Peachy Galaxy – Collaboration Project with Schulich School of Medicine and Dentistry

Team Peach

Computer Science 3307A - Object-Oriented Design & Analysis
University of Western Ontario, London ON
Canada N6A 3K7

Table of contents

1 Minimum requirements

1.1 Dashboard screens

Peachy Galaxy allows the user to load a CSV file containing records for each of the four subject areas: (1) Teaching, (2) Publications, (3) Presentations, and (4) Grants and Clinical Funding. This is accomplished by the MainWindow, CSVReader, RecordsManager and TreeModel classes. MainWindow controls the graphical user interface behaviour at run-time. It also creates the file path for the CSV file to be uploaded, which is passed to CSVReader. CSVReader parses the headers and records from the CSV file and returns them to MainWindow, which then sends it to RecordsManager. RecordsManager is the database used to store, format, and obtain aggregate information of the records based on the selected dashboard summary view. The processed data is then sent as a single string to the TreeModel class, the data structure used to visualize the records. Each dashboard view has its own TreeModel class since the hierarchies are slightly different in each case. Finally TreeModel passes its information back to MainWindow, which creates a TreeView and displays the dashboard summary view.

1.2 Visualizations

Peachy Galaxy also allows visualization of the data in the dashboard summary. For each subject area, the user can display a pie chart or bar graph of the record information by clicking on the member name within the dashboard view. More filtered visualizations can be viewed by clicking on a category within a given member name in the dashboard view. The following visualizations are supported::

- Publications by type for a given member name and date range
- Funded research by type for a given member name and date range
- Presentation types by type for a given member name and date range
- Teaching by program level, type of course/activity and program level for a given member name by date range

2 Stretch goals

We chose to implement the following features and functionality in Peachy Galaxy as part of the project's stretch goals: (1) error processing, (2) custom dashboard sorting, (3) exporting to PDF and printing to file , and (4) Mac OSX compatibility.

2.1 Error processing

Peach Galaxy can handle records with missing mandatory fields, which are enumerated below for each subject area:

- Grants and Clinical Funding: Member Name, Funding Type, Status, Peer Reviewed?, and Role
- Presentations: Member Name, Type, Role, and Title
- Publications: Member Name, Type, Role, and Title
- Teaching: Member Name and Program

When the CSV file is parsed, our application sets aside any records for error processing. The user is then prompted to either discard or edit the invalid entries. While the former simply excludes these records from being loaded, the latter allows to modify the missing fields. An ErrorEditDialog consisting of the records to be edited is opened. The user then has the option to save the modified data so that it may by loaded along with the other records. However it should be noted that all missing fields of each record must be corrected in order for this procedure to be successful. Furthermore, the error processing does not check that the edited entries are of a valid format.

2.2 Custom dashboard sorting

Each subject area has a default hierarchy for its dashboard summary view. However our application offers the option to create a custom dashboard view structure. By clicking the Create New Sort Order button, the user can build their own sorting order from the list of available filters. When the user selects the sorting order, Peachy Galaxy reformats the data to restructure the dashboard view. There are also options to switch between and delete sorting orders for maximum flexibility. It should be mentioned that each subject area's default sorting order cannot be deleted.

2.3 PDF export and print to file

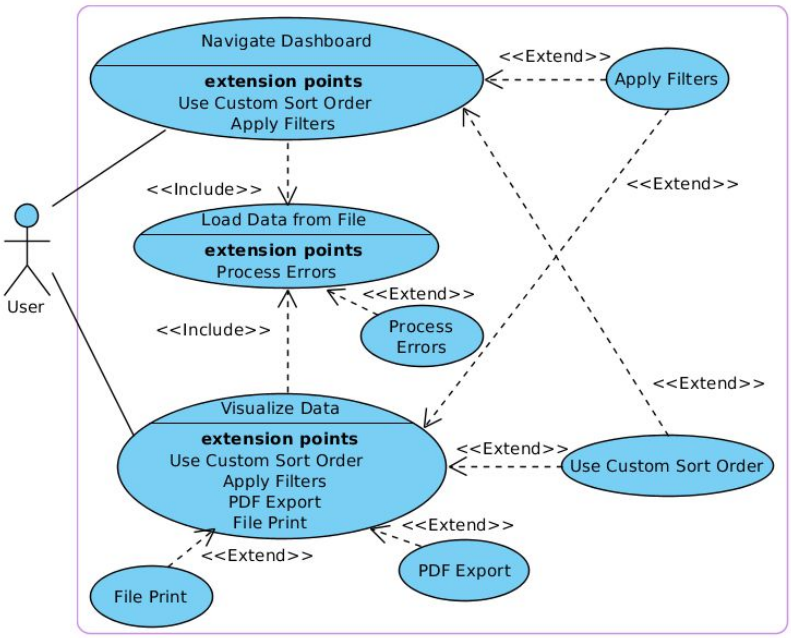
Peachy Galaxy allows any visualization, be it a pie chart or bar graph, to be exported as a PDF document or printed to a file. In both cases the application prompts the user for a file path and name. Although printing to file currently only supports PDF, there exists a framework for other formats such as PostScript. One can also consider developing functionality to print to a device (i.e. a physical printer).

2.4 Mac OSX Compatibility

In addition to running in the required Microsoft Windows 7 32-bit environment, our application is compatible with the Mac OSX operating system. It should be noted that each operating system requires its own specific installer and that the Mac OSX installer cannot be used in any Linux environment despite kernel similarities.

3 System design

3.1 Use case diagram



Above is the Peachy Galaxy 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 correspond 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:

3.1.1 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.

3.1.2 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.

4.1.3 Navigating the dashboard

Navigate Dashboard (Sea Level)

Main Success Scenario:

1. The user loads the data from file via the use case 3.*1.3 Loading data from file*. [Extension Point: 3.*1.4 Applying filters*. Applicable to step 3.] [Extension Point: *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.

3.1.4 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.
4. The system sets its member name range according to the values in the first and last letter of member last name boxes.

3.1.5 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]

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.

3.1.6 Visualizing data

Visualize Data (Sea Level)

Main Success Scenario:

1. The user loads the data from file via the use case 3.*1.3 Loading data from file*. [Extension Point: 3.*1.4 Applying filters*. Applicable to step 3] [Extension Point: *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: *3.1.7 Exporting to PDF*. Applicable to step 5] [Extension Point: *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.

3.1.7 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.

3.1.8 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

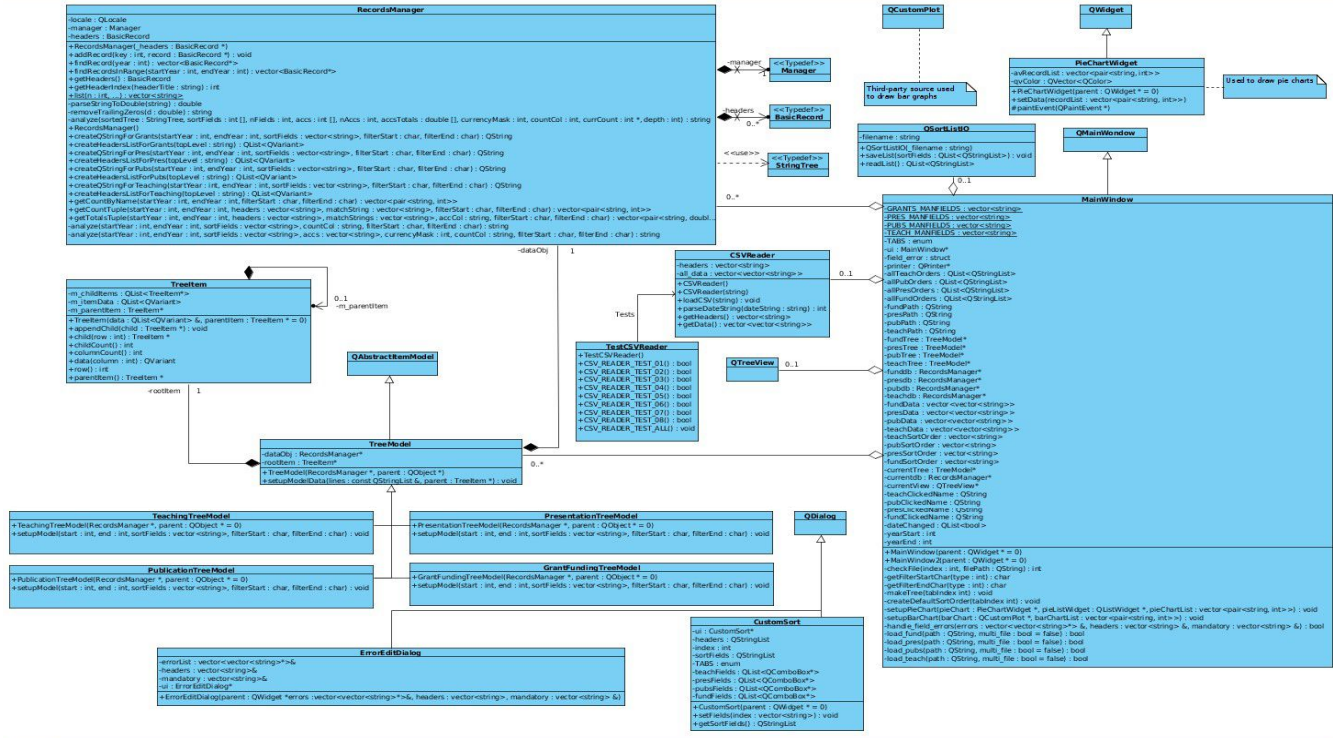
1. The system displays an error message.
2. The user accepts or closes the error message. [Return to Main Success Scenario step 3]

Alternate Course B: User clicks Cancel button

1. The system closes the file structure screen.

3.2 Class diagram

Below is the Peachy Galaxy class diagram describing the types of system objects and their static relationships. There are two main kinds of relationships between classes used in our diagram: associations, denoted by a line beginning at the source class and ending with a diamond at the target class, and generalizations, denoted by a line beginning at the source class and ending with a triangle at the target class.



Associations are read “<target class> has a <source class>”, while generalizations are read “<source class> is a <target class>”. The clear diamonds indicate shared associations while the black diamonds represent composite associations. In addition, each relationship a multiplicity indicating how many objects may fill the property. There is also a general association between TestCSVReader and CSVReader meaning the former is a test class of the latter. Below is a list of the classes and their relationships.

