Computer Science 3307A – Object-Oriented Design & Analysis
2016-17 Fall Semester
University of Western Ontario, London ON, Canada
N6A 3K7

Scorpius

Group: Team Scorpius
Daniel Deng
Jameel Kaba
Taylor LeBlanc
Winston Leung
Taha Nasir
Stephane Vaillancourt
Pengwai (Dave) Zhang

Instructor: Nazim Madhavji

TABLE OF CONTENTS

1 Requirements Implemented

All requirements (Both Mandatory and Stretch) were given to us by the client (Schulich School of Medicine and Dentistry)

1.1 Mandatory Requirements

1.1.1 Sort by Division
Included in the data files (.csv) are a column for “Department” and “Division” (or “Primary Domain”). We added the ability to sort members by these new data point(s) as a default sort order in addition to the original default sort order

1.1.2 Sort by User Selected List
After loading a data file, the user can select multiple pieces of data, and shows only the relevant data in the graphs.

1.1.3 Added 2 new graphs
After loading the data, the user can view a line graph, and a scatter plot, in addition to the pie and bar graphs that already exist.

1.1.4 Navigation of Erroneous Entries
If when loading the data, there are errors, the user can navigate through the erroneous entries on the file and fix them. If not all marked fields are edited, then a warning message will be displayed. If cancel is clicked all errors are discarded.

1.1.5 Save a Session
After starting the program, if there was a previous session it will ask the user if they want to continue it. If the user hits ok, it will continue the previous session, else it will delete it.

1.1.6 Deliver a user proof install file
We’ve delivered a simple install, all you’ve got to do is follow the instructions and a folder containing the file will be created

1.2 Stretch Requirements

1.2.1 Change Modify Sort Order

Allows user to edit the current selected sort order. The default and division sort orders cannot be modified (nor deleted). When the button is pressed, a dialog box opens up to allow the user to select fields to order by. The modified sort order replaced the old one and is saved.

1.2.2 Implement Advance Searching

Advanced searching: allows users to search the table for a particular member name. When the search button is pressed a dialog open up where the user puts in the first and last name. Clicking OK results in the member name being selected in the table as well as the default chart being displayed for that member name.

1.2.3 Mac OS Compatibility

We've made a separate install file for Mac users so that they can run the program as well, this install is in a different folder, compared to the Windows install.

1.2.4 Supply a Training Video

We've included a training video to help new users with using the system. This video goes over many of the features of the program.

2 Test Cases:

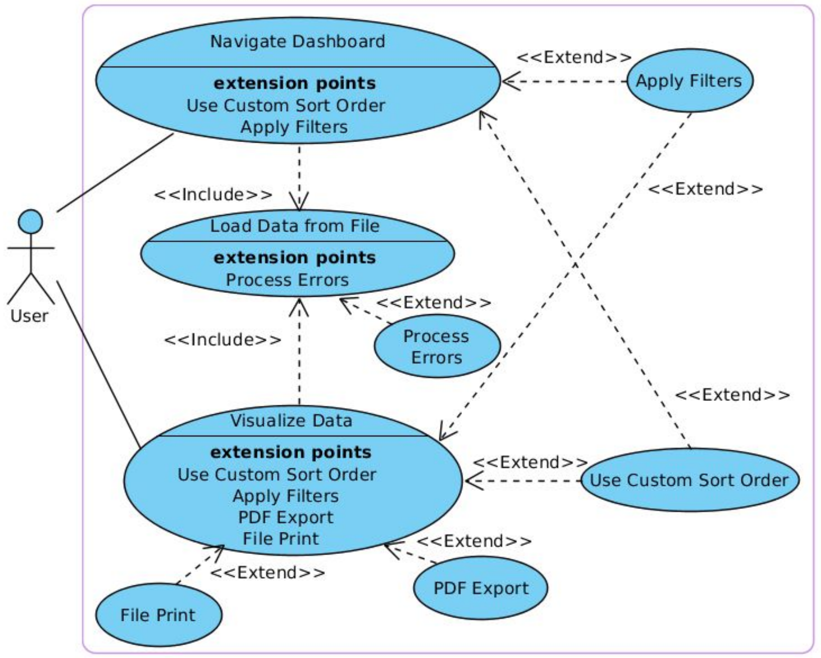
2.1 Test Case Matrix:

Below is a simplified version of the test matrix, for a more detailed representation refer to SCORPIUS_TEST_CASES.xlsx

		Requirements						
ID	Unit to test	Visualization	Error Processing	Custom Sorting	Loading Data	Data Manipulation	Pass/Fail	Comments
1	Line Graph	X					Pass	
2	Scatter Plot	X					Pass	
3	Able to Save Session				X	X	Pass	Needs dat folder with text file
4	Sort by Division			X			Pass	
5	Sort by User selected List			X			Pass	
6	Navigation of Erroneous Entries		X				Pass	Saves as test.csv
7	Modify Sort Order			X	X	X	Pass	Can’t edit default
8	Advance Searching			X			Pass	Need First and Last name of person to be searched

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] [Alternate Course D: All entries filled out]

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 saves the records with the filled out entries into a new .csv file called test [Return to Main Success Scenario step 5]

Alternate Course C: User clicks Cancel button

1. The system discards records with missing mandatory entries.

2. The system closes the error processing screen.

Alternate Course D: All entries filled out *

1. The system saves the records with the filled out entries into a new .csv file called test

   [Return to Main Success Scenario step 5]

3.1.3 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. [Alternate Course D: User selects to modify the chosen sort order]

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

Alternate Course A: User selects existing sort order

1. [Return to Main Success Scenario step 6]

Alternate Course B: User does not enter name

1. The system displays an error message.

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

Alternate Course C: User clicks Cancel button

1. The system closes the new sort order screen.

Alternate Course D: User selects to modify the chosen sort order

1. The user clicks the modify sort order button [Return to Main Success Scenario step 3]

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 one of the Graph radio buttons.

