *No methods contain public or private designations in front them in TestCSVReader, TreeModel, or CustomSort..*

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: *Determining what the class could be used for with slight modifications*
Comment on your findings: *TestCSVReader is a test class, and as a result, is fairly specific. TreeModel could be modified and reused by implementing a new model. CustomSort could be reused, but would require slight changes to the GUI and functionality. Specifically, there would need to be changes to the methods and cases.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Reading through the code and understanding the functionality*
Comment on your findings: *Function names are descriptive enough for users to understand.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Verify the presence of comments in the classes*
Comment on your findings: *Classes are documented and easy to understand. While CustomSort lacks comments, it is easy enough to understand.*

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: *Determining what would be need to add a new dashboard type*
Comment on your findings: *Changing the GUI, writing a few more methods, and adding different cases would be easy to perform given the structure of the application.*

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: *Looking through the code for unnecessary nested loops or inefficient data structures.*
Comment on your findings: *CustomSort does not use nested loops and has limited calls to other functions. TreeModel is also efficient given the nature of the data structure. The setupModelData() function has a nested loop, but does not impact efficiency to a large extent.*

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: *Determining the depth of inheritance.*
Comment on your findings: *CustomSort inherits QDialog, which is a Qt class. TreeModel implementations inherits from TreeModel which inherits from QAbstractItemModel. Inheritance does not run very deep.*

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: *Determine the number of children classes*
Comment on your findings: *TreeModel is inherited by four subclasses for each dashboard type.*

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:

vi) Title of scenario
vii) Anticipated frequency of use (high, normal, low)
viii) End-user trigger (starting point) for the scenario.
ix) Expected type of outputs.
x) 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.
_____-
_____

(Note: expand here as necessary for each scenario)

TestCSVReader is not used by the end user.

Title: Read a CSV file, allow for edits, and create the appropriate tree
Anticipated frequency of us: High
Starting Point: User clicks "Load File" and selects the appropriate CSV
Expected Outputs: A properly formatted TreeModel

Inputs:

- Start and end year of a date range: passed to RecordsManager
- A list of the fields (headers) to sort by: passed to RecordsManager
- A char range to filter the first header by: passed to RecordsManager

Walkthrough:

- A new TreeModel is created in MainWindow's method makeTree(int tabIndex)
- TreeModel.setupModel() is called by MainWindow, which passes the respective inputs
- RecordsManager.createQStringFor*() is called by setupModel(). This calls CSVReader that reads the CSV file and loads the data into vectors. These vectors are then passed to RecordsManager.createHeadersFor*(). This passes the user's inputs
- The returned QString and QList are passed to TreeModel.setupModelData()
- The QString is parsed through using setupModelData(), wich populates the QList of TreeItems
- Entries are stored relative to the parent TreeItem
- RecordManager calls the CSVReader to read and load the CSV file
- The tree is displayed to the user using QTreeView.setModel() in the MainWindow

Comment on your findings, with specific references to the design/code elements/file names/etc.:
The CSVReader loads the data to an appropriate vector using the loadCSV() method. User inputs taken from the GUI are passed to the TreeModel, which demonstrates low coupling.

Certain function calls are subclass specific, like createQStringForPres() are in the respective TreeModel subclass, while the universal call, setupModelData(), is called in the parent class. This demonstrates proper inheritance.

Title: Catching sort fields list of length zero
Anticipated frequency of use: Low
Starting Point: New sort order is determined by the user, which is taken as an input. However, while the user enters a name, the user does not pick the headers before saving.
Expected Outputs: Error dialog with no new sort order being created. The CustomSort dialog remains.

Inputs:

- Title of Sort Order: not passed to QSortListIO
- Sort Order Headers: not passed to QSortListIO
- Save button clicked: error dialog is displayed and the main dialog remains

Walkthrough:

- User clicks the Save button and the on_buttonBox_accepted() method is called
- The appropriate case is used, ex. Presentation
- Method determines if the dropdown is empty
- Code breaks to create a QMessageBox using an if else statement
- User is notified using the QMessageBox, and will remain in the CustomSort dialog

Comment on your findings, with specific references to the design/code elements/file names/etc.:
The system has features in place that allow it to catch errors without crashing. setNext() makes it so that the user cannot select the first header in a separate dropdown other than the first. In addition, on_buttonBox_accepted() is structured so that no saving is conducted before the input is verified. This removes the possibility of a logical error where the user doubles the headers.