3.2.1 CSVReader

CSVReader is used in MainWindow to read and parse the data from the CSV file. This class is designed to meet the requirement of loading of data from a CSV file and perform error processing.

3.2.2 RecordsManager

RecordsManager is used in MainWindow to create records from a CSV file for the various summary types. These records are the bulk of the data manipulated by the system to create the required dashboard and visualizations. It is also used by TreeModel to sort the data and build the appropriate dashboard view.

3.2.3 TreeItem

TreeItem is used in TreeModel to store the records for the dashboard summary. Each TreeItem has a parent except for the rootItem, forming a linked list which facilitates and minimizes record storage time.

3.2.4 TreeModel

TreeModel is used in MainWindow to create and meet the dashboard summary requirement. TreeModel has a TreeItem which acts as a pointer to the first record in the list. It also has a RecordsManagers which it uses to build the data structure from the unsorted loaded data and passes on to the graphical user interface components. It is a generalization of the GrantFundingTreeModel, PresentationTreeModel, PublicationTreeModel, and TeachingTreeModel classes and is also a subclass of QabstractItemModel.

3.2.5 MainWindow

MainWindow contains the user interface and main program functionality for loading, filtering, and presenting data through the dashboard and other types of visualization. MainWindow is a subclass of QMainWindow, which allows it to access a vast amount of QT library features, standardizing and simplifying the user interface and visualizations implementation.

3.2.6 PieChartWidget

PieChartWidget is created during the execution of MainWindow; it is its own temporary stand-alone class for visualizing the data kept in the records inside the TreeModel. PieChartWidget is a subclass of QWidget so that it may take full advantage of the QT library features and functionality.

3.2.7 CustomSort

CustomSort is also created during the execution of MainWindow, representing a dialog for the user to customize and filter the data they wish to select for visualization. CustomSort is a sublass of QDialog so as to facilitate integration with the other Q-type graphical user interface components.

3.2.8 ErrorEditDialog

ErrorEditDialog is also a subclass of QDialog and is created during the execution of MainWindow. It allows the user to either discard or edit missing mandatory fields to the loaded data.

3.2.9 QCustomPlot

QcustomPlot is a third-party source class used to plot bar graphs of the loaded data for various dashboard views. It is a stand-alone library and is built to be integrated with QT. It is created when MainWindow is executed once the desired data is loaded from the file.

3.3.10 QSortListIO

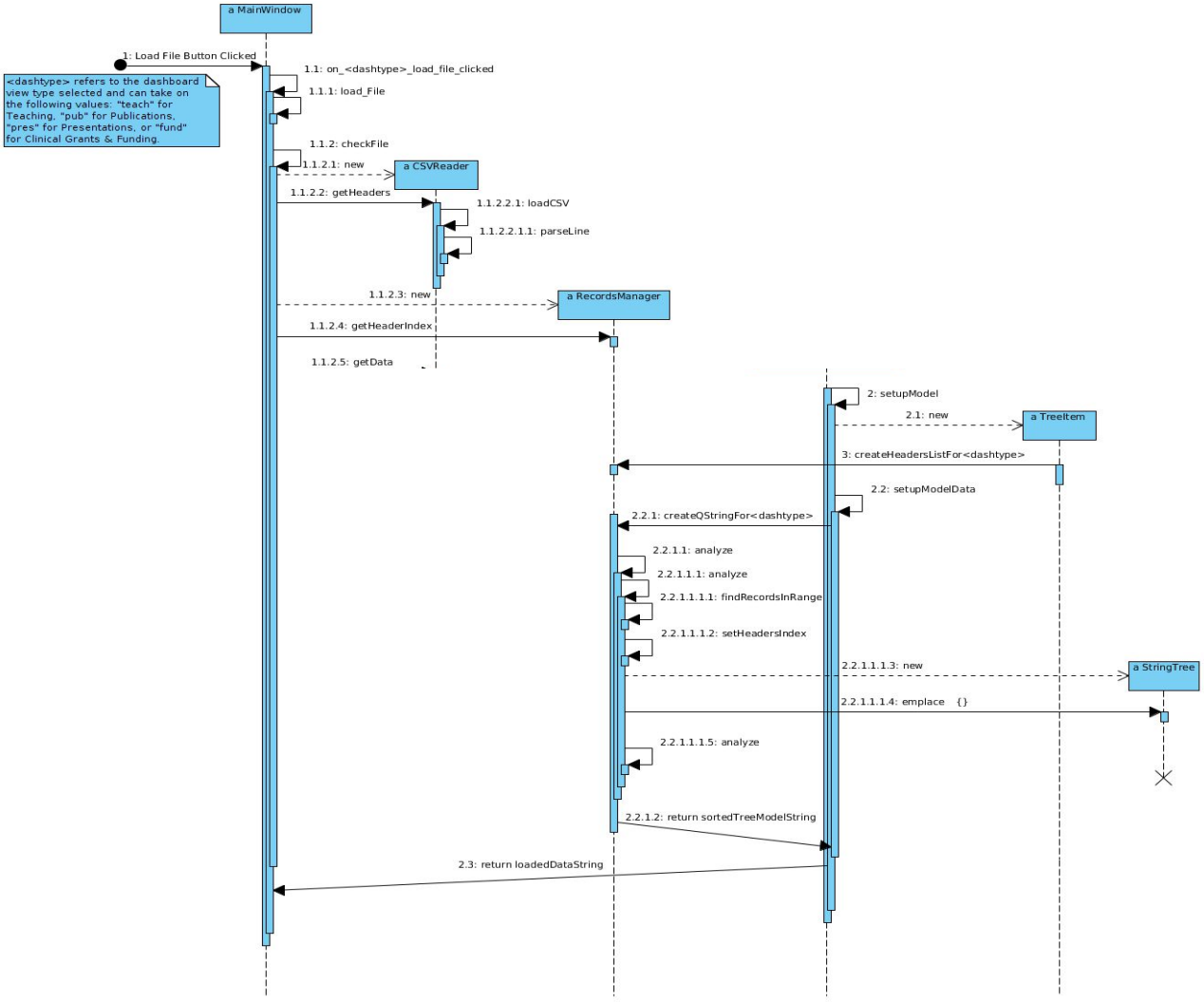
QsortListIO is used in MainWindow to read and write the custom sort order data. It serializes the information into a data stream which is then saved to a text file.

3.3 Sequence diagram – loading data from file

Above is the sequence diagram for the use case scenario “Load Data from File”, including the main scenario and extensions. Although the notation of the sequence diagram itself is quite self-explanatory, we present below the parts corresponding to each step in the scenario.

3.3.1 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.



3.3.2 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.

3.3.3 System verifies file is of proper type

This step is accounted for in the MainWindow “checkFile” method self-call, which first checks

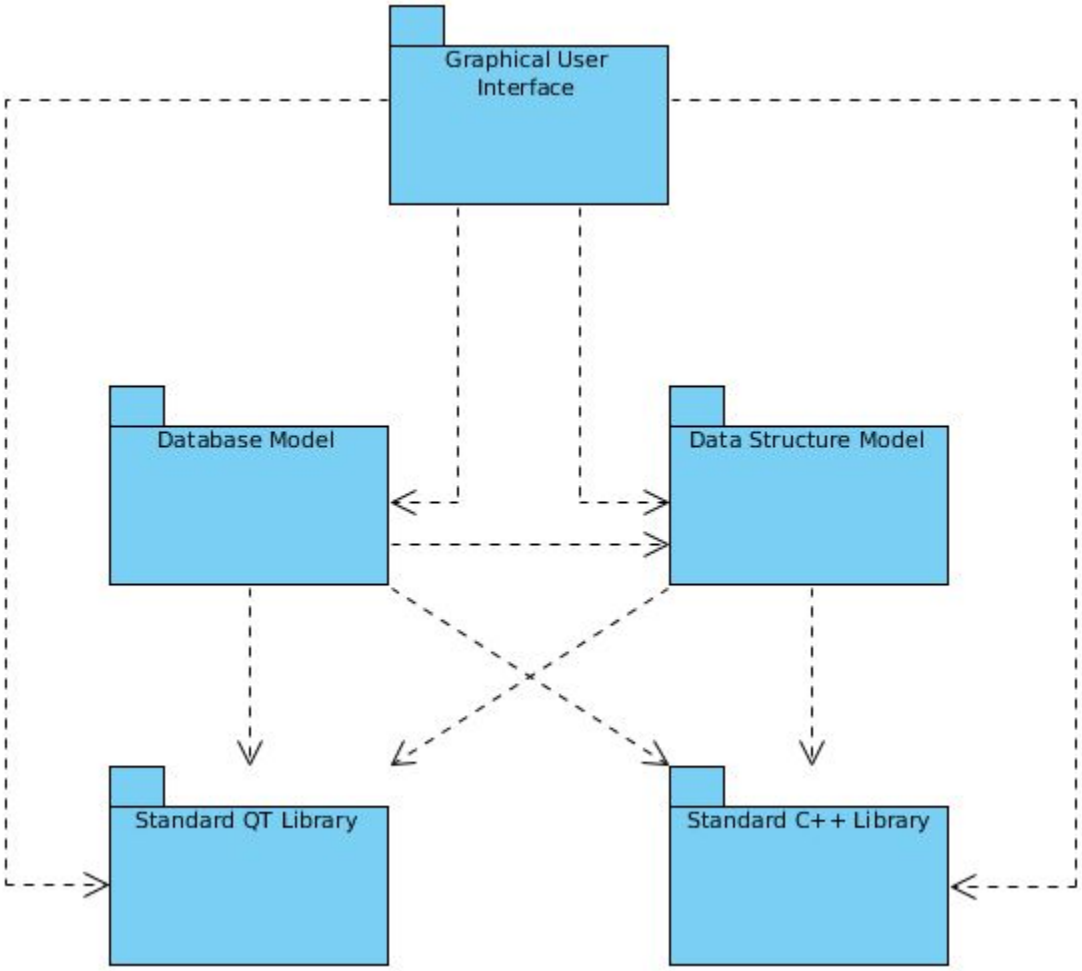
that the filepath is valid and the filetype matches the appropriate dashboard view. If either of these conditions fail, MainWindow displays the following message: “Not a valid <dashtype> file”. If however the conditions are met, MainWindow proceeds with loading the data. This approach was chosen in order 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.

