DELIVERABLE 5: Report: Documentation of Stage 3 Project

Team Libra

December 7, 2016

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

Table of Contents

1 Minimum System Requirements

1.1 Dashboard Screens

Just as the Phase I project allowed the user to load a CSV file and display it on the main window, Phase II allows the user to load records for each subject area. Through the MainWindow, CSVReader, RecordsManager, and TreeModel classes. Phase II further improves on this design, allowing the user to not only save their session state for later use, but export their session state to an external file, which can later be opened from within the program (see 2. Stretch Goals). The information loaded in from the file is then displayed in column by row fashion. Team Libra's addition of the checkFields() method in the ErrorEditDialog class searches through the specified row and check for the validity of entries, helping to guarantee a intact file.

1.2 Visualization

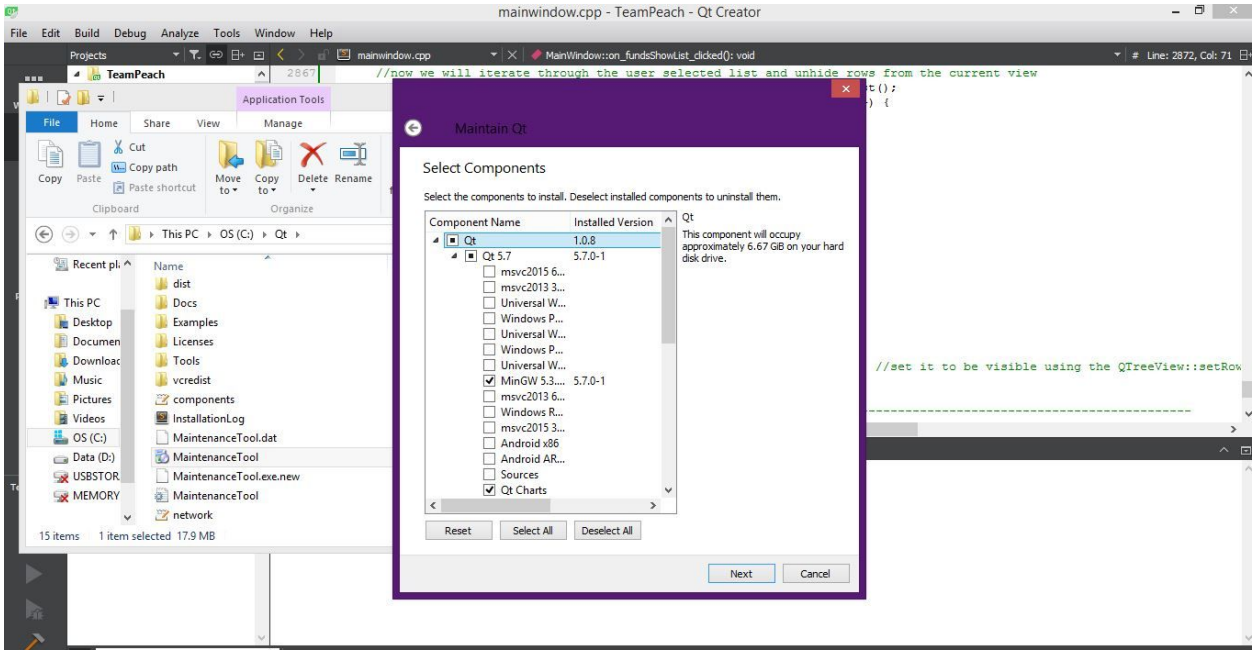
Phase II of the Peach Galaxy (Libra Galaxy) software includes all features of the previous iteration of the system, as well as two new graph types, the line graph, and the stacked bar chart. The Phase II project further adds the functionality to sort the list by member division and to create a user selected list. More information on the new implemented requirements are listed below

1.3 New Graph Types

Phase II of the system features a line graph representation of the CSV data, as well as a stacked bar chart representation. The added graph types allow for new visualisations of the data. To increase user experience, we've included animations when creating the stacked bar chart. Due to inconsistent screen resolutions, the graphs may not scale properly, but this is an easy fix as it would use the already existing scaling code for other graphs. Due to time limitations, the new graph type radio buttons have been accidentally left as disabled. Simply enabling them will fix the problem.

In the event of a module error when running from QT, add the "QT Charts" component via the QT Maintenance Tool. This should not apply to users who run via the executable created after install.

Below are the required components. Please ensure to install/update via Qt Maintenance tool



1.4 Ability to Save the Session State

Save session state capabilities have been added as part of the Phase II mandatory requirements. Whenever the user closes a tab, or the entire app, the session state is saved to a temp file. Whenever the program is reopened the user will be prompted to continue from their last session, which will load these temporary files. If they decline, a new session state will be started.

1.5 Sorting by Division

The ability to sort the data by the member's division has been added as an expansion to the Phase I project. The mainwindow and customSort classes have been modified. An additional sorting field has been implemented so that members can be sorted by division criteria. To do so, the division field has been added to the sorting fields in each tab.

1.6 User Selected List

The user can create a custom, user selected list to display the information passed to MainWindow in the manner of their choosing, from the options given. The user can click on a member, and the currently selected member can be added to the user selected list via “Add to List” button. The user can continue to do this, adding users to their own custom list, and may click “Show List” at any time to display the list. From this window, they have the options to remove a single index, remove the most recently added index, clear the entire list, or hide to list and return to normal view. Alternatively, the user can remove a user when in the normal view, as long as the currently selected member is in the list. The “Clear List”, “Remove from List” and “Undo Last Added” buttons become disabled if the user selected list is empty. Each list-modifying button prompt from the *mainwindow.ui* class sends the currently selected member as a parameter to the UserSelectedList class, while the list-displaying buttons just modify the current tree view by hiding rows not in the user selected list. The UserSelectedList class acts as a container for a QVector that stores QModelIndex objects.