END.

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.

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

Classes inspected: TreeModel.cpp, CustomSort.cpp, TestCSVReader.cpp
Examining the implementation of the bar graph

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 classes CustomSort, TreeModel, and MainWindow classes can be found in the attached class diagram found in this document.*
Comment on your findings: *All methods and attributes for CustomSort, TreeModel, and TestCSVReader can be found in the class diagram.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Compare functionality and performance with respect to their intended purpose according to the dashboard summary requirements.*
Comment on your findings: *Classes function correctly and are accurately represented in the class diagram.*

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: *Determine the functionality of each method and compare it with other methods that serve the class. Specifically, the use of the return values and presence of member variables will be observed.*
Comment on your findings: *TreeModel and TestCSVReader are cohesive. While CustomSort has cohesion, the presence of the switch statements could be improved.*

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: *Determine the number of includes, global variables, and function calls*
Comment on your findings: *TreeModel depends on TreeItems with minimal shared dat athroug hthe parameters. There is no access to global variables. CustomSort has low coupling and only communicates with QT and STD libraries. TestCSVReader has low coupling and communicates with CSVReader through the public methods.*

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: *Determine how the different cases are separated.*
Comment on your findings: *Cases are separated in TestCSVReader through test cases. The method setupModel() implements in four separate classes for each dashboard type.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Determine if there are private, protected, and public methods and if they make sense.*
Comment on your findings: *No methods contain public or private designations in front them in TestCSVReader, TreeModel, or CustomSort..*

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: *Determining what the class could be used for with slight modifications*
Comment on your findings: *TestCSVReader is a test class, and as a result, is fairly specific. TreeModel could be modified and reused by implementing a new model. CustomSort could be reused, but would require slight changes to the GUI and functionality. Specifically, there would need to be changes to the methods and cases.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Reading through the code and understanding the functionality*
Comment on your findings: *Function names are descriptive enough for users to understand.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Verify the presence of comments in the classes*
Comment on your findings: *Classes are documented and easy to understand. While CustomSort lacks comments, it is easy enough to understand.*

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: *Determining what would be need to add a new dashboard type*
Comment on your findings: *Changing the GUI, writing a few more methods, and adding different cases would be easy to perform given the structure of the application.*

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: *Looking through the code for unnecessary nested loops or inefficient data structures.*
Comment on your findings: *CustomSort does not use nested loops and has limited calls to other functions. While TreeModel has a nested loop, it is required. It is considered efficient given the nature of the data structure. The setupModelData() function has a nested loop, but does not impact efficiency to a large extent.*

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: *Determining the depth of inheritance.*
Comment on your findings: *TreeModel is the main class with inheritance, which goes one layer deep.*

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: *Determine the number of children classes*
Comment on your findings: *TreeModel is inherited by four subclasses for each dashboard type. However, child classes may not be necessary.*

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:

xi) Title of scenario
xii) Anticipated frequency of use (high, normal, low)
xiii) End-user trigger (starting point) for the scenario.
xiv) Expected type of outputs.
xv) 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.
_____-
_____

(Note: expand here as necessary for each scenario)

Title: Load CSV file and create stacked bar graph
Anticipated frequency of us: Medium
Starting Point: User clicks "Load File" and selects the appropriate CSV. User then selects the desired data and chooses the bar graph option.
Expected Outputs: A properly formatted bar graph

Inputs:

- Desired file: passed to CSVReader
- Desired data set: passed to RecordsManager
- Desired display option: used in MainWindow

Walkthrough:

- User selects a list of file names from a file dialog from the MainWindow class using on_actionLoad_file_triggered()
- MainWindow determines which file is loaded using checkFile()
- CSVReader is used to load the data in the files, and MainWindow checks if the data is error-free using break statements for specific scenarios
- RecordsManager is used to build the object using the select fields
- In on_sort_apply_clicked(), the indexes of all the highlighted rows are passed to the MySortFilterProxyMode class

END.