3.3.4 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 use.

3.4 Package diagram

Above is the package diagram for Peachy Galaxy. There is only a single type of relationship in our diagram, the dependency, denoted by a dotted arrow beginning at the source package and ending at the target package and is read “<source package> depends on <target package>”. Below is a description of each package and explanation of its dependencies.



3.4.1

This package contains the classes relevant to the the user interface such as MainWindow, PieChartWidget and CustomSort. These classes control the behavior of all the graphical components (dialog boxes, buttons, widgets, scroll bars, etc.) during user interaction and thus naturally rely on the standard QT and C++ libraries. In addition, MainWindow needs to create new database and data structure models each time the user loads a new CSV file which means it depends on both of these packages.

3.4.2 Database model

This package contains the CSVReader and RecordsManager classes necessary for the creation and use of the database storing the loaded data. CSVReader is a simple file parser and only relies on the standard C++ library for strings and vectors. However, RecordsManager is more complex as it must create the appropriate database for each dashboard view type (Grants and Clinical Funding, Presentations, Publications and Teaching implemented for Stage 2) in a way which the graphical user

interface can easily use. To do this, it is dependent on both the standard QT and C++ libraries as well as the data structure model.

3.4.3 Data structure model

This package encompasses the abstract TreeModel class, which makes use of the TreeItem class, and its implementations for each of the implemented dashboard views: GrantFundingTreeModel, PresentationTreeModel, PublicationTreeModel and TeachingTreeModel. TreeModel does make use of RecordsManager when building itself and so the database and database model packages are technically interdependent. Although this goes against the Acyclic Dependency Principle, we believe breaking this rule of thumb is okay as the interdependency is localized and that, in particular, it does cross the application's layers.

3.4.4 Standard QT and C++ libraries

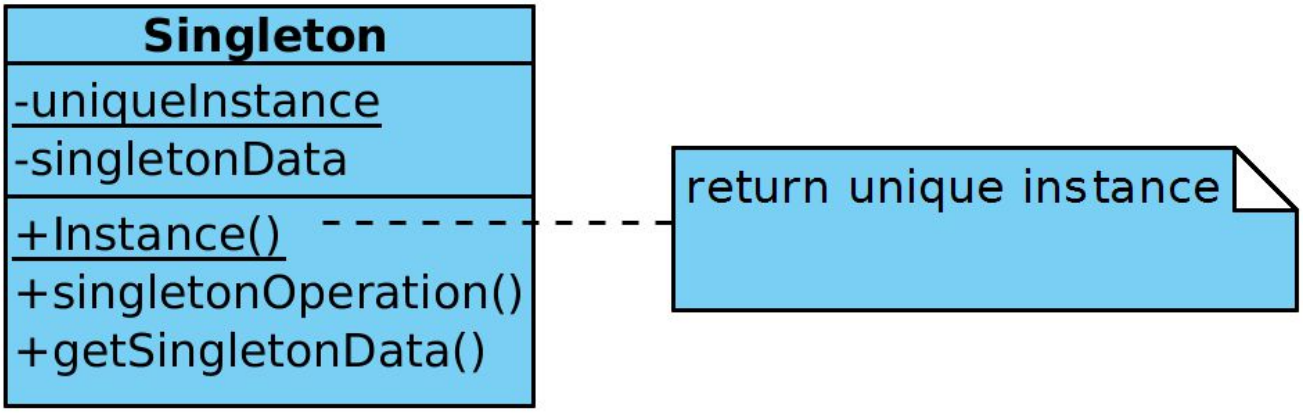
These are very well-documented stand-alone packages which are used by, but not in any way dependant on, the other packages in our application.

4 Design patterns

The Peachy Galaxy application makes use of two design patterns, Singleton and Prototype, both described below.

4.1 Singleton

The intention in using a Singleton pattern is to ensure a class only has one instance and provide a global point of access to it. One way to do this is by creating a global variable to make an object accessible, but this does not prevent one from instantiating multiple objects. A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the motivation behind the Singleton pattern illustrated below.



i. *Controlled access to sole instance*: because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
i. *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.
ii. *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.
iii. 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.
iv. *More flexible than 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.

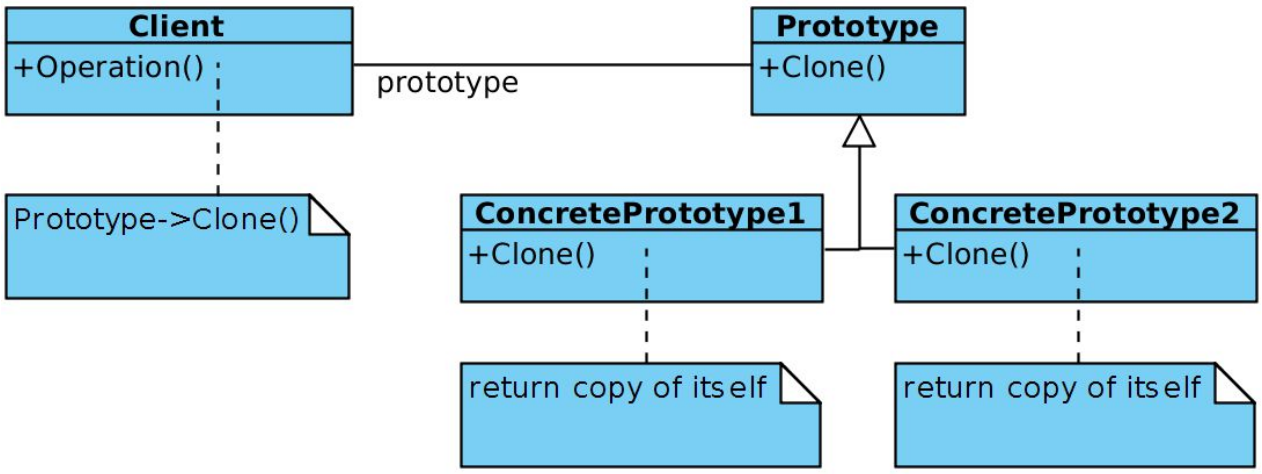
Although we do not require a variable number of instances, we acknowledge our flexibility in modifying our instance should the need arise. We use the Singleton design pattern in the implementation of MainWindow, which controls the runtime behaviour of the graphical user interface. This design decision 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.

There 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.

4.2 Prototype

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:

1) *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.
2) *Specifying new objects by varying values*: highly dynamic systems permit new behavior 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.
3) *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.
4) *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.
5) *Configuring 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.

In the case of TreeModel, 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.

5 Code and design inspection

Below is a table showing the classes inspected by each group member.

	A	B	C	D	E	F	G	H	I	J	K
1					X	X			X		
2	X			X						X	
3		X						X			X
4					X	X			X		
5		X						X			X
6			X				X				
7	X			X						X	
8			X				X				

The legend for the group members and class names is as follows:

A = CSVReader	B = CustomSort	C = ErrorEditDialog	D = MainWindow
E = PieChartWidget	F = QsortListIO	G = RecordsManager	H = TestCSVReader
I = TreeItem	J = TreeModel	K = <dashtype>TreeModel	1 = Member A
2 = Member B	3 = Member C	4 = Member D	5 = Member E
6 = Member F	7 = Member G	8 = Member H	

Complete code and design inspection documents can be found in Appendix A. It should be noted that we do not presently have reconciled versions because there were no major differences in the inspections between the pairs.

6 Development plans

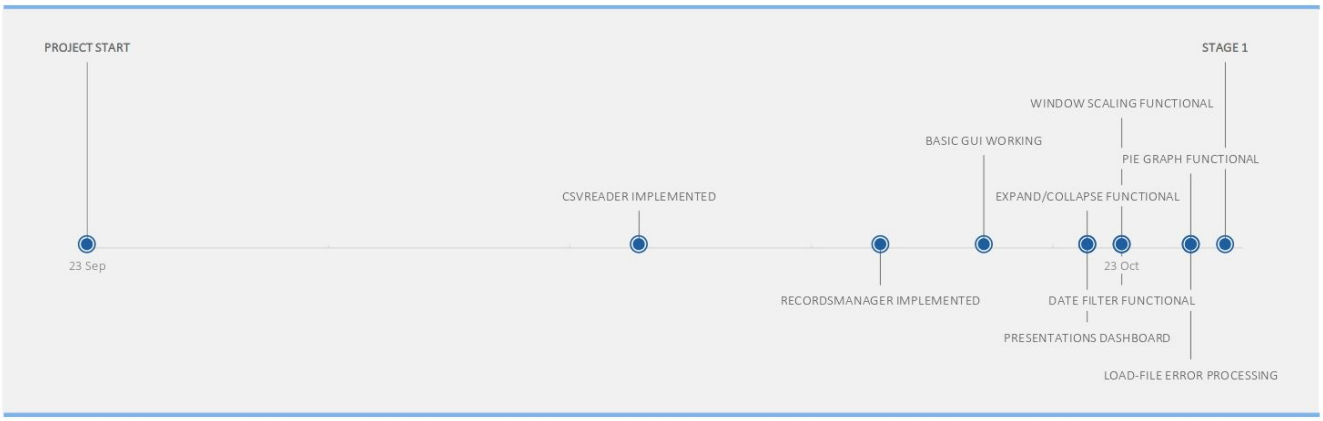
6.1 Agent-task tracker

Task: Stage 1	A	B	C	D	E	F	G	H	I
csvreader		X							
recordsmanager			X				X		
date filter						X			
basic GUI	X					X			
tree view of data			X						
presentations dashboard			X						
functional +/- buttons									
update GUI					X				
window resizing					X				
load-file error processing						X			
pie graph visualization				X					
use case diagram v1									X
timeline - achievements to date								X	
plan for stage 1 -> stage 2					X				
class diagram v1						X			