Due to conflicts and time constraints, the user selected list does not currently display the data of all of the users in the custom list in the graph view; however, this is an easy fix and the pseudocode is provided below:

//for each “Show List” button in each subject area tab
Collect all the indexes (users) in the list as a vector of strings
Call the ui’s graph display methods and pass the above vector
//note that this code is very similar to the already existing graph displaying code.
//Just a few parameters and variables need to be changed

Along with this pseudo code, commented out progress code was also included in the program’s code.

1.7 Navigation of Erroneous Entries

Upon loading a .csv file and being prompted to edit entries, the user may enter a secondary window and add or remove entries. This feature, with the navigation of erroneous entries implement, is improved upon, for now the user may scan through the list's highlighted entries quickly using the new buttons provided. This allows for quicker navigation and editing/discarding of file entries.

1.8 Verify the Code Deck for Operation

The application works for Windows 10, as well as Mac OSX.
When running the code deck on Mac, the splash screen feature causes the program to halt while displaying the screen. Despite our best efforts, and our investigation into the problem through documentation and the internet, there does not seem to be a resolution for the issue. Simply comment out the code in main.cpp if you intend to run the software on a Mac.

Also note that Libra Galaxy is intended to work with the new sample data files.

1.9 Fix all errors so that the finished product is error free

Our team fixed most of the errors from team Peach's code. There are a few new bugs in our code, but this can be easily fixed.

1.10 Easy Installation and Use

Our team included several resources to make installation easy for the user. We've included an installation guide, and easy install wizard, a user guide, and a video(see stretch goals) in addition to the exe file.

2 Stretch Goals

2.1 Save Session Export

In addition to the ability to save the session state, the user can at any time choose to export their current temp files to a permanent file, which can be loaded at a later date. This is important because, whenever a user exits a session and decides not to continue, the previous temp files are wiped. This way the user is capable of saving their states for later dates, across multiple sessions.

2.2 Change/Modify Sort Order

The user can click on the “Member Name” header at the top of the list of members. By doing so, the user modifies the sort order to toggle between alphabetic ascending or descending.

2.3 Ability to Close Select Subject Tabs

Another feature added is the ability to close subject tabs. Whenever a user closes a subject tab, they are prompted to save the session for that subject tab. (See 1.3.2)

2.4 Bug Fixes

Additional improvements have been made to the existing MainWindow class. The colour selection of the pie chart graph is no longer randomized and, as a result, will not generate a graph of a single solid colour for all sections. Tooltips and errors relating to them have also been fixed, as the Phase I program would display an “Export to PDF” tooltip over all interactable areas of the user interface.

2.5 User Can Modify Erroneous Data

The previous function of editing erroneous entries has been improved upon, as it was only called once. Now the user, via a button, can choose at any time to edit the fields in the CSV file. Although it was not completed the code can be found below:

```
/**
* Method to open error edit dialog window
*/
void MainWindow::open_errorDialog(int tabIndex)
{
  switch (tabIndex) {
    case FUNDING:
      if (f_errs_fund.size() > 0) {
        if(handle_field_errors(f_errs_fund, f_errs_fund_fixed, header, FUND_MANFIELDS)) {
          for (unsigned int i = 0; i < f_errs_fund_fixed.size(); i++) {
            funddb->addRecord(reader.parseDateString((*(f_errs_fund_fixed[i]))[sortHeaderIndex]),
f_errs_fund_fixed[i]);
          }
        }
      }
      refresh(FUNDING);
      break;

    case PRESENTATIONS:
      if (f_errs_pres.size() > 0) {
        if(handle_field_errors(f_errs_pres, f_errs_pres_fixed, header, PRES_MANFIELDS)) {
          for (unsigned int i = 0; i < f_errs_pres_fixed.size(); i++) {
            presdb->addRecord(reader.parseDateString((*(f_errs_pres_fixed[i]))[sortHeaderIndex]),
f_errs_pres_fixed[i]);
          }
        }
      }
      refresh(PRESENTATIONS);
      break;
    case PUBLICATIONS:
      if (f_errs_pub.size() > 0) {
        if(handle_field_errors(f_errs_pub, f_errs_pub_fixed, header, PUBS_MANFIELDS)) {
          for (unsigned int i = 0; i < f_errs_pub_fixed.size(); i++) {
            pubdb->addRecord(reader.parseDateString((*(f_errs_pub_fixed[i]))[sortHeaderIndex]),
f_errs_pub_fixed[i]);
          }
        }
      }
      refresh(PUBLICATIONS);
```

```
        break;

      case TEACH:
        if (f_errs_teach.size() > 0) {
          if(handle_field_errors(f_errs_teach, f_errs_teach_fixed, header, TEACH_MANFIELDS)) {
            for (unsigned int i = 0; i < f_errs_teach_fixed.size(); i++) {
              teachdb->addRecord(reader.parseDateString((*(f_errs_teach_fixed[i]))[sortHeaderIndex]),
f_errs_teach_fixed[i]);
            }
          }
        }
        refresh(TEACH);
        break;
  }
}
```