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.

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

Classes inspected: TreeModel, CSVReader, MainWindow

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 classes CSVReader, TreeModel, and MainWindow classes can be found in the attached class diagram found in this document.*

Comment on your findings: *All methods and attributes for CSVReader, TreeModel, and MainWindow can be found in the class diagram.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Compare functionality and performance with respect to their intended purpose according to the dashboard summary requirements.*

Comment on your findings: *Implementations of the TreeModel class could be simplified. Rather than copying the different functions for each of the four implementations, the code could be simplified to allow for just one occurrence of each function that could be used by each of the entities. This would simplify the current complexity.*

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: *Determine the functionality of each method and compare it with other methods that serve the class. Specifically, the use of the return values and presence of member variables will be observed.*

Comment on your findings: *TreeModel implements the TreeItem Class. The setupBarChart(QCustomPlot *barChart, std::vector<std::pair<std::string, double>>barChartList) method in MainWindow is encapsulated in MainWindow.*

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: *Determine the number of includes, global variables, and function calls*
Comment on your findings: *TreeModel depends on the TreeItem class. publicationTree, teachingTree, presentationTree, fundingTree implement the TreeModel class. TestCSVReader depends on the CSVReader 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: *Determine how the different cases are separated.*
Comment on your findings: *Distinct separation of the groups in the tree, items in the tree, data, and index. Represented in the MainWindow class: case FUNDING, case PRESENTATIONS, case PUBLICATIONS, case TEACH*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Determine if there are private, protected, and public methods and if they make sense.*
Comment on your findings: *No methods contain public or private designations in front them in CSVReader, TreeModel, or MainWindow.*

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: *Determining what the class could be used for with slight modifications*
Comment on your findings: *TreeModel can be constantly reused, especially since the switch cases differentiate the different dashboard types. CSVReader can be constantly reused.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Determine the size and ease of understanding for the classes.*
Comment on your findings: *There are difficulties with the ease of understanding of the MainWindow class from the size of it. It has ~2300 line of code given the amount of repetition need*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Verify the presence and clarity of comments in the classes*
Comment on your findings: *CustomSort is light on comments, but it is easy enough to understand. MainWindow lacks appropriate commenting. As an example, on_fundTreeView_click() is fairly difficult to understand and lacks the appropriate level of commenting.*

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: *Determining what would be need to add a new dashboard type*
Comment on your findings: *Changing the GUI, writing a few more methods, and adding different cases would be easy to perform given the structure of the application.*

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: *Looking through the code for unnecessary nested loops or inefficient data structures.*
Comment on your findings: *CustomSort does not use nested loops and has limited calls to other functions. TreeModel is also efficient given the nature of the data structure. The setupModelData() function has a nested loop, but does not impact efficiency to a large extent.*

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: *Determining the depth of inheritance.*
Comment on your findings: *CustomSort inherits QDialog, which is a Qt class. TreeModel implementations inherits from TreeModel which inherits from QAbstractItemModel. Inheritance does not run very deep.*

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: *Determine the number of children classes*
Comment on your findings: *TreeModel is inherited by four subclasses for each dashboard type.*

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:

xvi) Title of scenario
xvii) Anticipated frequency of use (high, normal, low)
xviii) End-user trigger (starting point) for the scenario.
xix) Expected type of outputs.
xx) 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.
_____-
_____

(Note: expand here as necessary for each scenario)

Title: Read a CSV file to create an appropriate tree Anticipated frequency of us: High
Starting Point: User clicks "Load File" and selects the appropriate CSV
Expected Outputs: A properly formatted TreeModel

Inputs:

- Start and end year of a date range: passed to RecordsManager
- A list of the fields (headers) to sort by: passed to RecordsManager
- A char range to filter the first header by: passed to RecordsManager

Walkthrough:

- Similar walkthrough to the CSVReader process with the addition of:
- When closeEvent() is run on MainWindow, saveProgramState() is called
- This creates a new QFile called "state.dat" which tracks the information need to start at the same place upon re-opening
- The constructor MainWindow() then opens the QFile when restarted to load the needed information

END.