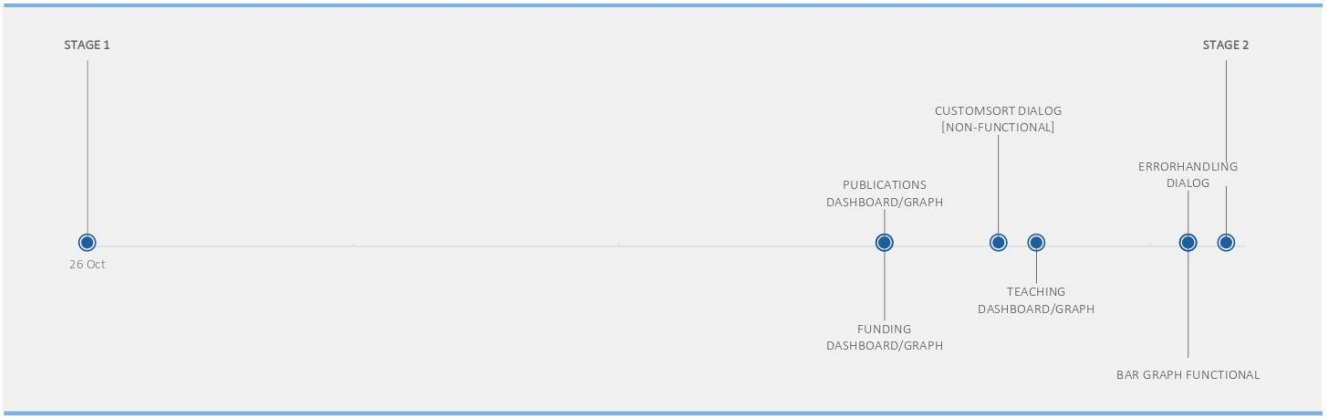
Task: Stage 2	A	B	C	D	E	F	G	H	I
teaching dashboard								X	
teaching visualizations								X	
publications dashboard	X								
publications visualizations	X								
funding dashboard									X
funding visualizations									X
error handling dialog						X			
bar graph visualization				X					
test cases							X		
use case diagram v2									X
class diagram v2						X			
sequence diagram for a scenario									X
package diagram									X
design patterns explanation									X
plan for stage 2 -> final submission					X				

Task: Final Submission	A	B	C	D	E	F	G	H	I
print/export functional						X			
fix up error handling						X			
sorting dialog functional			X		X				
custom filter dialog			X		X				
test cases	X						X		X
Mac OS compatibility							X	X	
clean up and refactoring	X	X							X
lessons learned	X	X	X	X	X	X	X	X	X
code inspections	X	X	X	X	X	X	X	X	X

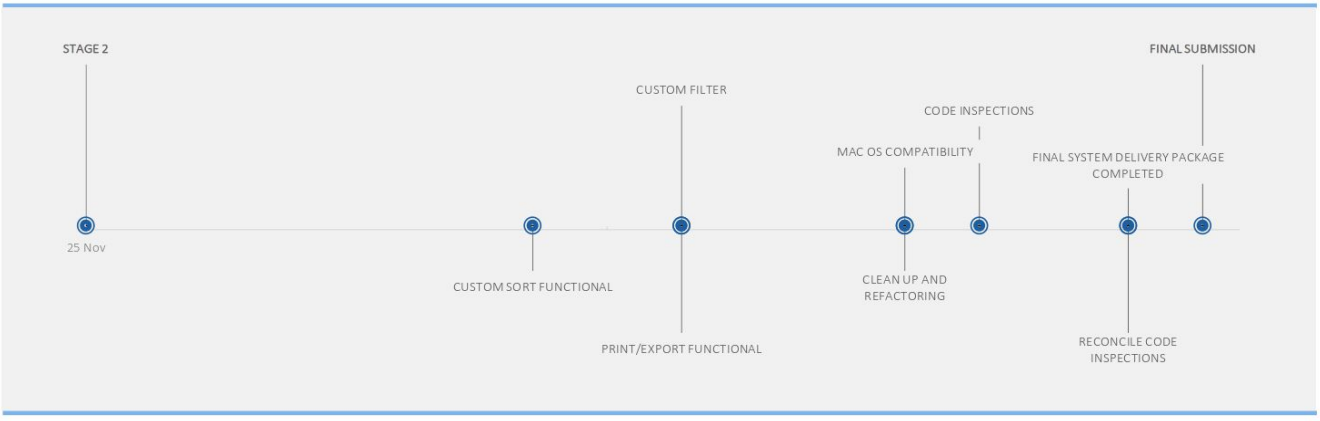
6.2 Stage 1 timeline



6.3 Stage 2 timeline



6.4 Final submission timeline



7 Lessons learnt

7.1 Problems with UML

Although the UML provides quite a considerable body of various diagrams that help to define an application, it is by no means a complete list of all the useful diagrams that one might want to use. One of the issues we encountered while making our UML diagrams was maintaining a consistent notation even though in general there exists no such standard. Let us take for example use cases, which are well known as an important part of the UML. The surprise is that in many ways, the definition of use cases in the UML is rather sparse. Nothing in the UML describes how one should capture the content of a use case. What the UML describes is a use case diagram, which shows how use cases relate to each other. But almost all the value of use cases lies in the content, and the diagram is of rather limited value.

7.2 Working with QT

Many of us already had experience with C++; consequently most of what we learned in terms of technical skills involved the QT library. For example an integral component of developing the graphical user interface involved understanding how QT handles actions when the user interacts with it through the use of signals and slots. In comparison, we found that signals and slots are much more intuitive than Java's action listeners in its Swing library. Specific components of the library that we worked with and gained a fairly good understanding of are listed in no particular order:

- QWidgets in the form of dates, buttons, and tree views
- QDialogs: loading multiple files in a dialog, creating custom dialog for error handling, and manipulating error dialogs
- -QT Creator Design Mode: organizing widgets using layouts, implementing methods for actions, connecting signals and slots

8 Future work

8.1 Application development

8.1.1 Increasing Functionality

In addition to meeting the minimal set of requirements prescribed by the user, we took on a number of orthogonal stretch goals. Some of the implementations of these goals are more functional than others. For instance, our custom sort orders feature is integrated with the dashboard summaries

and visualizations. It has been well-developed and will be easy to maintain and update. Our date range option allows filtering by year, which could be made more granular (i.e. by month or day) with minimal effort. However our error processing capability is still quite rudimentary. A simple improvement would be to give the user more flexibility in selecting which records they would like to edit or discard. A more involved process would be to perform a more rigorous inspection of the data to ensure proper formatting. The trajectory of this approach is likely to be limited by the way data is acquired in the ACUITY Star database. Although this idea goes beyond the scope of this project, we will touch on it briefly in section 8.2.

8.1.2 Enhancing usability

An important aspect of creating high-quality applications is maximizing the user experience. We offer a few suggestions as to how we can approach this problem:

1. *More pictures, less words*: this would entail removing letters from buttons and replacing them with meaningful icons which are easy to understand. This is what it means to adopt a truly graphical user interface and we have already begun doing so with our export and print buttons.
2. *Be strategic with your space*: this means having an intentive attitude toward the layout and presentation of the graphical user interface. As of now, the top third of Peachy Galaxy's main window is used for buttons and logos. This can be easily redesigned from a graphical perspective to occupy less space. The result would organize the flow of the application, making it more intuitive for the user, while freeing up the area for additional features. Classifications such as subject areas can be compacted further by layering them as tabs under a single button, while buttons which are often used (i.e. Load file) can be highlighted to make them more identifiable.

8.2 Project direction

8.2.1 Improving the database-application layer

This project's goal was to develop an application that manipulated information from CSV files. The records in these files are likely the result of a broad query to the ACUITY Star database. There are notions we would like to discuss on this topic. The first point is that the quality (i.e. format) of the data is substantially impacted by the organization and robustness of the database itself. Error processing to this effect will only be able to correct invalid entries to a certain degree. This is a key point to consider

should one wish to continue developing Peachy Galaxy (or any similar application for that matter) along these lines. However the second and arguably more fundamental point to be made is whether an application such as Peachy Galaxy should exist at all. One could combine PHP and HTML to create a web-based application which may have even more potential functionality because all information would be obtained by directly querying the database. Such an application would also shift the system requirements problem to a browser dependency problem, removing the issue of operating in legacy systems such as Windows XP. Furthermore, it would directly solve the human skills problem of employees being able to manipulate data based on their Excel proficiency, which was one of the strongest drivers to develop Peachy Galaxy in the first place.

8.2.2 Collaboration visualization

Peachy galaxy's internal system is designed to build a tree-like structure out of the data to build the various dashboard view types. To this end many classes are tree-oriented and have tree-optimized algorithms. Collaboration visualization on the other hand would ideally work in a graph-like environment. Therefore developing this type of feature in Peachy Galaxy may prove to be insurmountably difficulty without having to redesign the entire system. The is a link between trees and graphs: both can be implemented using linked lists. Although traversing a graph can produce a forest of rooted trees, we are not certain how to do this the other way around. We thus suggest to consider building the authorship collaboration network as a stand-alone application and, if succesful, attempt to incorporate dashboard summary views and visualizations.

Appendix A: Code and design inspection documents

Below are the complete code and design inspection documents completed by each group member.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: PieChartWidget, QSortListIO, TreeItem
Performed by: Member A

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: *Compared class diagrams with header files.*

Comment on your findings: *Classes are accurately represented.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Tested program with reference to requirements appendices.*

Comment on your findings: *Classes work as intended; QSortListIO is a simple IO to serialize custom sort orders (additional feature beyond minimum requirements).*

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

☑ Yes ☐ No ☐ Partly (Can be increased)

Comment on your analysis: *Examined classes.*

Comment on your findings: *For each class, all methods access common data.*

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

☐ Yes ☑ No ☐ Partly (Can be reduced)

Comment on your analysis: *Examined classes.*

Comment on your findings: *All classes have low coupling and do not share common variables with other classes; data is shared via mutator and accessor 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: *Examined classes.*

Comment on your findings: *All the classes are generalized.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Examined headers.*

Comment on your findings: *PieChartWidget::paintEvent could be private.*

Reusability:
Are the programmed classes reusable in other applications or situations?