2.6 Enhanced User Experience

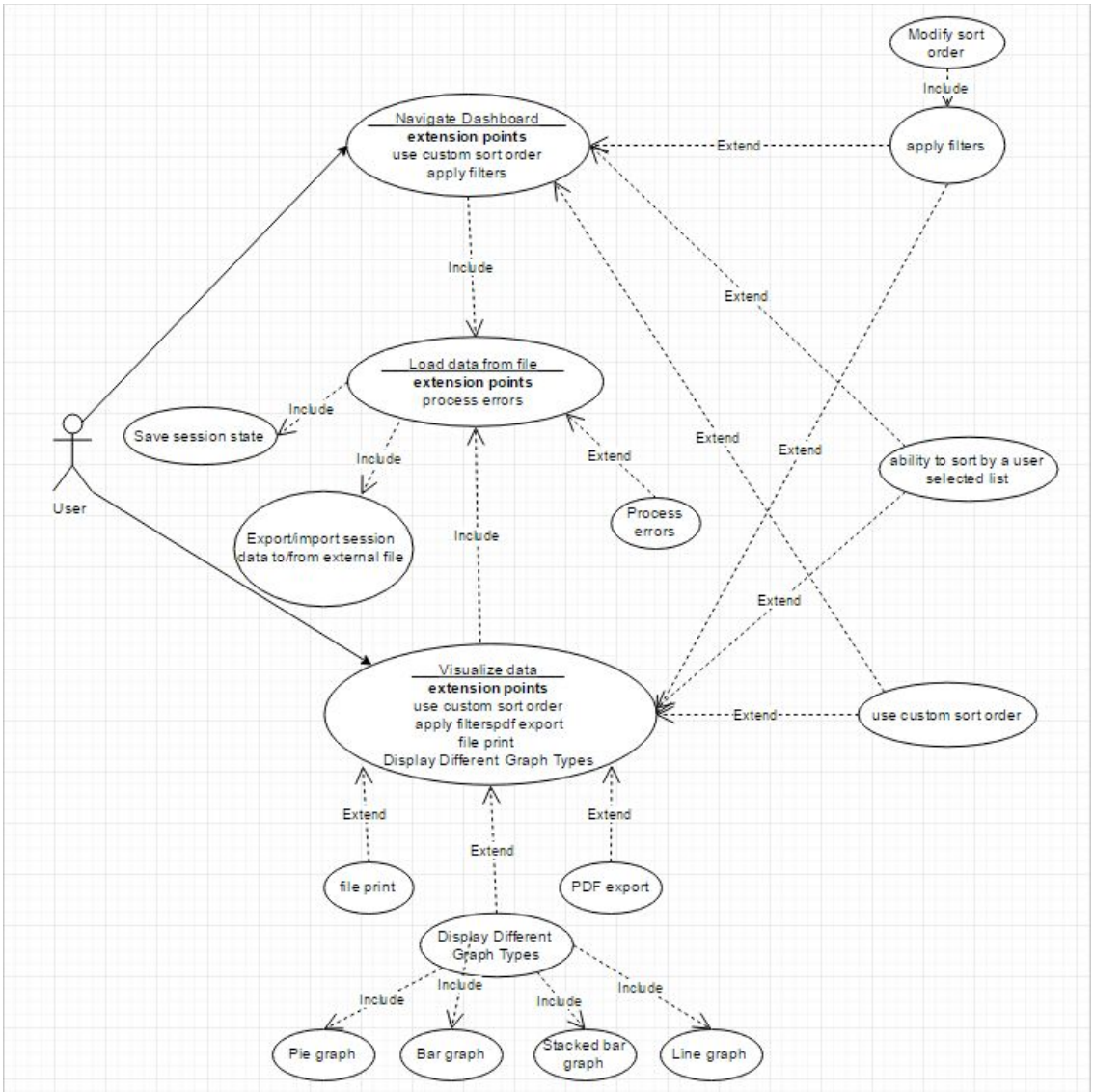
Our team implemented several changes to the UI to enhance user experience. We've implemented a splash screen on startup and a button to toggle day/night mode. The Day/Night button toggles between visual themes to make it easier for the user's eyes.

2.7 Video

We've included a video to assist and familiarize the user with the functionality and features of the program.

3 System Design

3.1 Use Case Diagram



The use case diagram, as seen above, outlines the typical user's interaction with the system. Each user, whether they be faculty or a manager, has two primary use cases. Firstly, the "Navigate Dashboard" handles all user interface requirements, and the dashboard screen requirement. Secondly, the "Visualize Data" use case addresses all visualization requirements for the system, such as the many graph types and the Phase I PDF export. Both use cases include the "Load Data from File" use case, which addresses the file data processing requirement, as well as the mandatory save session state requirement, and the related stretch goal of importing/exporting session data to an external file. The specifications for each use case are detailed below:

3.1.1 Loading Data from File

1. The user selects a subject area tab.
2. The user clicks the Load button.

3. The user selects a CSV file and click the Open button [Branch to Alternate Case A if file is not valid. Branch to Alternate Case B is user prematurely cancels]
4. System verifies integrity of file, and checks for missing fields.
5. System loads records, and information is displayed for user.

Alternate Case A: File is not CSV, or file is loaded in incorrect tab

1. Display error message.
2. User closes dialogue box, and file does not load.

Alternate Case B: User prematurely cancels file loading

1. System closes file structure screen.

3.1.2 Process Errors

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

Alternate Case A: User click Discard instead of Edit

1. System discards records with missing entries and closes error processing screen.

Alternate Case B: All entries not filled out

1. System displays error message.
2. User closes error dialogue box.

Alternate Case C: User clicks Cancel

1. System discards records with missing entries and closes error processing screen.

3.1.3 Save Session State and Export/Import Session

1. System prompts user to continue from last session on start up.
2. User declines. [Alternate Case A: User accepts]

3. New system session is created.
4. User closes software dashboard view or a given tab.
5. System prompts user to save session state
6. User accepts. [Alternate Case B: User declines]
7. Data is saved for next run of the program.
8. User is prompted to export save file.
9. User accepts. [Alternate Case C: User declines]
10. Save file can be exported to local hard drive and loaded at a later date.