6. The system displays the appropriate graph for 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.

3.1.9 Sort by User Selected List

Sorting by a user selected list (sea level) Main Success Scenario:

1. The user loads the data from file via the use case 3*.1.3 Loading data from file.*

2. The system displays the dashboard summary view.

3. The user clicks on different data elements by pressing control and clicking that element.

4. The system displays the summary based on the header the user clicked.

3.1.10 Using advanced Searching
Using advanced searching (sea level) Main Success Scenario:

1. The user loads the data from file via the use case 3*.1.3 Loading data from file.*

2. The system displays the dashboard summary view.

3. The user clicks on the search button

4. The user fills out the search info by filling in the first and last name of the person they're searching for [Alternate Course A: User doesn't fill in both the first and last name] [Alternate Course B: User presses Cancel]

5. The system highlights the info for the person specified in the search

Alternate Course A: User doesn't fill in both the first and last name

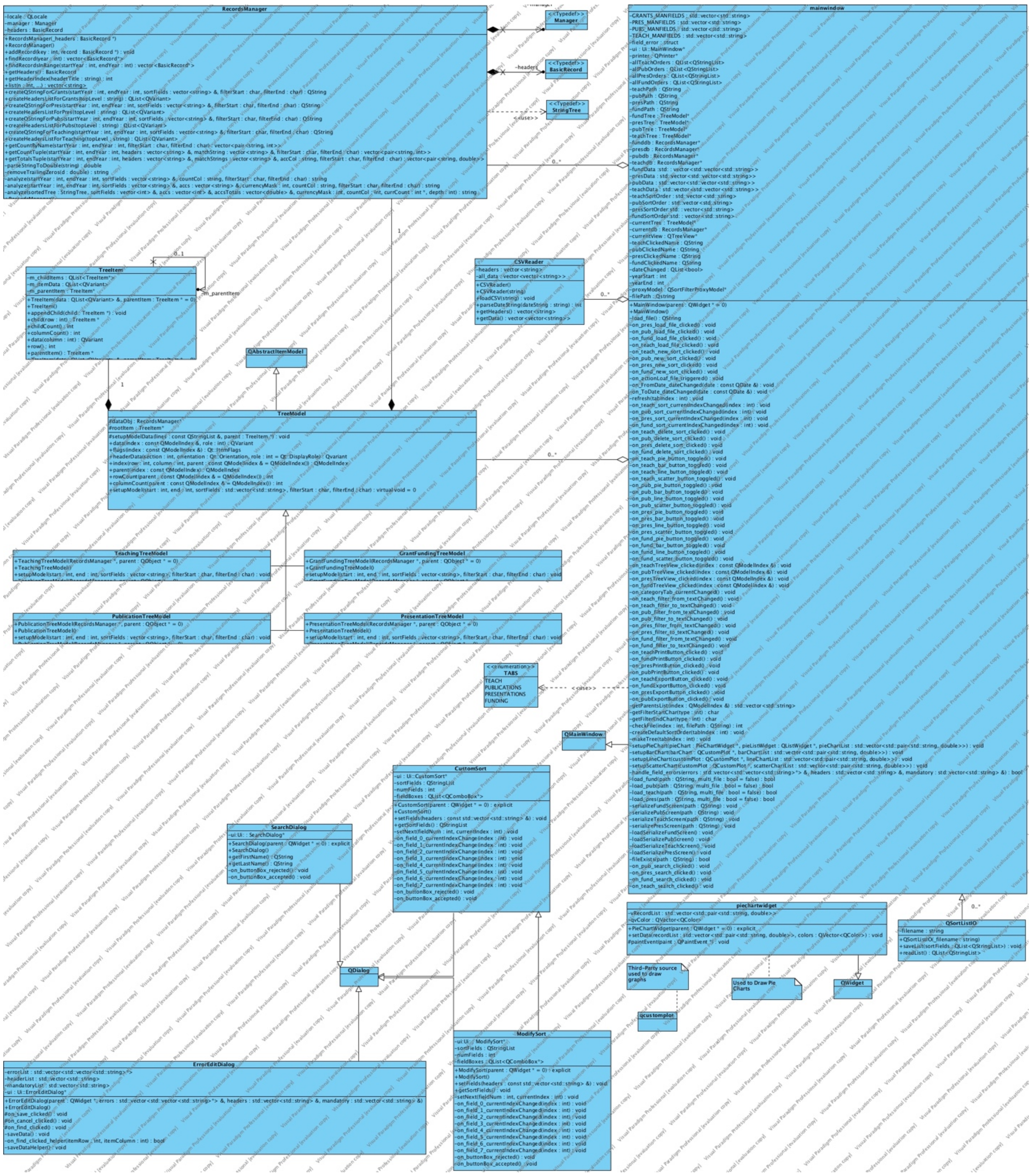
1. The system provides an error message to the user

2. The user clicks ok to go back to filling in the search info. [Return to Main Success Scenario step 4]

Alternate Course B: User presses cancel

1. The system closes the search box. [Return to Main Success Scenario step 2]

3.2 Class Diagram



Above is the enhanced 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.

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 and save 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.

3.3.11 ModifySort

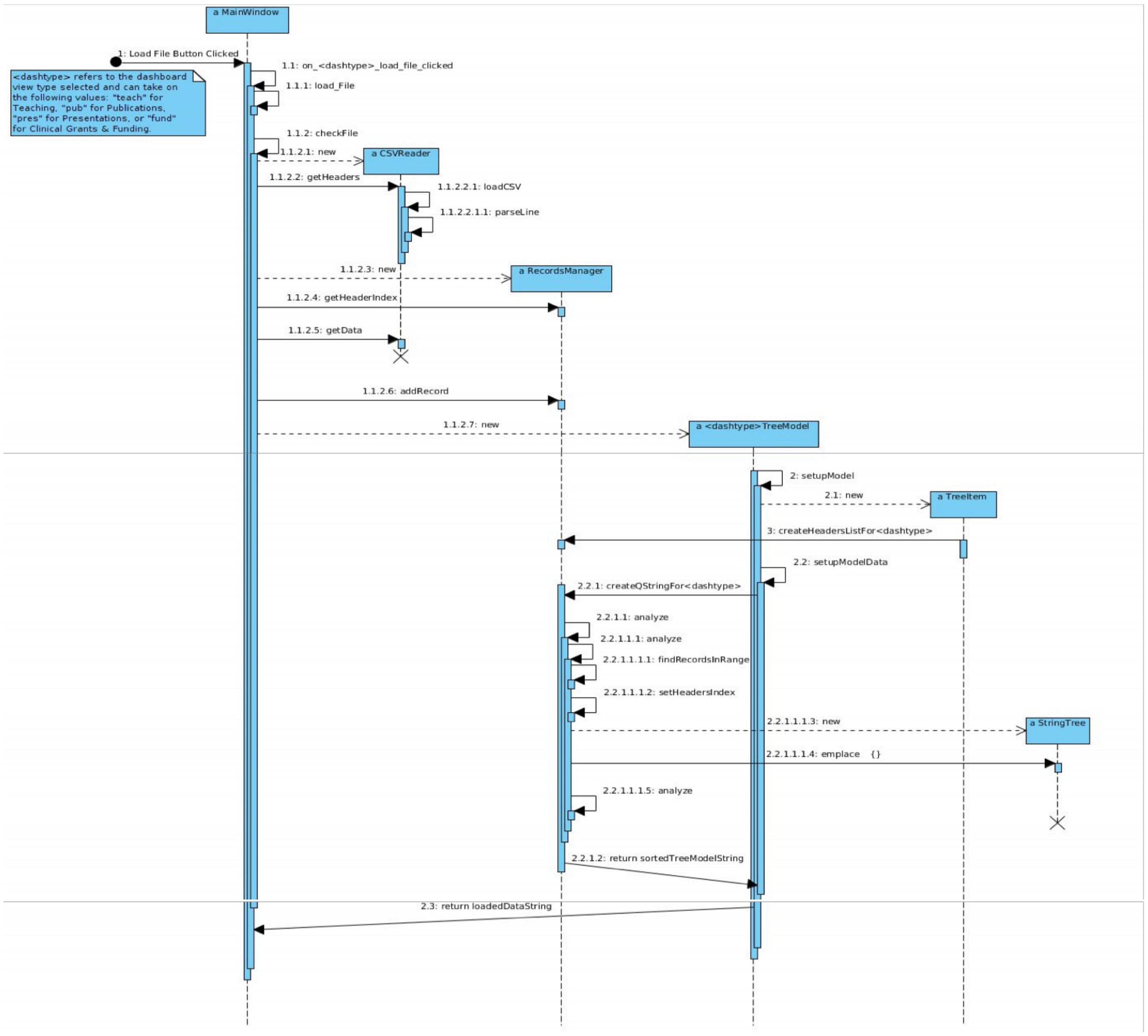
ModifySort is also created during the execution of MainWindow, representing a dialog for the user to customize a CustomSort object ModifySort is a sublass of QDialog so as to facilitate integration with the other Q-type graphical user interface components. Modify Sort is also a subclass of CustomSort so as to allow the user to edit the custom sort.