☑ Yes, most of the classes ☐ No, none of the classes ☐ Partly, some of the classes
☐ Don't know

Comment on your analysis: *Examined classes.*

Comment on your findings: *TreeItem contains information about its place within a tree data structure. QSortListIO performs 2 simple tasks: read and write a QList<QStringList> to a parameter filename. All of these classes perform a defined task independently and hence can be easily reused.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Examined headers.*

Comment on your findings: *Methods are straightforward and easily identifiable.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Examined classes.*

Comment on your findings: *PieChartWidget has no comments.*

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: *Examined classes.*

Comment on your findings: *Classes are generalized and could be easily enhanced with minor modifications to the code.*

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

☐ Yes ☑ No ☐ Partly (Can be improved) ☐ Don't know

Comment on your analysis: *Examined classes.*

Comment on your findings: *No nested loops.*

Depth of inheritance:
Do the inheritance relationships between the ancestor/decendent 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: *Examined headers.*

Comment on your findings: *PieChartWidget inherits QWidget, QSortListIO and TreeItem don't inherit anything.*

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: *Examined headers.*

Comment on your findings: *Classes are not extended.*

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

Each scenario is described as follows:

1) Title of scenario
2) Anticipated frequency of use (high, normal, low)
3) End-user trigger (starting point) for the scenario.

4) Expected type of outputs.
5) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be “touched” by the scenario and producing the anticipated outputs.

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

Scenario 1:

1) Show pie chart visualization
2) High
3) Click field in tree model view
4) Refresh visualizations
5) User clicks on tree view -> RecordsManager -> QCustomPlot -> PieChartWidget

Comment on your findings, with specific references to the design/code elements/file names/etc.:
Checks are performed when a user clicks on the tree view (do not refresh if same field is clicked twice, or if field clicked is a number value), the field string is then analyzed by RecordsManager to get the correct data to be displayed, which is then passed to QCustomPlot and PieChartWidget to update the visualizations.

Scenario 2:

1) Create new custom sort
2) Low
3) Click on ‘Create New Sort Order’ button
4) Open CustomSort dialog and serialize sort order if accepted
5) User clicks ‘Create New Sort Order’ button -> CustomSort -> QSortListIO

Comment on your findings, with specific references to the design/code elements/file names/etc.:
Creates a new CustomSort dialog to accept user input, on successful accepted(), the sort order will be added to the sort orders list and serialized via QSortListIO

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: CSVReader, MainWindow, TreeModel
Performed by: Member A

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: *CSVReader class, TreeModel class, mainwindow class are in class diagram.*

Comment on your findings: *In the class diagram, including all the methods in CSVReader, TreeModel and mainwindow.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *CSVReader is created to load csv file, it can return header vector, all data*

and year. TreeModel is created to create tree, and it works. mainwindow is created to group together all classes, some part does not works well because of the complextity.

Comment on your findings: L*oad file button is ugly but works. Treemodel is used in pubTree, teachTree, presTree, fundTree and works well.*

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

☑ Yes ☐ No ☐ Partly (Can be increased)

Comment on your analysis: *All the load files function is in the CSVReader, method are encapsulated in the clas. All the user interface button at here, method are encapsulated in the class. TreeModel can interact with other classes to make tree.*

Comment on your findings: *TreeModel includes TreeItem class.*
void MainWindow::setupBarChart(QCustomPlot *barChart,
std::vector<std::pair <std::string, double>> barChartList)

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

☐ Yes ☐ No ☑ Partly (Can be reduced)

Comment on your analysis: *CSVReader is used to load files. When files are loaded, CSVReader does not participate other functions after load files. columnCount,count number use TreeItem. mainwindow is used to group other function so must have some function couple other classes variables.*

Comment on your findings: *There is no class include CSVReader class, except TestCSVReader.cpp. pubTree, teachTree, presTree, fundTree in each part of make tree. TreeModel includes TreeItem 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: *Load file problem is decomposed into 4 parts. a tree has a lot of things like node ,leaf so different thinking when solve the different part of a tree. The works in mainwindows*

mainly be separated by 4 parts-GrantFunding,Presentation,Publication,Teaching.