Alternate Case A: User Continues Previous Session

1. System dashboard parses previous session data and displays information.

Alternate Case B: User declines saving session state

1. System clears dashboard data and does not save.

Alternate Case C: User declines file export

1. No file is exported, and the system closes.

3.1.4 Navigate Dashboard

1. User loads in CSV file via use case 3.1.1.
2. System displays the updated dashboard summary view.
3. User expands/collapses elements of the summary view.
4. System updates with expanded/collapsed elements.

3.1.5 Applying Filters (Extended from 3.1.4)

1. User modifies the values in the start and end date boxes.
2. System sets date range according to user input.
3. User modifies the values for last name starting letters.
4. System sets member name range according to user input.

3.1.6 Using Custom Sort Order (Extended from 3.1.4)

1. User clicks “Create New Sort Order” [Alternate Case A: User selects existing sort order]

2. System displays new sort order screen.
3. User names and selects sorting pattern for the list. [Alternate Case B: User does not enter name][Alternate Case C: User click Cancel]
4. System closes the new sort order screen.
5. The system adds the new sort order to the list of existing sort orders.
6. User selects sort order.
7. System sorts file information based on user custom sort order.

Alternate Case A: User selects existing sort order

1. [Branch to 6]

Alternate Case B: User fails to enter name

1. System prompts user with error message until list is named
2. [Branch to 3]

Alternate Case C: User clicks Cancel button prematurely

1. System closes new sort order screen

3.1.7 Sorting by a User Selected List (Extended from 3.1.4)

1. When on current tab, user clicks on a member.
2. User clicks “Add Index” button to add selected entry to list.
3. User chooses to display list via “Show List” button.
4. User closes list with “Hide List” button. [Alternate Case A: User removes an entry from the list][Alternate Case B: User clears entire list]
5. System returns to dashboard, whether to continue adding entries to selected list, or continue software use otherwise.

Alternate Case A: User removes an entry with “Remove Index” button

1. Entry is removed from user selected list.
2. [Branch to 4]

Alternate Case B: User clears entire list with “Clear List” Button

1. System clears entire user selected list.
2. [Branch to 4]

3.1.8 Visualizing Data

1. The user loads the data from file.
2. The system displays the dashboard summary view.

3. The user clicks on an element in the dashboard summary view.
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.

3.1.9 Exporting to PDF

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 Case A: No file name entered] [Alternate Case 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 Case A: No file name entered
1. The system displays an error message.
2. The user accepts or closes the error message. [Branch to 3]

Alternate Case B: User clicks Cancel button
1. The system closes the file structure screen.

3.1.10 Printing to File

1. The user clicks the Print button.
2. The system displays a file structure screen.
3. The user selects a file path (default is current working directory), enters a file name (default is “print”) and clicks the Print button. [Alternate Case A: No file name entered] [Alternate Case 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.

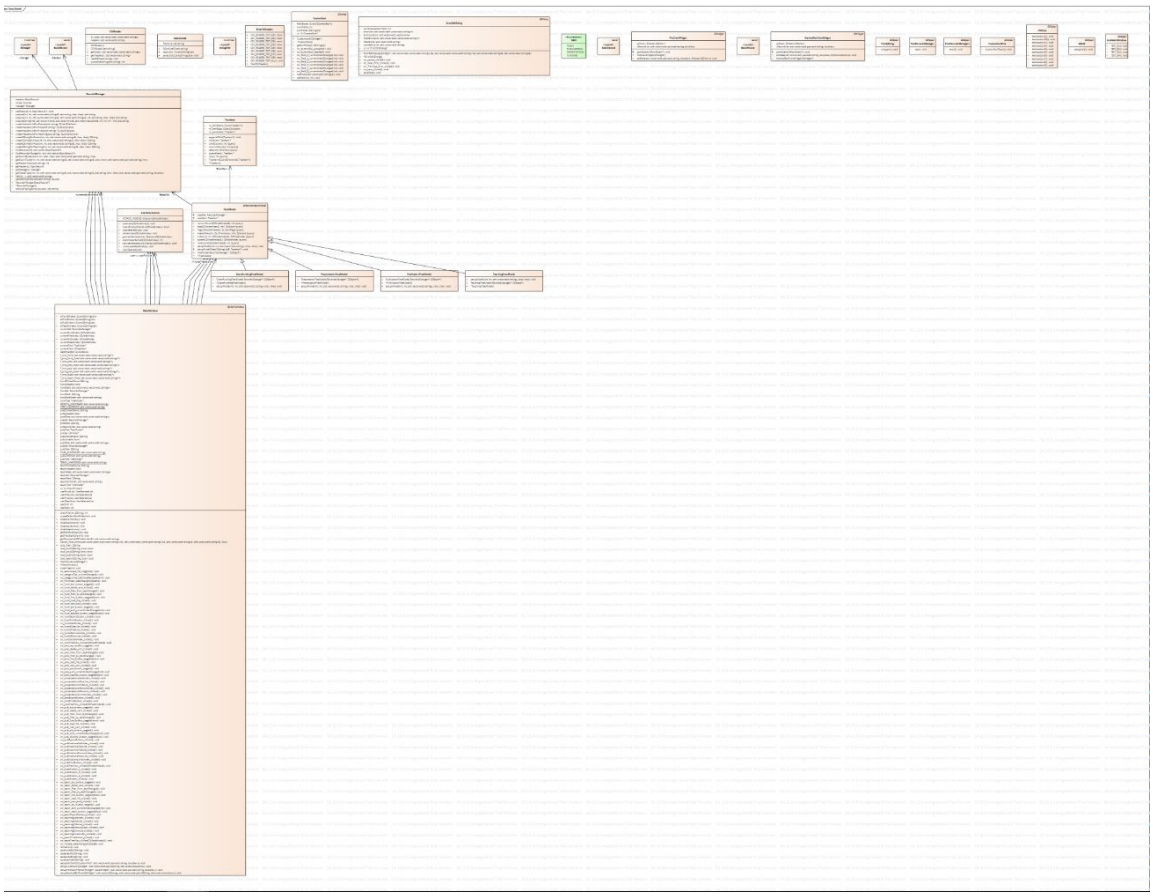
Alternate Course A: No file name entered
1. The system displays an error message.
2. The user accepts or closes the error message. [Branch to 3]