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.

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

Classes inspected: MainWindow.cpp, SortFilterProxyModel.cpp, MySortFilterProxyModel.cpp

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 classes SortFilterProxyModel and MainWindow classes can be found in the attached class diagram found in this document.*

Comment on your findings: *All methods and attributes for SortFilterProxyModel and MainWindow can be found in the class diagram.*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Compare functionality and performance with respect to their intended purpose according to the dashboard summary requirements.*
Comment on your findings: *Functionality in the two classes work 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: *Determine the functionality of each method and compare it with other methods that serve the class. Specifically, the use of the return values and presence of member variables will be observed.*

Comment on your findings: *The setupBarChart(QCustomPlot *barChart, std::vector<std::pair<std::string, double>>barChartList) method in MainWindow is encapsulated in MainWindow. SortFilterProxyModel methods are properly encapsulated. The filterAcceptsRow() method has high cohesion.*

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: *Determine the number of includes, global variables, and function calls*
Comment on your findings: *While MainWindow depends on different classes through function calls, the interdependency is needed to ensure proper encapsulation.*

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: *Determine how the different cases are separated.*

Comment on your findings: *Distinct separation of the groups in the tree, items in the tree, data, and index. Represented in the MainWindow class: case FUNDING, case PRESENTATIONS, case PUBLICATIONS, case TEACH*

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

■ Yes ☐No ☐Partly (Can be improved)

Comment on your analysis: *Determine if there are private, protected, and public methods and if they make sense.*
Comment on your findings: *No methods contain public or private designations in front them in SortFilterProxyModel or MainWindow.*

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: *Determining what the class could be used for with slight modifications*
Comment on your findings: *Classes can be constantly reused especially since they can adapt to different dashboard types and scenarios.*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Determine the size and ease of understanding for the classes.*
Comment on your findings: *There are difficulties with the ease of understanding of the MainWindow class from the size of it. It has ~2300 line of code given the amount of repetition need*

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

☐ Yes ☐No ■Partly (Can be improved)

Comment on your analysis: *Verify the presence and clarity of comments in the classes*
Comment on your findings: *MainWindow lacks appropriate commenting. As an example, on_fundTreeView_click() is fairly difficult to understand and lacks the appropriate level of commenting.*

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: *Determining what would be need to add a new dashboard type*
Comment on your findings: *Changing the GUI, writing a few more methods, and adding different cases would be easy to perform given the structure of the application.*

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: *Looking through the code for unnecessary nested loops or inefficient data structures.*
Comment on your findings: Classes do not have excessive nested loops or function calls.

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: *Determining the depth of inheritance.*
Comment on your findings: *Inheritance does not run very deep.*

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: *Determine the number of children classes*
Comment on your findings: *Classes do not have 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:

xxi) Title of scenario
xxii) Anticipated frequency of use (high, normal, low)
xxiii) End-user trigger (starting point) for the scenario.
xxiv) Expected type of outputs.
xxv) 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.
_____-
_____

(Note: expand here as necessary for each scenario)

Title: Read a CSV file to create an appropriate tree Anticipated frequency of us: High
Starting Point: User clicks "Load File" and selects the appropriate CSV
Expected Outputs: A properly formatted TreeModel

Inputs:

- Start and end year of a date range: passed to RecordsManager
- A list of the fields (headers) to sort by: passed to RecordsManager
- A char range to filter the first header by: passed to RecordsManager

Walkthrough:

- Similar walkthrough to the CSVReader process with the addition of:
- When closeEvent() is run on MainWindow, saveProgramState() is called
- This creates a new QFile called "state.dat" which tracks the information need to start at the same place upon re-opening
- The constructor MainWindow() then opens the QFile when restarted to load the needed information

END.