3.3.12 SearchDialog

SearchDialog is another object created during the execution of MainWindow, representing a dialog for the user to perform an advanced search. SearchDialog is a subclass of QDialog so as to facilitate integration with the other Q-type graphical user interface components.

3.3 Sequence diagrams

3.3.1 Sequence diagram – loading data from file


a MainWindow
1: Load File Button Clicked
<dashtype> refers to the dashboard view type selected and can take on the following values: "teach" for Teaching, "pub" for Publications, "pres" for Presentations, or "fund" for Clinical Grants & Funding.
1.1: on_<dashtype>_load_file_clicked
1.1.1: load_File
1.1.2: checkFile
1.1.2.1: new
a CSVReader
1.1.2.2: getHeaders
1.1.2.2.1: loadCSV
1.1.2.2.1.1: parseLine
1.1.2.3: new
a RecordsManager
1.1.2.4: getHeaderIndex
1.1.2.5: getData
1.1.2.6: addRecord
1.1.2.7: new
a <dashtype>TreeModel
2: setupModel
2.1: new
a TreeItem
3: createHeadersListFor<dashtype>
2.2: setupModelData
2.2.1: createQStringFor<dashtype>
2.2.1.1: analyze
2.2.1.1.1: analyze
2.2.1.1.1.1: findRecordsInRange
2.2.1.1.1.2: setHeadersIndex
2.2.1.1.1.3: new
a StringTree
2.2.1.1.1.4: emplace {}
2.2.1.1.1.5: analyze
2.2.1.2: return sortedTreeModelString
2.3: return loadedDataString

Above is the sequence diagram for the use case scenario 3.1.1 “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.

3.3.1.1 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.

3.3.1.2 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.

3.3.1.3 System verifies file is of proper type

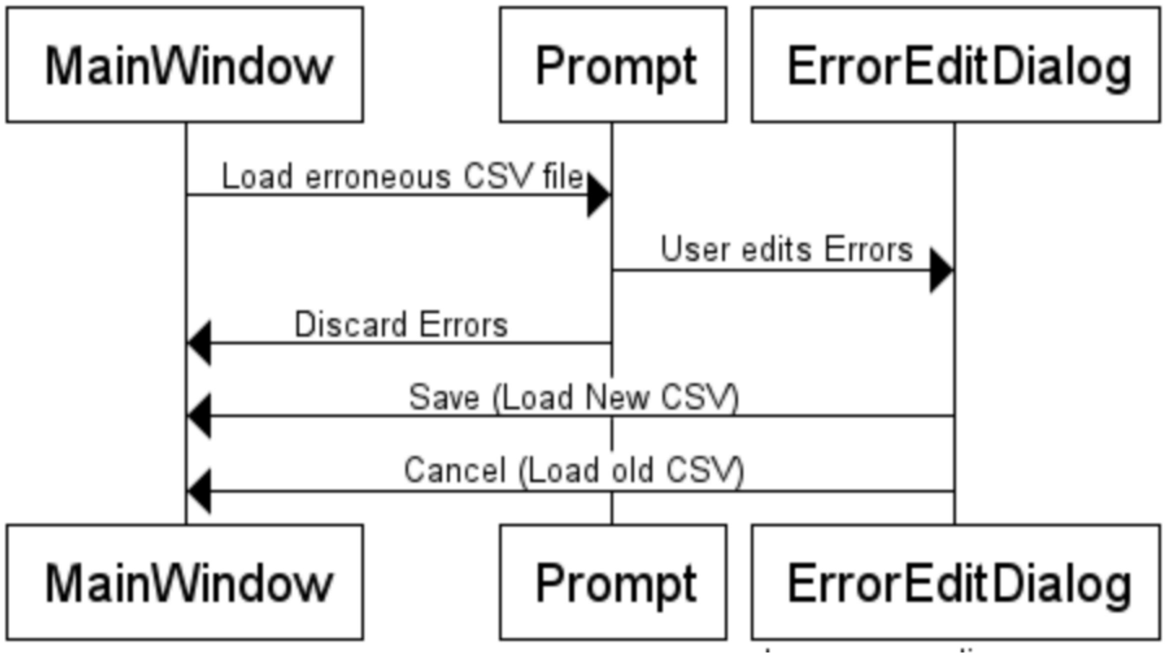
This step is accounted for in the MainWindow “checkFile” method self-call, which first checks

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.

3.3.1.4 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.

3.3.2 Sequence Diagram – loading data from file, fixing errors



Above is the sequence diagram for the use case scenario 3.1.2 “Error Processing”

3.3.2.1 User loads error filled CSV file

This step is represented by the message “Load CSV with Errors”. This runs through the entire loading process explained in use case scenario 3.1.1.