Alternate Course B: User clicks Cancel button
1. The system closes the file structure screen.

3.2 Class Diagram

Please see the attached file, "UML Libra.bmp," a for a larger version of the class diagram.

Below is the Galaxy Libra 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.

- **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.
- **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.
- **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.
- **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.
- **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.
- **PieChartWidget:** PieChartWidget is created during the execution of MainWindow; it is its own temporary standalone 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.
- **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 subclass of QDialog so as to facilitate integration with the other Q-type graphical user interface components.
- **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.
- **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.
- **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.
- **StackedBarChartWidget:** StackedBarChartWidget is created during the execution of MainWindow; it is its own temporary standalone class for visualizing the data kept in the records inside the TreeModel. StackedBarChartWidget is a subclass of QWidget so that it may take full advantage of the QT library features and functionality.

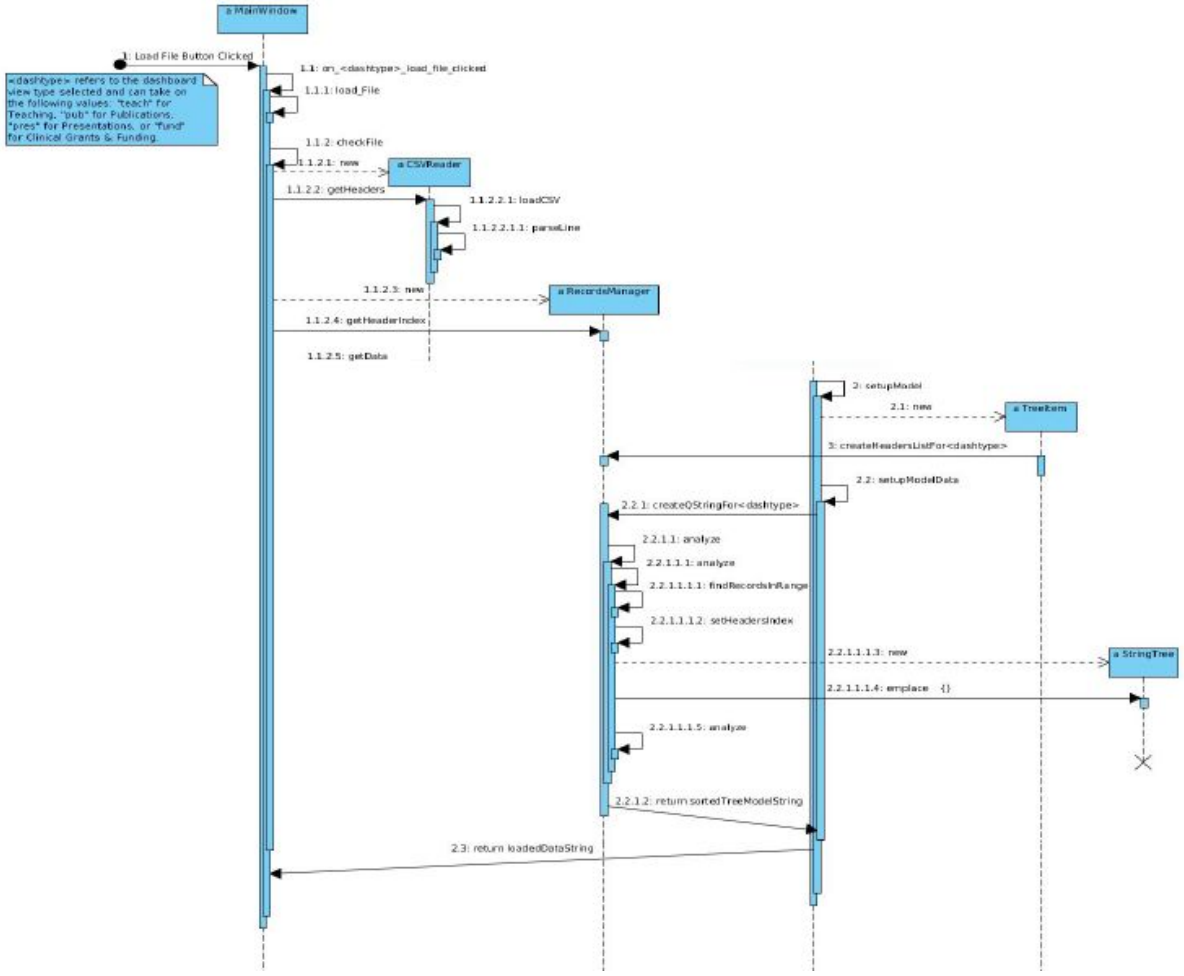
- **UserSelectedList:** this class is a custom made class that is created during the execution of MainWindow. Four instances are created for their respective tabs. UserSelectedList is a class that has a QVector to store QModelIndexes