Comment on your findings: *To distinct Header of tree and item in the tree,distinct data and index . in class, always see:*

```
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: *Because CSVReader is seldom to connect to other class, so no use to distinct public or private.in other classes which is used in mainwindow has contain proper access specifications so mainwindow does not need. TreeModel does not need it.*

Comment on your findings: *There is no public or private before each method and definition of function in CSVReader. There is no public or private before each method and definition of function in TreeModel class. There is no public or private before each method and definition of function in mainwindow.cpp.*

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: *A lot of applications need load file function . A lot of applications create Tree. A lot of applications depend on other classes , if without other classes, it does not work nut the format can be reused.*

Comment on your findings: *Email, Word, Power Point load file function. Tree is a data structure always be used in developing. the use of switch cases to integrate other classes in.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Commands are clear. commands are clear, but class is too big and hard to read. need to command in the process to create a tree.*

Comment on your findings: *The whole mainwindow class is more than 1500 lines. the whole TreeModel class is around 200 lines.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Load CSV file function is most complicate part but it a little bit hard to understand without read code. there are a lot of lines in each method that means it have more chance hard to understand.*

Comment on your findings: //Loads the CSV file at filename
// Returns the year in a date string.
// Returns the header vector for a CSV
// Returns all data from a CSV

just a few command
// While the file's not empty
// For each character in the line
// For each element
// If we have a quote, continue till the next quote
// Last quote

//Returns the number of columns (headers) this node has.
//Returns a flag ensuring the model is read-only.
// Returns the header information that's stored in the root.
// Returns the number of rows this node has.
// Sets up the initial data in the model.

void MainWindow::on_fundTreeView_clicked(const QModelIndex &index) *function is one of most complicate part but it does not have command.*

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: *Mainwindow class interacts with a lot of classes, it hard to change without change other classes.*

Comment on your findings: MakeTree is hard to change without change TreeModel class and see case FUNDING, PRESENTATIONS, PUBLICATIONS, TEACH

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: CSVReader is not a big class ,code are simple. TreeModel seldom to use complexity algorithm but some part does . mainwindow use a lot of if, switch,for.

Comment on your findings: *Each method just 2-3 lines;* void TreeModel::setupModelData(const QStringList &lines, TreeItem *parent).

Depth of inheritance:
Do the inheritance relationships between the ancestor/decendent 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: *No interface in CSVReader. no interface in mainwindow. TreeModel is not superclass ,the project do not have differentiate load files species.*

Comment on your findings: *No class interface CSVReader and CSVReader does not interface other class. No class interface mainwindow and mainwindow does not interface other class. No class implement TreeModel and TreeModel does not implement other class.*

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

☐ Yes ☑ No ☐ Partly (Can be improved)

Comment on your analysis: *CSVReader is not superclass ,the project do not have differentiate load files species. TreeModel is not superclass ,the project do not have differentiate load files species. mainwindow is not superclass ,the project do not have differentiate load files species.*

Comment on your findings: *No class implement CSVReader and CSVReader does not implement other class. No class implement TreeModel and TreeModel does not implement other class. No class implement mainwindow and mainwindow does not implement other class.*

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 .

1) Each scenario is described as follows:
2) Title of scenario
3) Anticipated frequency of use (high, normal, low)
4) End-user trigger (starting point) for the scenario.
5) Expected type of outputs.
6) 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.

Title: Load TeachingCSV File
Anticipated frequency of use: high frequency
Starting Point: Touching load file button and choose a CSV files from project information
Expected Outputs: A Teaching Data Tree and graph
Inputs:

- Year data to CSVReader:: parseDateString ()
- The data of the CSV file to CSVReader::getData()
- The header is going to CSVReader::getHeaders()

Walkthrough

1. mainwindow creates a TeachingTreeModel, which is filtered by switch case.
2. TreeModel.setupModelData() create TeachingTree.
3. TeachingTreeModel::setupModel() get data and pass data to TreeItem
4. TreeItem get the data and create the TeachingTree

Comment on your findings, with specific references to the design/code elements/file names/etc.:
CSVReader load file, RecordsManager analysis data. Mainwindow is the interface between user and Code.TeachingTreeModel,TreeModel,TreeItem is used to create the TeachingTree and through mainwindow to show to the user.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: TreeModel.cpp (incl. children), CustomSort.cpp, TestCSVReader.cpp
Performed by: Member B
All tests were run on the grants dashboard summary.

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: *Compared the class diagram with the header source files.*

Comment on your findings: *The diagrams are congruent with the programmed application.*

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

☐ Yes ☐ No ☑Partly (Can be improved)

Comment on your analysis: *Ran the program to test the functionality of the various components.*
Comment on your findings:
-TreeModel and its derivatives produces a valid TreeModel object accepted by the TreeModelView

widgets.
-Customsort produces a valid dialog to create a custom ordering in the tree.
-TestCSVReader only tests if the reader has read data in from a given file. Could be improved to test for correctness of read data. For example, the number of headers is equal to the number of columns. Or given a sample test file, does it read the header strings correctly even with special characters intermingled.

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

☑ Yes ☐ No ☐ Partly (Can be increased)

Comment on your analysis: *Inspect the methods located in the source files.*

Comment on your findings:
-TreeModel is a data structure containing tree items that contain strings of data. Most methods in TreeModel only access the data provided in its dataObj and the tree's root item.
-CustomSort has high cohesion since most of the methods access the same member object.
-All methods in TestCSVReader access similar data; either header data or table data.

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

☐ Yes ☑No ☐ Partly (Can be reduced)

Comment on your analysis: *Inspect class diagram to view associations with source as reference.*

Comment on your findings:
-TreeModel has low coupling; all interactions are with the TreeItems class with minimal shared data through parameters. Child classes access the dataObj through its methods to retrieve data. No access to global variables.
-CustomSort has low coupling; it only interacts with the QT and STD libraries. Uses message passing to share it's data.
-TestCSVReader has low coupling; interacts with CSVReader through its 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: *Define purposes for classes and related concerns. Assess whether concerns are encapsulated.*

Comment on your findings:
-TreeModel: uses inheritance to encapsulate the various types of tree model's that exist. Allows for run time polymorphism (though not currently implemented). The constructor should be generalized to just accept the data that is required rather than having it access it itself.
-CustomSort: It could possibly also be subclassed into different types for each dashboard view. Code is repeated, it may be possible to generalize.

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: I*nspection of header files and assessing purpose of functions.*

Comment on your findings:
TreeModel: This is the required interface for the QWidget TreeView. Functions that deal with the internal working of the file are protected.
CustomSort: Only two methods are exposed to the public the rest are private. The exposed methods are required for the operation of the class.
TestCSVReader: All methods are public to run the specific tests.

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: *Assess the ability of the classes to perform indepently.*

Comment on your findings:
TreeModel: Since TreeModel is fed the data in a specific format, it can be generalized. The constructor should remove the pointer parameter to the RecordsManager object and instead just have it be passed the data it requires to build the tree.
CustomSort: For the most part is pretty specialized for this project and one could imagine a more generalized class that produces a QDialog like this.
TestCSVReader: It is very specific for this project since it tests our CSV Reader; However the CSVReader can be used to read CSVs in a similar format; therefore, TestCSVReader can be reused.

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

☐ Yes ☐No ☑Partly (Can be improved)

Comment on your analysis: *Scan the headers methods to see if it is easily understood what the function is performing. Reference source if not clear.*

Comment on your findings:
-TreeModel: could use some clarity in some method names (e.g. data method), but the source is pretty simple to understand what is happening.
-CustomSort: Some methods don't seem very clear in their functionality. For example the setFields method, I am unsure what fields I am setting.
-TestCSVReader, the tests are very straightforward.

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

☐ Yes ☐ No ☑Partly (Can be improved)

Comment on your analysis: *Examine source files for comments.*

Comment on your findings:
TreeModel contains adequate comments.
CustomSort requires comments.

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: *Think of possible enhancements and how they would be implemented (new test cases, new dashboard views)*

Comment on your findings: *For the most part enhancements would only require minor extensions to the code.*

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

☐ Yes ☑No ☐Partly (Can be improved) ☐ Don't know

Comment on your analysis: **Assess complexity of methods.**

Comment on your findings:
TreeModel: Has a nested loop, but it is required since we are dealing with tabular data (has to go through columns/rows)
CustomerSort: Has no nested loops.
TestCSVReader: negligible.

Depth of inheritance:
Do the inheritance relationships between the ancestor/decendent 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*: Review class diagram*

Comment on your findings: *TreeModel is the only class with inheritance, it 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: *Review purpose of child classes.*

Comment on your findings: *the rest of the code, it is debatable whether the child classes are really needed.*

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

Each scenario is described as follows:
1) Title of scenario
2) Anticipated frequency of use (high, normal, low)
3) End-user trigger (starting point) for the scenario.

4) Expected type of outputs.
5) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be “touched” by the scenario and producing the anticipated outputs.

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

1) Load multiple CSV file
2) Low
3) Click Menu Load File…
4) Correct dashboard tree view and graph.
5) List of file names selected by user -> MainWindow -> CSVReader -> ErrorEditDialog (if errors) -> RecordsManager (multiple times depending on how many file types loaded) -> TreeModel derivatives (again depending on how many / what files are loaded)

-User selects a list of file names from a file dialog launched in MainWindow class.

-MainWindow ascertains what types of files are loaded (grants, teaching, etc) and will load the first encountered in the list if multiple of the same types are loaded.

-CSVReader is used to read in the data in the file names provided from the user’s selection.

-MainWindow checks whether the data from CSVReader is error free; if not an error dialog is provided.

-RecordsManager is used to build an object representing the data.

-TreeModels are created using public member functions of the RecordsManager to access the data and “feed” it to the model.

Comment on your findings, with specific references to the design/code elements/file names/etc.:
It would appear that the scenario is implemented well. Components are separated into distinct classes with distinct functionality. Models are decoupled from the data and do not access it directly but is provided it by the records manager.

1) Edit date range.
2) High
3) Edit start and/or end date range (Assuming file is already loaded).
4) Updated dashboard and graph models.
5) User updates date -> RecordsManager -> TreeModel -> MainWindow

-User enters a new date (only one can be changed a time), say the end date is changed.
-MainWindow catches the change in date box.
-MainWindow requests data from the RecordsManager in the new date range.
-MainWindow builds a new tree model for the dashboard view and replaces the widget.

Comment on your findings, with specific references to the design/code elements/file names/etc.:
Similar to the last analysis but expanding on how TreeModels handle the data. It appears through the walkthrough that the child classes of TreeModel are not really required. The TreeModel should not have to be passed a pointer to a unique data type in its constructor, because then TreeModel has to access the data through unique methods in a relatively specialized class for this project. Instead TreeModel should be provided the data it needs in its constructor by the MainWindow class.

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)
- ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: TreeItem.cpp, QSortListO.cpp, piechartwidget.cpp
Performed by: Member C

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: *Compare the class diagrams with the implemented code.*

Comment on your findings: *Matches.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Look at whether the code successfully performs as intended and if it works as per the customer's requirements.*

Comment on your findings: *TreeItem is a container for items of data supplied by the simple tree model. QSortListO works as intended sorting the data, PieChartWidget were implemented as part of a GUI only runs if the user select pie chart as an option of display and nothing else.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Observe the functionality of each function, the presence of member variables, and the use of the return values of the functions.*

Comment on your findings: *TreeItem , pichartwidget and QSortListO have a specific function. They are cohesive in that each function performs only one test.*

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

☐ Yes ☑ No ☐ Partly (Can be improved)

Comment on your analysis: *Look at the number of includes, look for global variables, look for function calls to members of other classes.*

Comment on your findings: *No global variables. All classes have a minimal number of includes. Each class can be understood on its own, without looking at the other classes.*

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: *Observe how different cases are separated, if applicable.*

Comment on your findings: *TreeItem is a container for items of data supplied by the simple tree model. QSortListO works as intended sorting the data, PieChartWidget were implemented as part of a GUI only runs if the user select pie chart as an option of display and nothing else.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Look for public, private, protected methods and whether they make sense.*

Comment on your findings: *Functions have proper access specifications.*

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: *Consider what other scenarios these classes might be used for, with minor modifications.*