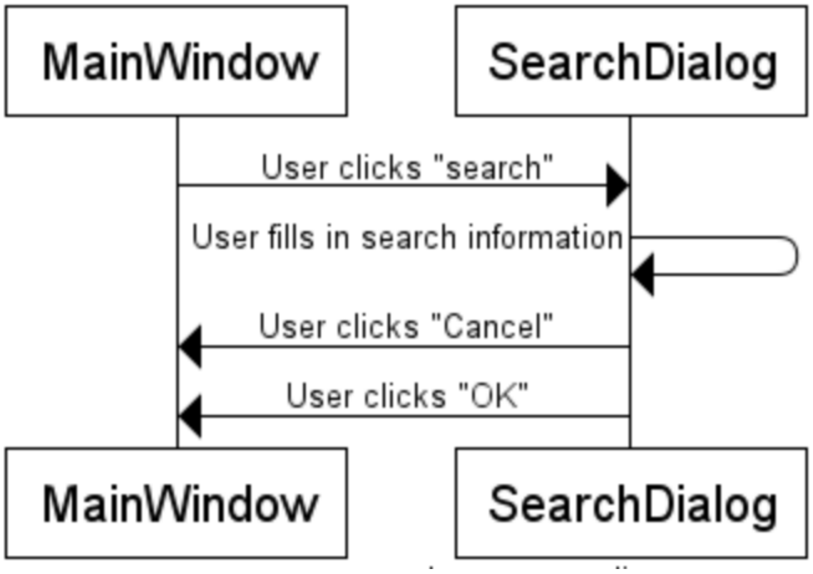
3.3.2.2 System prompts user whether they would like to fix the erroneous file.

The system will display a message to the user. If the user decides to edit the file, the system will open up an ErrorEditDialog to allow the user to fix the errors. If they decide not to fix the errors, then the erroneous entries are discarded and whatever is left will be displayed on screen.

3.3.2.3 User saves or cancels changes made to the erroneous file. ‘

If the user decides to save the changes made, then the new file will be saved as “test.csv” and will be loaded and displayed on screen. If they hit cancel then all the changes (if any) will be deleted and the old csv file will be loaded and displayed.

3.3.3 Sequence Diagram – using advanced searching



Above is the sequence diagram for the use case scenario 3.1.10 “Using advanced searching”

3.3.3.1 User clicks “search” button

When the user clicks the search button, the MainWindow calls upon the appropriate search method (on_---_search_clicked where --- represents the appropriate tab). This will open up a SearchDialog for the user to search with.

3.3.3.2 User fills in the required information and clicks “OK”

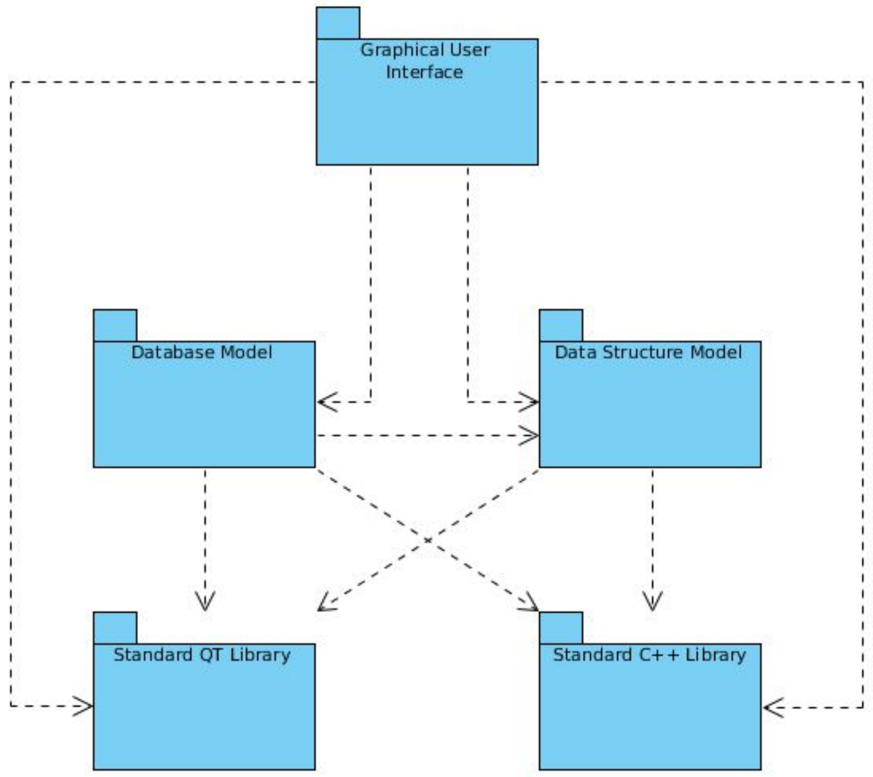
When the SearchDialog appears it will request that the user fills in the first and last name of the person they are searching for. If the user were to fill out the required information, and press “OK” then the SearchDialog will search for the person and return to the MainWindow highlighting the person who the user was searching for.

3.3.3.3 User clicks “Cancel”

If the user were to click cancel at any point, the SearchDialog will close and the user will not get any search results and will be back to where they were at the beginning of this sequence.

3.4 Package Diagram

Below is the package diagram, it is the same as the initial version of 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 Graphical User Interface

This package contains the classes relevant to the user interface such as MainWindow, PieChartWidget and CustomSort. These classes control the behavior of the graphical componenets (dialog boxes, buttons, widgets, etc…) during user interaction and therefore relies on the standard QT and C++ libraries. In addition, Main Window needs to create a 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) 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

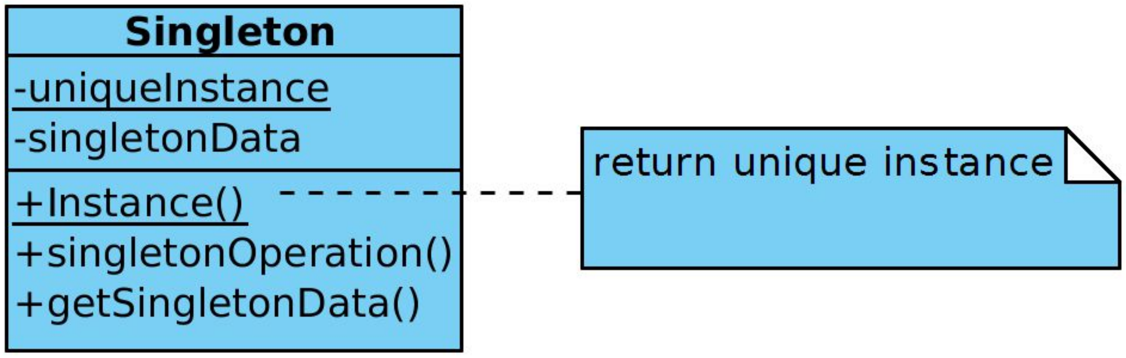
This package encompasses the abstract TreeModel class, which makes use of the TreeItem class, and its implementations for each of the implemented dashboard views: GrantFundingTreeModel, PresentationTreeModel, PublicationTreeModel and TeachingTreeModel. TreeModel does make use of RecordsManager when building 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 well documented stand-alone packages which are used by, but not in any way dependent on, the other packages in our application.