3.3 Sequence Diagrams

Below are sequence diagrams for the following use case scenarios:
“Load Data from File”
“Load Data from File and Create a User Selected List”
“Error Edit Dialogue”
“Save Session State After Closing Tab”
“Save Session State After Editing Data”

3.31 “Load Data from File”

Below is the sequence diagram for the use case scenario “Load Data from File”, including the main scenario and extensions. An in depth description of the diagram is similar to the next case:“Load Data from File and Create a User Selected List”

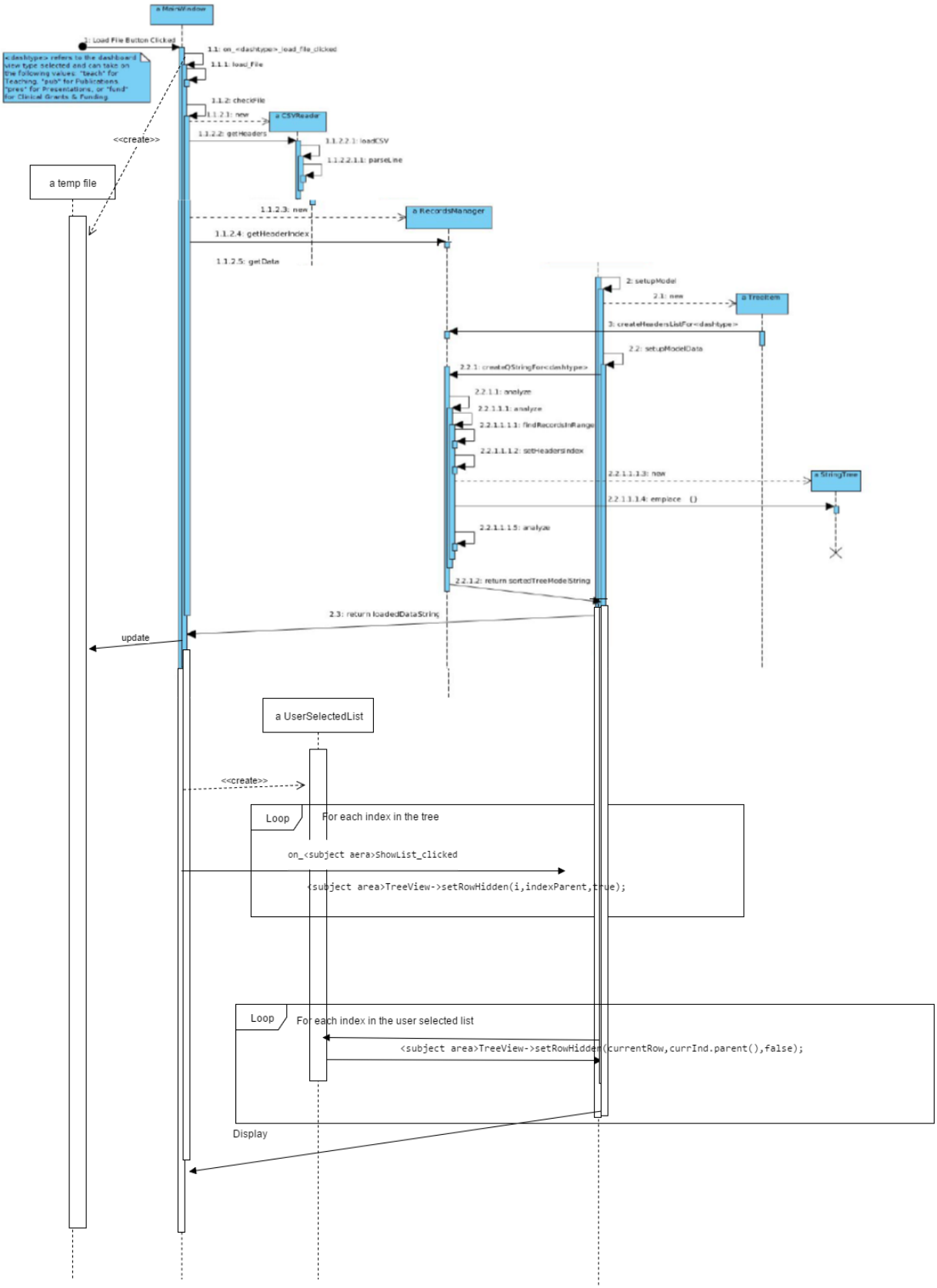


3.32 “Load Data from File and Create a User Selected List”

Load Data from File and Create a User Selected List including the main scenario, extensions and including creating a temp file for saving the session state (this case only includes showing the user selected list as it assumes a list already exists). Although the notation of the sequence diagram itself is quite self-explanatory, we’ve included a text explanation:

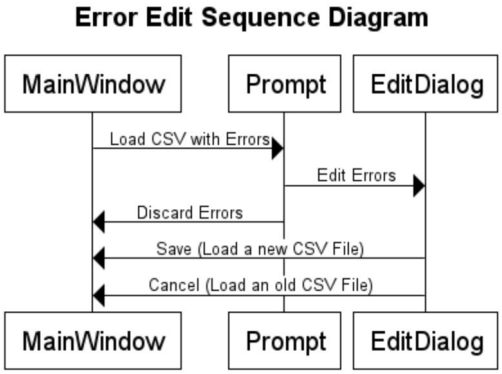
- **User clicks load button:** This step is represented by the found message, an arrow with a dotted end in the top left of the diagram entitled “Load Data Button Click”. It activates the MainWindow self-call “on_<dashtype>_load_file_clicked”, catalyzing the entire sequence.
- **System displays file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