Comment on your findings: *TreItem is pretty specific as it's a test class and could easily reuse. QSortListO and piechartwidget could be reused, but it would take a few big changes (writing a few more methods, changing the GUI, adding another case to the switch statements.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Read through the function names. Do you get a gist of what it should do?*

Comment on your findings: *Function names in TreeItem, QSortlistO and piechartwidget are descriptive enough for anyone to understand what they should do.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Look for the presence of comments, particularly in complicated code.*

Comment on your findings: *TreeItem and QSortListO are pretty well commented. piechartwidget lacks comments but the code is pretty easy 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: *Consider adding a new dashboard type. What would be required to implement it?*

Comment on your findings: *Simply write new test cases for TreeItem. Otherwise, how to add a new dashboard type was discussed in the Reusability section.*

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: *Look at the code for unnecessary nested loops or inefficient (use of) data structures, etc.*

Comment on your findings: *QSortListO has no nested loops and uses limited calls to other functions. piechatwidget is slightly more complicated but is also quite efficient. TreeItem has a single nested loop, which negligibly affects efficiency (and is likely unavoidable).*

Depth of inheritance:
Do the inheritance relationships between the ancestor/decedent 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: *Numerically determine the depth of inheritance. Does this make sense?*

Comment on your findings: *piechartwidget doesn't inherit anything. QSortListO inherits QDialog, which is a Qt class for dealing with dialog windows. TreeItem is by TreeModel which inherits from QAbstractItemModel. The inheritance here doesn't go deep enough for it to not make sense.*

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: *Count the number of children classes. Does this make sense?*

Comment on your findings: *TreeItem, QSortListO and piechatwidget have no subclasses.*

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

Each scenario is described as follows:

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

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

TreeItem is not used by the end user.

Title: Load up a CSV File, created vectors
Anticipated frequency of use: High
Starting Point: User clicks load file and selects an appropriate CSV
Expected Outputs: Vectors with the data sorted by year
Inputs:

- Start and end year of a date range to filter entries by -> 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

1. mainwindow creates a new *TreeModel (ex. PublicationTreeModel)
2. mainwindow calls TreeModel.setupModel(), passing it the user's inputs
3. setupModel() calls RecordsManager.createQStringFor*() that calls the CSVReader that load the data into the vectors and pass it to the and RecordsManager.createHeadersFor*(), passing it the user's inputs
4. The return of these methods (a QString and a QList, respectively) is passed to TreeModel.setupModelData()
5. setupModelData() parses through the QString, populating a QList<TreeItem*>
6. All these entries are stored relative to a root TreeItem (parent)
7. The CSVReader is then used by RecordManager to load the CSV file

8. The TreeModel is then used by mainwindow in QTreeView.setModel() to display the tree to the user

Comment on your findings, with specific references to the design/code elements/file names/etc.:
To load the data to the vector is a function of the CSVReader, Most of the analysis is done by RecordsManager. The TreeModel function are passed the user inputs from the GUI and thus demonstrates low coupling. Specific function calls, such as createQStringForPubs(), are made in the specific TreeModel subclass, whereas the universal calls (setupModelData()) are done in the parent class, demonstrating proper use of inheritance.

Title: Catching a 0-length Sort Fields List
Anticipated frequency of use: Low
Starting Point: User clicks to create a new sort order, enters a name, but forgets to pick the headers before clicking Save
Expected Outputs: An error dialog to the user, the CustomSort dialog remains, no new sort order is created
Inputs:

- Title of Sort Order -> *not* passed to QSortListIO
- Sort Order Headers -> *not* passed to QSortListIO
- Save button clicked -> error dialog appears, main dialog remains

Walkthrough

1. User clicks Save and on_buttonBox_accepted() is called
2. The appropriate switch case is entered
3. Function identifies that the first dropdown (header) is empty
4. Before calling QSortListIO, function breaks to create a QmessageBox
5. User is notified and because we broke out of the processing function only, we are still in our CustomSort dialog

Comment on your findings, with specific references to the design/code elements/file names/etc.:
System is robustly designed so that it catches an absent-minded user's mistakes without crashing. Because of setNext(), the user cannot select the first header in a separate dropdown than the first (i.e. the first dropdown must be filled to create a sort order). Because on_buttonBox_accepted() is designed so that none of the saving is actually before user input is verified, we don't run into logic error where the user doubles one of the headers or something.

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: *TreeModel.cpp, CustomSort.cpp, TestCSVReader.cpp
Performed by: Member D

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: *Compare the class diagrams with the implemented code.*

Comment on your findings: *Matches.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Look at whether the code successfully performs as intended and if it works as per the customer's requirements.*

Comment on your findings: *TreeModel classes work. CustomSort works as intended, but were*

implemented as part of a stretch requirement. TestCSVReader only tests if CSVReader can open the file and nothing else

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Observe the functionality of each function, the presence of member variables, and the use of the return values of the functions.*

Comment on your findings: *TreeModel has many cohesive functions. TestCSVReader is cohesive in that each function performs only one test (except for CSV_READER_TEST_ALL()). CustomSort is quite good, but the presence of too many switch statement lead me to think it could be improved.*

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

☐ Yes ☑ No ☐ Partly (Can be improved)

Comment on your analysis: *Look at the number of includes, look for global variables, look for function calls to members of other classes.*

Comment on your findings: *No global variables. All classes have a minimal number of includes. Each class can be understood on its own, without looking at the other classes.*

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: *Observe how different cases are separated, if applicable.*

Comment on your findings: *TestCSVReader separates the different test cases. TreeModel abstract a function (setupModel()) which is implemented in 4 separate classes, one for each dashboard type. CustomSort uses switch statements instead. You could change this to 4 different subclasses instead.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Look for public, private, protected methods and whether they make sense.*

Comment on your findings: *Functions have proper access specifications.*

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: *Consider what other scenarios these classes might be used for, with minor modifications.*

Comment on your findings: *TestCSVReader is pretty specific as it's a test class. One could easily reuse TreeModel for another dashboard type by simply implementing a new setupModel(). CustomSort could be reused, but it would take a few big changes (writing a few more methods, changing the GUI, adding another case to the switch statements.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Read through the function names. Do you get a gist of what it should do?*

Comment on your findings: *Function names in TreeModel and CustomSort are descriptive enough for anyone to understand what they should do. Test case name could be more descriptive in TestCSVReader.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Look for the presence of comments, particularly in complicated code.*

Comment on your findings: *TreeModel is pretty well commented. TestCSVReader lacks comments but the code is pretty easy to understand. CustomSort lacks comments and could use some parts of the code or at least could use some function headers.*

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: *Consider adding a new dashboard type. What would be required to implement it?*
Comment on your findings: *Simply write new test cases for TestCSVReader. Otherwise, how to add a new dashboard type was discussed in the Reusability section.*

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: *Look at the code for unnecessary nested loops or inefficient (use of) data structures, etc.*

Comment on your findings: *CustomSort has no nested loops and uses limited calls to other functions. TreeModel is slightly more complicated but is also quite efficient. The main function, setupModelData(), has a single nested loop, which negligibly affects efficiency (and is likely unavoidable). It doesn't make sense to talk about efficiency of TestCSVReader.*

Depth of inheritance:
Do the inheritance relationships between the ancestor/decedent 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: *Numerically determine the depth of inheritance. Does this make sense?*

Comment on your findings: *TestCSVReader doesn't inherit anything. CustomSort inherits QDialog, which is a Qt class for dealing with dialog windows. The individual TreeModel classes inherit from TreeModel which inherits from QAbstractItemModel. The inheritance here doesn't go deep enough for it to not make sense.*

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: *Count the number of children classes. Does this make sense?*

Comment on your findings: *TestCSVReader and CustomSort have no subclasses. TreeModel is inherited by four subclass, one for each dashboard type, which makes sense.*

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:

6) Title of scenario
7) Anticipated frequency of use (high, normal, low)
8) End-user trigger (starting point) for the scenario.
9) Expected type of outputs.
10) 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.

TestCSVReader is not used by the end user.

Title: Load up a CSV File, Create the Tree
Anticipated frequency of use: High
Starting Point: User clicks load file and selects an appropriate CSV
Expected Outputs: A properly formatted and correct TreeModel
Inputs:

- Start and end year of a date range to filter entries by -> 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

1. mainwindow creates a new *TreeModel (ex. PublicationTreeModel)
2. mainwindow calls TreeModel.setupModel(), passing it the user's inputs
3. setupModel() calls RecordsManager.createQStringFor*() and RecordsManager.createHeadersFor*(), passing it the user's inputs
4. The return of these methods (a QString and a QList, respectively) is passed to

TreeModel.setupModelData()
5. setupModelData() parses through the QString, populating a QList<TreeItem*>
6. All these entries are stored relative to a root TreeItem (parent)
7. The TreeModel is then used by mainwindow in QTreeView.setModel() to display the tree to the user

Comment on your findings, with specific references to the design/code elements/file names/etc.:
Most of the analysis is done by RecordsManager. The TreeModel function are passed the user inputs from the GUI and thus demonstrates low coupling. Specific function calls, such as createQStringForPubs(), are made in the specific TreeModel subclass, whereas the universal calls (setupModelData()) are done in the parent class, demonstrating proper use of inheritance.

Title: Catching a 0-length Sort Fields List
Anticipated frequency of use: Low
Starting Point: User clicks to create a new sort order, enters a name, but forgets to pick the headers before clicking Save
Expected Outputs: An error dialog to the user, the CustomSort dialog remains, no new sort order is created
Inputs:
- Title of Sort Order -> *not* passed to QSortListIO
- Sort Order Headers -> *not* passed to QSortListIO
- Save button clicked -> error dialog appears, main dialog remains

Walkthrough
1. User clicks Save and on_buttonBox_accepted() is called
2. The appropriate switch case is entered
3. Function identifies that the first dropdown (header) is empty
4. Before calling QSortListIO, function breaks to create a QMessageBox
5. User is notified and because we broke out of the processing function only, we are still in our CustomSort dialog

Comment on your findings, with specific references to the design/code elements/file names/etc.:
System is robustly designed so that it catches an absent-minded user's mistakes without crashing. Because of setNext(), the user cannot select the first header in a separate dropdown than the first (i.e. the first dropdown must be filled to create a sort order). Because on_buttonBox_accepted() is designed so that none of the saving is actually before user input is verified, we don't run into logic error where the user doubles one of the headers or something.

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes inspected: ErrorEditDialog, RecordsManager
Performed by: Member E

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: *Compared fields, functions code-wise.*

Comment on your findings: *Some fields are misnamed for ErrorEditDialog.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Ran program.*

Comment on your findings:*Works as specified.*

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

☑ Yes ☐ No ☐ Partly (Can be increased)

Comment on your analysis: *Re-evaluated classes and their members.*

Comment on your findings: *Specific classes have proper members, and do not depend on other classes externally for internal functionaing.*

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

☐ Yes ☑ No ☐ Partly (Can be reduced)

Comment on your analysis: *Examined structure of classes.*

Comment on your findings: *MainWindow instantiates all other classes, but this is expected.*

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: *Examined classes.*

Comment on your findings: *Each of ErrorEditDialog, RecordsManager, MainWindow a different concern.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Examined headers.*

Comment on your findings: *Access specifications are in check.*

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: *Header analysis.*

Comment on your findings: *MainWindow obviously isn't. RecordsManager specifically has members for each of the different csv types (Grants, Publications etc). ErrorEditDialog in addition works only for the specified CSVs.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Class name analysis.*

Comment on your findings: *Clearly recognizable what each class does.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Examined code.*

Comment on your findings: *ErrorEditDialog has no comments, other two do.*

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) ☐ D on't know

Comment on your analysis: *Code analysis.*

Comment on your findings: *Many methods are generic and extensible.*

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: *Examined code.*

Comment on your findings: Code is functional and neat.

Depth of inheritance:
Do the inheritance relationships between the ancestor/decendent 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: *Header analysis.*

Comment on your findings: *Code does not inherit much (at most once).*

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: *Header analysis.*

Comment on your findings: *ErrorEditDialog, RecordsManager, MainWindow are not extended.*

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

Each scenario is described as follows:

1) Title of scenario
2) Anticipated frequency of use (high, normal, low)
3) End-user trigger (starting point) for the scenario.
4) Expected type of outputs.

5) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be “touched” by the scenario and producing the anticipated outputs.

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

Scenario 1:

i. Loading Publications
ii. High
iii. User opens Peachy Galaxy
iv. List of publications
v. User clicks Load File in the GUI under the Publications tab; user loads CSV file for Publications; Publications GUI get updated according to MainWindow specification

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

Scenario 2:

i. Fixing Errors