4 Design Patterns
Scorpius makes use of the same two design patterns as Peachy Galaxy, Singleton and Prototype.

4.1 Singleton
The reasoning behind a Singleton pattern is to ensure that a class will only have one instance, and that it has a global point of access. One way to do this is by creating global variables such that it allows an object to be accessible by others. This however does not prevent the class from instantiating multiple objects. A better way to make a Singleton is to make class responsible for it's actions, or responsible for keeping track of itself. That way the class can intercept requests to create new objects thus allowing no other instance to be created. The class can also provide a way for others to access the instance. This is the reasoning and motivation behind the Singleton pattern as illustrated below.



1) Controlled access to sole instance:

- Because the class is a Singleton it encapsulates its own instance, giing it strict control over how and when the client accesses it.

2) Reduced name space:

- Compared to global variables, the Singleton pattern is an improvement. It's an improvement as it avoids polluting the name space with global variables that store sole instances.

3) Permits refinement of operations and representation:

- Singleton classes can be sub classed, this in turn makes it easy to configure an application with an instance of this extended class. This is done as anyone can configure the application with an instance of the class required at run-time.

4) Permits a variable number of instances:

- The Singleton pattern makes it easy for someone to change their mind and allow more than one instance of the Singleton class. This same approach can be used to control the number of instances that the application uses. The only change that needs to be made is within the operation that grants access to the Singleton instance.

5) More flexible than class operations:

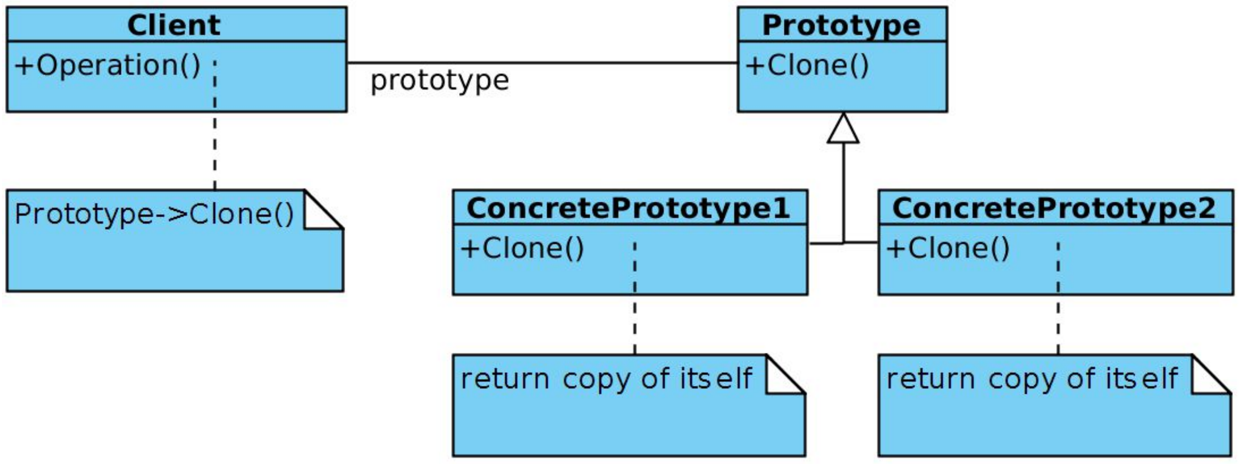
- Packaging a Singleton's functionality another way allows it to use class operations like static member functions in C++. Though by doing this, it becomes harder to change the design to allow more than one instance of a class. As well, subclasses can't override static member functions in C++ polymorphically. This is due to static member functions never being virtual.

Though we did not require a variable number of instances, we acknowledge that we can modify our Singleton instance if necessary. The Singleton design pattern is used in the implementation of MainWindow, which controls the runtime behavior of the graphical user interface. This design was used as a result that there should only be one instance of the MainWindow and it must be accessible to clients and users from a well-known access point.

The biggest issue with the Singleton pattern that was considered during implementation, was to ensure that there is only one instance. The Singleton pattern makes this sole instance a normal instance of the class, but the class was written so that only one instance can ever be created. A common way of accomplishing this is to hide the operation that creates the instance behind a static member function or a class method. This guarantees that only one instance is ever created. The 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 the Singleton is created and initialized before being used. This can be done by defining 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 object to create a prototypical instance, and allows us to create new object by copying this prototype. As an example, suppose that our system has many objects that are only slightly different from one another and exhibit almost identical behavior. Knowing the composition of the object is a flexible alternative to sub classing. We want the framework for our software takes advantage of this to parameterize instances based on the type of class it will create. The answer to our problems lies in making a framework that creates a new instance by copying an instance of the desired class. This instance is called a prototype and its structure is show below.



The Prototype pattern has many of the same consequences in common with other creational design patterns. The concrete product classes are hidden from the client, which reduces the number of names that the client would need to know about. These patterns also allow a client to work with application-specific classes without the need to modify them. Below you will find some additional benefits that are unique to prototypes.

1) Adding and removing products at run-time:

   - Prototypes allow us to incorporate a new concrete product class into a system by registering a prototypical instance with the client. This allows some flexibility when compared to other creational patterns as the client can install and delete prototypes at run-time.

2) Specifying new objects by varying values:

   - In highly dynamic systems new behaviors are allowed through object composition and not through defining new classes. This is done as one can define new kinds of objects by instantiating existing classes and setting the instances to be prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype. This type of design allows users to define their own new classes without the need to program them. In fact, you could compare cloning a prototype similar to instantiating a class. The prototype pattern can reduce the number of classes a system needs.

3) Specifying new objects by varying structure:

- Many applications tend to build objects from parts and subparts. An example would be our graphical user interface as it is built from widgets, dialog boxes, radio buttons, etc. These types of applications are often allowing one instantiate complex, user-defined structures use a specific widget again and again. The Prototype pattern supports this as well. We simply add the widget as a prototype to the palette of available graphical user interface elements.

4) Reduced subclassing:

- Alternatives to the Prototype pattern, like the Factory method produce a hierarchy of Creator classes the are parallel to the product class hierarchy. The Prototype pattern allows one to clone a prototype rather than asking the Factory method to make a new object. So, one does not need a Creator class hierarchy at all. This benefit applies mostly to languages such as C++ which don't treat classes as first-class objects.

5) Configuring an application with classes dynamically:

- Some run-time environments, like that of our application, allows one to load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities using a language like C++. For an application that wants to create instances of a dynamically loaded class will not be able to reference its constructor statically. The run-time environment creates an instance of each class automatically when it is loaded, and it registers the instance with a prototype manager. That way the application can ask the prototype manager for instances of classes which weren't originally linked with the program.

We chose to use the Prototype pattern as we felt that our system needed to be independent of how it creates, composes, and represents its products. This was considered as the best choice for the TreeModel class, the instantiation of which is specified at run-time through dynamic loading. We wanted to avoid building a class hierarchy of factories, which is why we did not opt for the Factory method. Our MainWindow, PieChartWidget, and CustomSort classes instances can have one of only a few different combinations of states. It was more convenient to install a corresponding number of of prototypes and clone them instead of instantiating the class manually, every time with the appropriate state.

The main issue with the Prototype pattern is that each subclass must implement the Clone operation, which could end up being very difficult. For example, adding the Clone operation 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 considered as a prototype manager. When the number of prototype in a system isn't fixed, we keep a registry of available prototype. Clients will not manage the 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 have operations for resetting/setting key pieces of state. In this case, then an initialization operation that takes initialization parameters as arguments and sets the clone's internal state had to be introduced. An example of this is the setupModel() operation in TreeItem.

5 Implementation in C++

5.1 Does the code satisfy the design?

Yes, the code does satisfy the design.

Everything implemented has a connection to MainWindow as that is where most of the work is done.

For adding the 2 new graph types they've been implemented as methods in MainWindow. The methods are setupLineChart and setupScatterChart, they make use of methods from QCustomSort to create and draw the graphs.

Adding the ability to save a session is also implemented as methods in MainWindow. The method to save a session is serialize___Screen and to load is loadSerialize___Screen, where ___ is the appropriate tab to save or load.

In order to add the ability to sort by a member's division we've implemented it by adding it into the method createDefaultSortOrder in MainWindow. We decided to add it into the createDefaultSortOrder method rather than making a new method as it allows the code to flow nicer, rather than going from one method to another.

To add the ability to sort by a user selected list we needed to include QSortFilterProxyModel and added it into MainWindow. Using QSortFilterProxyModel allows us to sort and filter through the data easily. So, when a user selects a list, the proxy model will sort and filter through the data before we display the new order of data.

In order to navigate through erroneous entries, we needed to make changes in the ErrorEditDialog files. The MainWindow is what calls the ErrorEditDialog if there is an error in the file. We use QTableWidgetItem to allow us to navigate through the erroneous entries. We currently have the ErrorEditDialog save methods to save the fixed file as test.csv if the user wants to save their changes.

Allowing the user to edit sort orders is implemented through ModifySort files and also methods in MainWindow called on_---_modify_sort_clicked() where --- is the appropriate tab. The ModifySort takes provides the user with a dialog box where they can edit an existing custom sort that isn't default or division sort. The fields from the dialog box once filled out, modifies the custom sort accordingly.

Finally, we implemented advance searching in this deliverable. Advance searching is implemented through the SearchDialog files and methods in MainWindow known as on_---_search_clicked where --- is the appropriate tab. The SearchDialog files creates a dialog box for the user to enter the search info and handles the logic related to that dialog box. In MainWindow is where the SearchDialog results are implemented. The result provided from SearchDialog will be highlighted in the MainWindow display.

5.2 Code Files

Refer to folder "Scorpius_Code"

6 Development Plans:

6.1 Timeline

Refer to the Excel Sheet "Scorpius_Timeline" for a nicer representation

ACTIVITY	PLAN START	PLAN END	ACTUAL START	ACTUAL END
Phase 1 system documentation understood	17-Oct	19-Oct	17-Oct	19-Oct
Phase 1 system's operational aspects understood	17-Oct	21-Oct	17-Oct	22-Oct
Improvements to Phase 1 system identified/logged	19-Oct	22-Oct	19-Oct	24-Oct
Test Cases for Phase 1 system	20-Oct	21-Oct	21-Oct	21-Oct
Demo 1 Enhancement requirements determined	24-Oct	24-Oct	24-Oct	24-Oct
Demo 1 enhancement requirements allocated	24-Oct	24-Oct	24-Oct	24-Oct
Add 2 new graph types (1 line, 1 other)	24-Oct	14-Nov	24-Oct	14-Nov
Able to Save session	24-Oct	14-Nov	24-Oct	14-Nov
Add Ability to sort by number division	24-Oct	28-Oct	24-Oct	29-Oct
Add ability to sort by user selected list	24-Oct	28-Oct	24-Oct	29-Oct
Demo 1 Test Cases	28-Oct	31-Oct	30-Oct	31-Oct
Submission of Demo 1	31-Oct	31-Oct	31-Oct	31-Oct
Demo 2 Enhancement Requirements Determined	31-Oct	31-Oct	30-Oct	30-Oct
Demo 2 Enhancement Requirements Allocated	31-Oct	31-Oct	31-Oct	31-Oct
Final Submission Requirements determined	31-Oct	31-Oct	30-Oct	30-Oct
Navigation of Erroneous Entries	31-Oct	10-Nov	24-Oct	14-Nov
Change Modify Sort Order	31-Oct	10-Nov	31-Oct	10-Nov
Navigation of Erroneous Entries	31-Oct	10-Nov	31-Oct	10-Nov
Final Submission Requirements Allocated	14-Nov	14-Nov	14-Nov	14-Nov
Fix All Errors	14-Nov	28-Nov	14-Nov	28-Nov
Deliver a User-Proof Install	14-Nov	28-Nov	14-Nov	22-Nov
Demo 2 Test Cases	20-Nov	23-Nov	20-Nov	23-Nov
Submission of Demo 2	23-Nov	23-Nov	23-Nov	23-Nov
Verify Code Deck for Windows 7	24-Nov	28-Nov	24-Nov	01-Dec
Supply a Training Video	24-Nov	26-Nov	24-Nov	01-Dec
Verify Code Deck on Mac OS	24-Nov	28-Nov	24-Nov	23-Nov
PREPARING FINAL SUBMISSION	01-Dec	01-Dec	01-Dec	01-Dec
FINAL SUBMISSION	07-Dec	07-Dec	07-Dec	07-Dec

Means Major Point

6.2 Agent Task View

The agent task view can be found in the pdf “Agent-Task_View_Scorpius”.
The agent task view was packaged separately for privacy reasons

7 Lessons Learnt and Retrospective Analysis:

7.1 Top three lessons learnt (Technical Issues)

1. When understanding the old code, there weren't many comments which made it confusing trying to figure out what certain parts of code were actually doing. In hindsight, we would try to spend more time playing with the code to understand it as well as adding in comments once we actually knew what the code does.
2. We had trouble initially testing the system as we had never used QT before, and we didn't know how to actually use QT testing. In hindsight, we would definitely spend more time learning and practicing how QT testing works rather than spending a few hours rushing to learn it.
3. The last technical issue we ran into was creating the install file. Trying to figure out what plugins are necessary took quite a while and a lot of guessing and google searching. In hindsight, we could have and probably should have asked for help rather than trying to tough it out and figure it out ourselves.