ii. User opens Peachy Galaxy
iii. User loads CSV
iv. User loads CSV file; edit/delete prompt comes up with erros; clicking edit opens dialog to udpate errors; clicking Save fixes the errors in the program (but not the original file)

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

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes Inspected: CSVReader.cpp, mainwindow.cpp, *TreeModel.cpp
Performed by: Member F

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: T*he mainwindow, csvreader, tree models are in the class diagram.*

Comment on your findings: *All methods are also 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: *The intended operations are all performed. However, there seems to be some inefficiency in terms of the numerous methods.*
Comment on your findings: *The number of methods in mainwindow can be reduced to improve performance and readability.*

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

☑ Yes ☐ No ☐ Partly (Can be increased)

Comment on your analysis: *All methods in mainwindow are specific to the mainwindow design, the tree models are specific to the data structures used for the data, csvreader is specific to reading and parsing the csv file, thus the methods within each class perform with high-cohesion.*

Comment on your findings: *Again, the encapsulated methods can be reduced and simplified to improve performance and readability.*

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

☐ Yes ☐ No ☑ Partly (Can be reduced)

Comment on your analysis: *Mainwindow is highly inter-dependent with the tree models, csvreader has low dependency with the other classes.*

Comment on your findings: *The high coupling might make it difficult to simplify mainwindow and treemodel classes.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Each unique dashboard is separated into its own tree model that responds to its separate methods in mainwindow.*

Comment on your findings: *However, the inter-dependency between mainwindow and the treemodels makes it difficult to identify the functionality of each method.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *The treemodel and csvreader methods used by mainwindow are public, mainwindow methods are private.*

Comment on your findings: *These are the proper access specifications with regard to the interdependency between the classes.*

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: *The csvreader can parse any csv file with the specified delimiter, which the treemodels will load. However, the header fields are specific to the four dashboards in this project. The methods and slots in mainwindow are specific to the gui layout.*

Comment on your findings: *Csvreader and treemodel have potential use for other applications. Mainwindow has very limited use for other applications.*

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Csvreader has very simple functionalities. The functionality of the methods in the treemodels are not immediately obvious as to which specific tree data structure is being used. The methods in mainwindow have well defined functionality with regard to the gui on_click, loadfile functions. The inter-dependency with the treemodels is easily identifiable but the functionalities are not easy to understand.*

Comment on your findings: The methods for the treemodels and mainwindow can be simplified for easier understanding.

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

☐ Yes ☐ No ☑ Partly (Can be improved)

Comment on your analysis: *Csvreader, treemodels have appropriate comments for ease of understanding. Mainwindow has appropriate comments for the first half of code, but insufficient comments for the second half of code.*

Comment on your findings: *Perhaps the code in the second half of mainwindow were added later/separately, they should include more comments for easier understanding.*

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: *Mainwindow layout, gui interactions can be easily changed. Csvreader can be easily changed. The tree models are well-defined and specific to the dashboard.*

Comment on your findings: *It will be difficult to enhance the tree models without simply making a new model. The mainwindow and csvreader can be easily updated and enhanced.*

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: *There is a slight noticeable delay once large numbers of data are loaded into the dashboard. The loading of data into records manager utilizes nested loops. Large number of methods might introduce inefficiencies.*

Comment on your findings: *The loading of data have potential for performance tuning. The large number of methods can be reduced to improve both efficiency and readability.*

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

☐ Yes ☑ No ☐ Partly (Can be improved)

Comment on your analysis: *Not a lot of inheritance, mainwindow inherits several QT widgets, treemodel inherits QabstractItemModel.*

Comment on your findings: *Inheritance is not an issue in understanding the functionalities of the code.*

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: *The classes do not have children classes.*

Comment on your findings: *There is no abstraction problem.*

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

Each scenario is described as follows:

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

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

1) Loading Presentations CSV File
2) High
3) Load file button
4) Populates and displays treeview of the presentation data
5) Walkthrough:
    1. User clicks on load file button and loads the presentation csv file
    2. mainwindow calls load_file() method
    3. mainwindow calls checkFile() to determine that it is a presentation file by calling csvreader to parse the file, and finally loads the data into a records manager
    4. mainwindow calls maketree() to create a tree of the data in records manager using the appropriate tree model, and then passes the tree to the gui treeview display
    5. User will now see the treeview of presentation data

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

END

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

- The purpose of this document is to assist in the inspection of object-oriented design.
- Under each question is a choice of answers; please choose one (either replace the box with a checkmark or highlight it)

☐ yes ☐ no ☐ partly, could be improved

- Two types of comments are required under each question. One is your analysis. The other is 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 of the system to be considered for inspection:

- With reference to Appendix B – Dashboard Screens, take Demo 1 feature, focusing on that part of the code that produces one Dashboard summary.
- Visualisation code is out of scope of this inspection.

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

Classes inspected: ErrorEditDialog, RecordsManager
Performed by: Member G

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: *Compare the content of the header and implementation files of the classes with their depiction in the class diagram.*

Comment on your findings: *Implementation of classes are accurately represented 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 of classes with respect to the dashboard summary requirements.*

Comment on your findings: *RecordsManager correctly builds the appropriate database from the loaded data, restructures it based on the required dashboard view parameters, and sends it the apprpriate tree model class which builds the dashboard view. ErrorEditDialog's error processing functionality performs beyond the scope of the dashboard view requirements.*

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

☐ Yes ☐ No ☑Partly (Can be increased)

Comment on your analysis: *Examine methods within the class implementation files.*

Comment on your findings: *ErrorEditDialog has very well-defined functions which performs specific tasks. However some of RecordsManager's methods are dashboard type-specific and implemented individually rather than overloaded. Moreover, the entire database analysis is encompassed in a single method, which although well-defined could be decomposed into smaller, more compact and task-specific methods.*

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

☐ Yes ☐ No ☑Partly (Can be reduced)

Comment on your analysis: *Identify the functional dependencies of the classes during their execution.*

Comment on your findings: *RecordsManager is where the bulk of the data processing takes place and thus has quite a few functional dependencies. In particular, it is interdependent with TreeModel and requries the latter's proper execution in order to analyze the data. It also defines new classes to facilitate data manipulation which are built from classes defined in TreeModel. ErrorEditDialog is a standard QDialog implemenation and has minimal coupling.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Interpret the requirements into a problem which can be decomposed into concerns. The resolution of each of these conerns embodied in the implementation of classes are scrutinized for interconnections.*

Comment on your findings: *The dashboard summary view problem can be broken down into 4 broad components: (1) reading the data from a CSV file, (2) checking for missing mandatory fields in the data, (3) loading the data into the database, and (4) creating the dashboard summary view. ErrorEditDialog addresses component (2) and RecordsManager addresses component (3). Each component is well-defined and seperate from the others.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Examine access modifiers in class header files.*

Comment on your findings: *ErrorEditDialog's only public method is its constructor and destructor. All other attributes and methods are private. This is important to maintain the integrity of the data ErrorEditDialog handles as well as to minimize unexpected behaviour. With respect to RecordsManager, all methods requiring interation with TreeModel are made public to ensure accessibility. All other attributes and methods, such as the method for analyzing and restructuring the data, are private. This is also a best practice for encapsulation of the data used by this class.*

Reusability:
Are the programmed classes reusable in other applications or situations?

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

Comment on your analysis: *Identify application dependencies and evaluate feasibility of class extrapolation to other situations.*

Comment on your findings: *RecordsManager is very application-specific. It is implemented with the ultimate goal of passing structured information to TreeModel in order to build the dashboard summary view. The level of change required for meeting the requirements of another situation would depend on the similarity between the data to be loaded, but in general would be quite high. On the other hand, ErrorEditDialog is very well encapsulated and is general enough to be reusable in any application where missing data needs to be either edited or discarded. It is also flexible enough to expand its data processing abilities to include other features such as automated data correction.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Evaluate ease of understanding of class implemenation files.*

Comment on your findings: *At the method level both ErrorEditDialog and RecordsManager are simple to understand. Functionality is well-defined and easy to identify.*

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

☑ Yes ☐ No ☐ Partly (Can be improved)

Comment on your analysis: *Determine quantity and quality of comments within code blocks with a high degree of complexity.*

Comment on your findings: *ErrorEditDialog's code complexity is quite low and thus only has comments at the beginning of each method. RecordsManager is a highly complex class and therefore has much more commenting for ease of understanding. Method parameters are well-defined and line-by-line comments are provided when necessary. This is most identifiable in the analyze() method.*

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) ☐ D on’t know

Comment on your analysis: *Attempt to predict amount of code enhancement required to make enhancements or updates.*

Comment on your findings: As mentioned above, *ErrorEditDialog is simple and general enough to add functionality. There is also enough scope in RecordsManager to allow easy enhancements, although most functionality here is well-established and updates would mostly be directed to improving efficiency.*

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: *Gauge algorithm efficiencies within time and space constraints of the data.*

Comment on your findings: *Both in RecordsManager and ErrorEditDialog, methods use minimal conditional and loop constructs. Only in the analyze() method has a single recursive call. Run-time efficiencies are at most proportional to the square of the amount of input data (i.e. number of records in CSV file).*

Depth of inheritance:
Do the inheritance relationships between the ancestor/decendent 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: *Examine generalizations in class diagram.*

Comment on your findings: *ErrorEditDialog has a single parent class Qdialog and no child classes. RecordsManager has no child or parent classes.*

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

☐ Yes ☑ No ☐ Partly (Can be improved)

Comment on your analysis: *Examine generalizations in class diagram.*

Comment on your findings: *Neither ErrorEditDialog nor RecordsManager 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:

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

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

RecordsManager has no interaction with the end-user.

Title: Edit Records with Missing Data Fields
Anticipated frequency of use: High
Starting Point: User is prompted with message asking whether to edit or discard records with missing data fields
Expected Outputs: A correctly formatted dashboard summary view with modified records no longer containing missing data fields
Inputs:

- Edit button clicked by user -> passed to MainWindow
- Empty fields filled in by user -> passed to ErrorEditDialog
- Save buton clicked by user -> passed to ErrorEditDialog

Walkthrough

1. MainWindow returns the user's prompt, creates a new ErrorEditDialog and executes it.
2. ErrorEditDialog creates and populates a table with the records containing missing fields. The cells representing the missing entries are empty and coloured red.
3. When the Save button is clicked, ErrorEditDialog checks the missing fields to ensure they are no longer empty. If a single entry is still empty, ErrorEditDialog displays an error saying so. Otherwise, it succesfully returns back to MainWindow.
4. The newly modified records are included in the loaded data from which the dashboard view is created back in MainWindow.

Comment on your findings, with specific references to the design/code elements/file names/etc.:
ErrorEditDialog only populates the table with records containing missing manadatory fields, as expected. It manipulates all the data internaly as opposed to interacting with RecordsManager, a sign of encapsulation and low coupling. It also makes proper use of inheritance by calling QDialog's exec() and accept() methods instead of implemenating its own methods for running and succesfully returning.

Title: Cancel Editing Records with Missing Data Fields
Anticipated frequency of use: Low
Starting Point: User is prompted with message asking whether to edit or discard records with missing data fields
Expected Outputs: A correctly formatted dashboard summary view without records containing missing data fields
Inputs:

- Edit button clicked by user -> passed to MainWindow
- Empty fields filled in by user (optional) -> passed to ErrorEditDialog
- Cancel buton clicked by user -> passed to ErrorEditDialog

Walkthrough

1. MainWindow returns the user's prompt, creates a new ErrorEditDialog and executes it.

2. ErrorEditDialog creates and populates a table with the records containing missing fields. The cells representing the missing entries are empty and coloured red.
3. When the Cancel button is clicked, ErrorEditDialog unsuccesfully returns back to MainWindow.
4. The records are discarded from the loaded data from which the dashboard view is created back in MainWindow.

Comment on your findings, with specific references to the design/code elements/file names/etc.:
ErrorEditDialog unsuccesfully returns in a graceful fashion as opposed to crashing unexpectedly. It also makes proper use of inheritance by calling QDialog's exec() and reject() methods instead of implemenating its own methods for running and unsuccesfully returning.

END.