7.2 Retrospective Analysis

Our team was for the most part, well organized and managed. We knew what we had to do and when it had to be done. Though for the first few deliverables we cut it pretty close in regards to submitting it on time, we started working sooner and getting as much done as possible for each deliverable. We would have weekly meetings for about half an hour every Thursday. Though as time passed we ended up just having a brief check-in on Saturdays, and as a result we ended up getting more work done.

In regards to our project performance, we got all of the mandatory goals done and majority of the stretch goals done. The one stretch goal we could not complete was for "Editable Text Fields". This was not completed due to merge conflicts that could not be resolved due to a lack of time and awareness. In hindsight, we should have ensured it was integrated in the main program when it was done rather than waiting until the last few days.

We had one group member just completely cut off all communication with us after we submitted the stage 1 system. As a result, they did not do anything for the last 2 deliverables.

Something's that we could have done differently are noted below:

- Not procrastinate for the first couple deliverables
- Work more as a team rather than separately on our own
- Have one major meeting at the start of each deliverable to assign tasks, and have weekly check-ins to ensure that everyone making good progress.

7.3 Value

Our group found the experiences from this project listed below beneficial towards our learning

- Experience with the programming language C++
- Experience using the Qt libraries
- Experience in evolving an existing, real system
- Experience in dealing with a real customer
- Experience with design inspections
- Experience in project documentation
- Experience in developing the project with demos being produced at each stage

8 Code and Design Inspections

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

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

Scope and process:

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

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

Classes Inspected: QSortListIO, CustomSort, TreeModel
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 class diagrams to the implemented code.
Comment on your findings: Comparison checks out.

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.

Comment on your findings: All Classes work.

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

☐ Yes ☐No [✓ ☐Partly (Can be increased)]

Comment on your analysis:

Comment on your findings: TreeModel has many cohesive functions. CustomSort is pretty good, but there are a lot of switch statements which makes me think that is could be improved. QSortListIO is cohesive as it performs only one task.

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

☐ Yes [✓ ☐No] ☐Partly (Can be reduced)

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

Comment on your findings: No global variables. All classes contain a minimal number of includes. Each class can be understood on its own, without looking at 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: TreeModel has an abstract function (setupModel()) which is implemented in four separate classes, one for each dashboard type. CustomSort uses switch statements instead, this could be changed to four different subclasses instead. QSortListIO works as intended.

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 adjustments.

Comment on your findings: One could reuse TreeModel for another dashboard type by simply implementing a new setupModel(). CustomSort could be reused, but it would take a few changes. QSortListIO could be reused, but may require some changes.

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

[✓ ☐ Yes] ☐No ☐Partly (Can be improved)

Comment on your analysis: Based off of the names of functions, is it easily understandable.

Comment on your findings: Function names in TreeModel and CustomSort are descriptive enough for anyone to understand.

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

☐ Yes ☐No [✓ ☐Partly (Can be improved)]

Comment on your analysis: Look for the presence of comments, especially when the code is complex

Comment on your findings: TreeModel is well commented; however, CustomSort lacks comments and could use some in parts of the codes.

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: Considered adding a new dashboard and what would be required.

Comment on your findings: Potentially end up making another child for TreeModel, or finding a way to combine all of the children into one super child class.

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

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. QSortListIO has no nested loops and uses limited calls to other functions.

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved)

Comment on your analysis: Numerically determine the depth of inheritance.

Comment on your findings: CustomSort and QSortListIO both inherits QDialog which is the Qt class for dealing with dialog windows. The individual TreeModel classes inherit from TreeModel which inherits from QAbstractItemModel.

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: Counted the number of children classes

Comment on your findings: CustomSort has no children, while TreeModel is inherited by four subclasses, one for each dashboard type.

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

Each scenario is described as follows:

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

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

Title: Sort by division
Anticipated frequency of use: normal
Starting point: Click on "division" sort order
Output: Data sorted by division
Input: User selects custom sort order "division" → TreeModel

Comment on your findings, with specific references to the design/code elements/file names/etc.: When the program is initialized it will start by creating the default and division custom sort orders via CustomSort, which are saved and loaded via QSortListIO. When the user wants to sort by division, the MainWindow will call the TreeModel to rearrange the data according to the division.

Title: Sort by user selected list
Anticipated frequency: normal
Starting Point: Click on a tree model header

Output: TreeItems sorted either ascending or descending based on the column clicked
Input: User clicks a header → TreeModel

Comment on your findings, with specific references to the design/code elements/file names/etc.: When the user clicks on the header, MainWindow will create a new TreeModel based off of the header selected which will sort the TreeItems in either ascending or descending order.

(Note: expand here as necessary for each scenario)

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

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

  ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question: (i) your analysis, and (ii) your finding (in the form of a comment). The analysis would typically show how you arrived at the finding.
- Add new lines as necessary for your analysis or findings.

Scope and process:

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

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

Classes Inspected: TreeItem, RecordsManager, MainWindow*

Structural correspondence between Design and Code:

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

[✓ ☐ Yes] ☐No ☐Partly (Can be improved)

Comment on your analysis: 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

Comment on your findings: Classes and methods work as intended.

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

☐ Yes [✓ ☐No] ☐Partly (Can be increased)

Comment on your analysis: 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 (bad): In this case a class shares a common variable with another class, or relies on, or controls the execution of, another class.)

☐ Yes [✓ ☐No] ☐Partly (Can be reduced)

Comment on your analysis: Examined classes

Comment on your findings: All classes have low coupling and do not share common variables with 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: Examined Classes

Comment on your findings: All 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 Classes

Comment on your findings: All classes are generalized

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.

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 can be found easily.

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: MainWindow methods used to make line and scatter charts are both commented

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 can easily be updated or enhanced through minor modifications.

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved) ☐Don't know

Comment on your analysis: Examined Classes.

Comment on your findings: No nested loops found

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved)

Comment on your analysis: Examined Headers.

Comment on your findings: MainWindow inherits QMainWindow, TreeItem and RecordsManager do not 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, **<u>create several scenarios</u>** starting from the **<u>user's</u>** point of view: consider identifying one or more **<u>typical</u>** scenarios (e.g., those expected to be used with high frequency) and one or more **<u>low-frequency</u>** scenarios .

Each scenario is described as follows:

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

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

Title: Drawing a Line Chart / Scatter Plot
Anticipates frequency of use: normal
Starting Point: User has loaded data and is looking at dashboard summary view
Outputs: A Line chart is drawn in the dashboard summary view
Inputs:

- User clicks on tree view
- User clicks on the button for Line chart or Scatter plot → RecordsManager → QCustomPlot → MainWindow

Comment on your findings, with specific references to the design/code elements/file names/etc.: Checks are performed when a user clicks the button, the field string is analyzed by RecordsManager to get the data to be displayed, which is then passed to QCustomPlot and the appropriate method in MainWindow to update the visualizations.

(Note: expand here as necessary for each scenario)

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

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

  ☐ yes ☐ no ☐ partly, could be improved
- Two types of comments are required under each question: (i) your analysis, and (ii) your finding (in the form of a comment). The analysis would typically show how you arrived at the finding.
- Add new lines as necessary for your analysis or findings.

Scope and process:

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

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

Classes Inspected: ErrorEditDialog, RecordsManager

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 to the appropriate 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 (good): the functionalities embedded in a class, accessed through its methods, have much in common, e.g., access common data)

☐ Yes ☐No [✓ ☐Partly (Can be increased)]

Comment on your analysis: Examine methods within the classes

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. The entire database analysis is performed in one method, which although it is well-defined could be broken down into smaller, more compact and task-specific methods.

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

☐ Yes ☐No [✓ ☐Partly (Can be reduced)]

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

Comment on your findings: RecordsManager handles the bulk of the data processing and has quite a few functional dependencies. In particular, it is interdependent with TreeModel and requires 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 implementation 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 concerns 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 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 separate 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 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 behavior. With respect to RecordsManager, all methods requiring interaction with TreeModel are public to ensure accessibility. All other attributes and methods are private. This is the 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 the 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. ErrorEditDialog is very well encapsulated and is general enough to be reusable in any applications 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.

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

[✓ ☐ Yes] ☐No ☐Partly (Can be improved)

Comment on your analysis: Evaluate how easy it is to understand the implemented classes.

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

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

☐ Yes ☐No [✓ ☐Partly (Can be improved)]

Comment on your analysis: Determine the quantity and quality of comments within code clocks with a high degree of complexity

Comment on your findings: ErrorEditDialog's code complexity is quite low and thus has comments where necessary, though it could use more comments on the method level to help new users to understand the code. RecordsManager is a highly complex class and has more comments to help understand the code. Method parameters are well-defined and line-by-ine comments are provided when necessary.

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: Attempt to predict the amount of code enhancements required to update or enhance the code

Comment on your findings: 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 be directed more towards 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 the analyze() method has a single recursive call. Run-time efficiencies are at most proportional to the square of the amount of input data (for example, the number of records in the .csv file)

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved)

Comment on your analysis: Examine generalizations from the class diagram

Comment on your findings: ErrorEditDialog has a single parent class in Qdialog and has no child classes. RecordsManager has no children 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 from the class diagrams

Comment on your findings: Neither ErrorEditDialog or 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:

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

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

Title: Edit Records with Missing Data Fields / Navigation of Erroneous Entries
Anticipated frequency of use: high
Starting Point: User is prompted with message asking whether to edit or discard erroneous files.
Excepted Outputs:

- A correctly formatted dashboard summary view with modified records no longer containing missing data fields.
- Changes saved in "test.csv"

Inputs:

- Edit button clicked → passed to MainWindow
- Find Next button clicked (optional) → passed to ErrorEditDialog
- Empty fields filled in → passed to ErrorEditDialog
- Save button clicked → passed to ErrorEditDialog

Walkthrough:

- MainWindow returns the user's prompt, creates a new ErrorEditDialog and executes it.
- ErrorEditDialog creates and populates a table with the records containing missing fields. The cells which are colored red represent erroneous entries.
- When Save button is clicked, ErrorEditDialog will save the changes into a new .csv file.
- 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.: The ErrorEditDialog populates the table with records containing missing mandatory fields. The data is manipulated internally instead of interacting with the RecordsManager, this can be seen as a sign of both encapsulation and low coupling. It makes use of inheritance by calling the QDialog methods of exec() and accept() instead of implementing its own methods for running and successfully 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 erroneous records
Expected Output: A correctly formatted dashboard summary view without erroneous records

Inputs:

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

Walkthrough:

- MainWindow returns the user's prompt, creates a new ErrorEditDialog and executes it
- ErrorEditDialog creates and populates a table with erroneous records. The red colored cells represent the missing entries.
- When the Cancel button is clicked, ErrorEditDialog returns back to MainWindow
- 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 returns to the MainWindow as if the user pressed discard instead of edit. It makes use of inheritance by calling QDialog's exec() and reject() methods instead of implementing its own methods for running and returning.

END.

cs3307a – Object oriented analysis and design

Design Inspection Instrument

Instructions:

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

Scope and process:

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

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

Classes Inspected: MainWindow

Structural correspondence between Design and Code:

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

[✓ ☐ Yes] ☐No ☐Partly (Can be improved)

Comment on your analysis: MainWindow is 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.

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 (good): the functionalities embedded in a class, accessed through its methods, have much in common, e.g., access common data)

[✓ ☐ Yes] ☐No ☐Partly (Can be increased)

Comment on your analysis: All methods in MainWindow are specific to the MainWindow design.

Comment on your findings: The encapsulated methods can be reduced and simplified to improve performance and readability.

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

☐ Yes ☐No [✓ ☐Partly (Can be reduced)]

Comment on your analysis: MainWindow is highly inter-dependent with tree models.

Comment on your findings: The high coupling might make it difficult to simplify MainWindow

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: Inter-dependency between MainWindow and the tree models 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: MainWindow methods are private, except for the constructor.

Comment on your findings: These are the proper access specifications with regard to the interdependency between 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 methods and slots in MainWindow are specific to the gui layout.

Comment on your findings: 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: The methods in MainWindow have well defined functionality with regards to the gui functionality.

Comment on your findings: The methods for MainWindow can be simplified to help understand it better.

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved)

Comment on your analysis: MainWindow has comments for only some parts of the code while most of the old code needs more comments.

Comment on your findings: Perhaps there wasn't enough time to fully understand the old code and add comments to it.

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

Comment on your findings: MainWindow 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 was no delay when testing the ability to save and load sessions.

Comment on your findings: The methods used to save and load are very basic, and do not have any nested loops or delays.

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

☐ Yes [✓ ☐No] ☐Partly (Can be improved)

Comment on your analysis: MainWindow inherits several QT widgets.

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: MainWindow does not have children classes

Comment on your findings: There is no abstraction problem.

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

Each scenario is described as follows:

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

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

Title: Saving and Loading a session
Anticipated frequency of use: High
Starting point: About to quit session after using the software
Output: Able to load your session after you quit.
Input:

- User exits the program → MainWindow saves
- User opens the program → MainWindow loads the session

Comment on your findings, with specific references to the design/code elements/file names/etc.: When the user exits the program, MainWindow will save the session by saving any tabs that currently had data loaded. This is done through the serialize___Screen where ___ is the appropriate tab. Next time the user opens the program MainWindow will load the session by using the loadSerialize___Screen where ___ is the appropriate tab.

(Note: expand here as necessary for each scenario)

END.