- **System Shows a User Selected list:** Assuming a user selected list already exists, the user clicks the “Show List” button and the current view would display the use selected list. The system achieves this by first hiding all the rows, and then iterating through the user selected list and showing these rows.
- **Save Session State:** Upon startup of the application, the user has the option to load a previous session state (excluded from the diagram as our case we are only dealing with saving). After a file is loaded the session is saved to a temp file, which saves the current session state, as well as the option to save any new modified data files (refer to stretch goal).


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
<<create>>
1.1.2.2: getHeaders
1.1.2.2.1: loadCSV
1.1.2.2.1.1: parseLine
a temp file
1.1.2.3: new
a RecordsManager
1.1.2.4: getHeaderIndex
1.1.2.5: getData
2: setupModel
2.1: new
a TreeItem
3: createHeadersListFor<dashtype>
2.2: setupModelData
2.2.1: createOStringFor<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_back()
2.2.1.1.1.5: analyze
2.2.1.2: return sortedTreeModelString
2.3: return loadedDataString
update
a UserSelectedList
<<create>>
Loop
For each index in the tree
on_<subject aera>ShowList_clicked
<subject area>TreeView->setRowHidden(i,indexParent,true);
Loop
For each index in the user selected list
<subject area>TreeView->setRowHidden(currentRow,currInd.parent(),false);
Display

3.33 “Error Edit Dialogue”.

Error Edit Dialogue prompts the user if they are missing mandatory fields in the .CSV file they uploaded, and allows the user the option to fill these fields in.

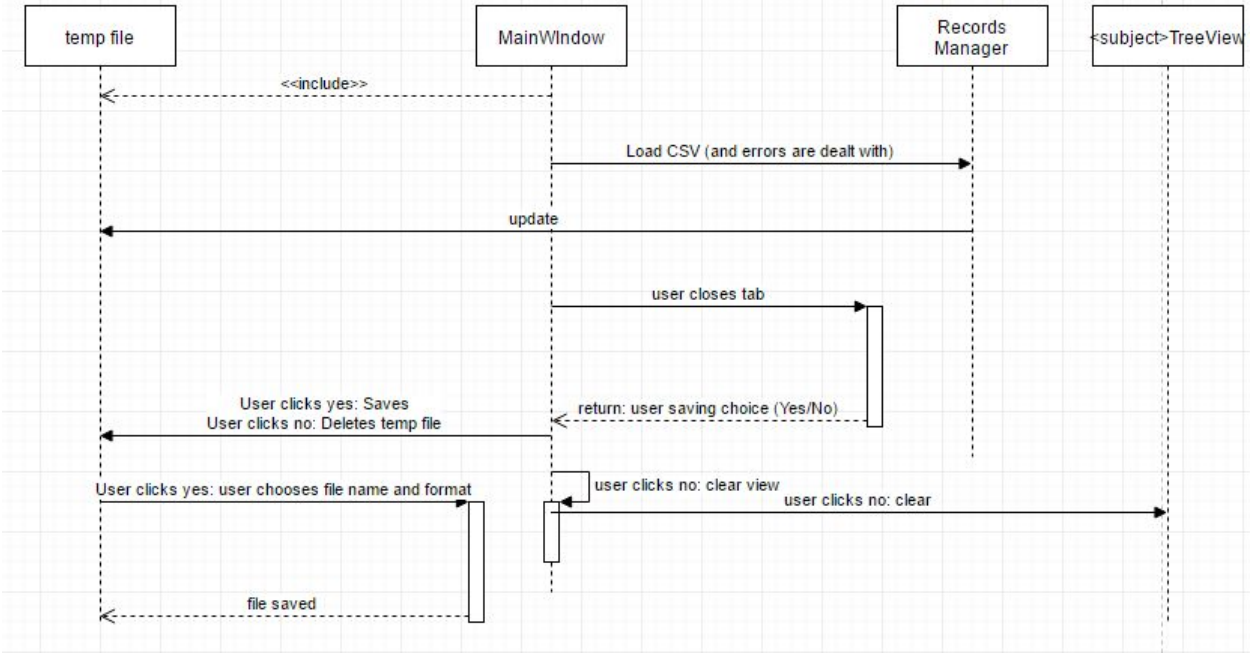


3.34 “Save Session State After Closing Tab”

Assumes data has been loaded.Upon startup of the application, the user has the option to load a previous session state (excluded from the diagram as our case we are only dealing with saving). The contents of its record manager is auto saved to a file. Upon closing of the tab user is prompt whether they want to save the content:

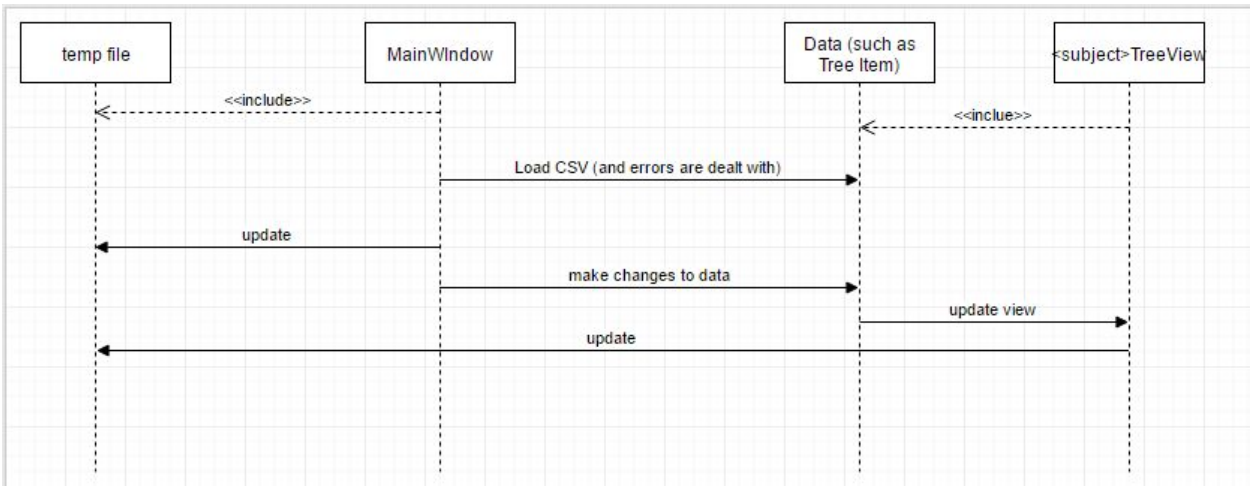
- **User clicks Yes:** the data in the temp file is saved to a file of the user’s choice
- **User clicks No:** the temp file is deleted

When a tab is closed: everything is set to blank. Note: some details and processes have been excluded from view.

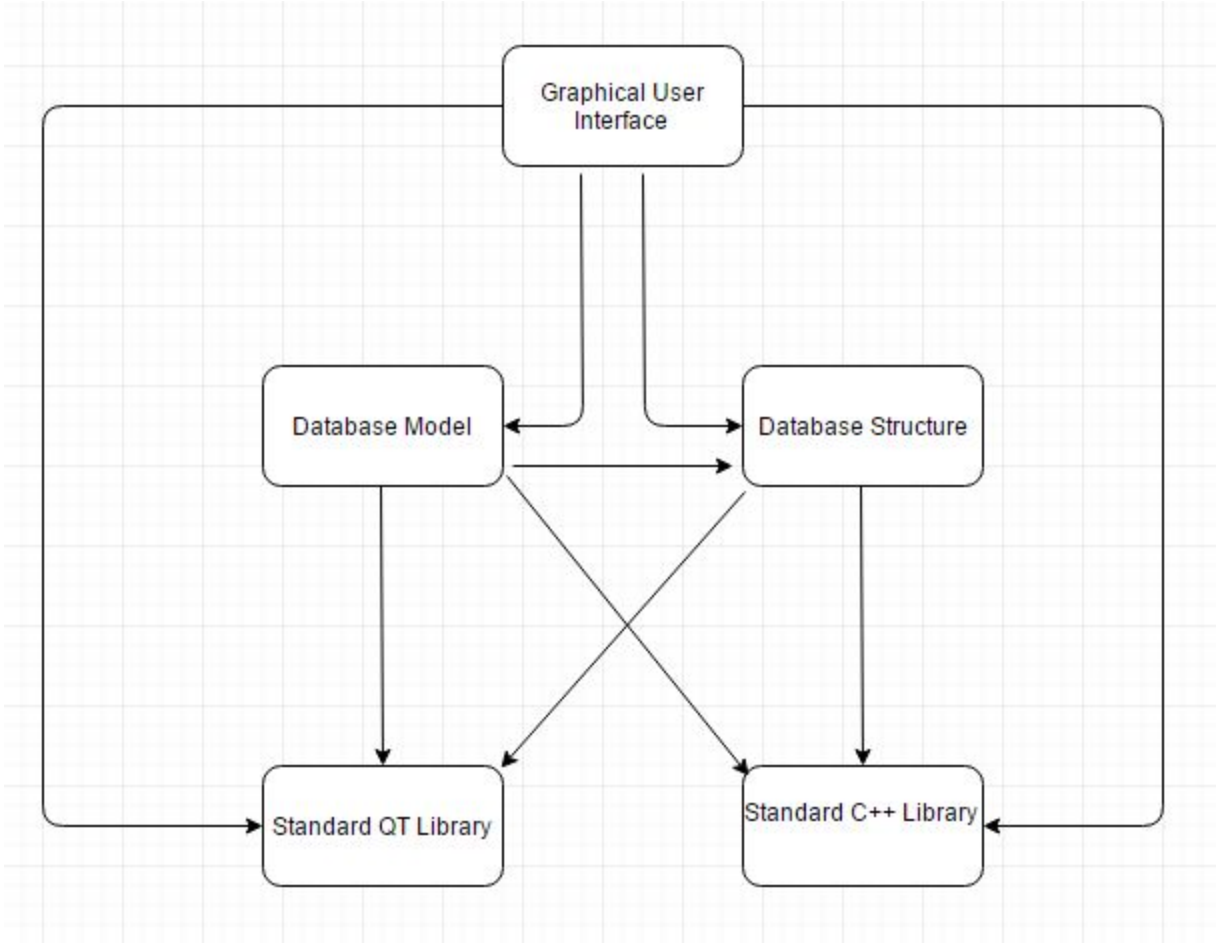


3.35 “Save Session State After Editing Data”

Assumes data has been loaded.Upon startup of the application, the user has the option to load a previous session state (excluded from the diagram as our case we are only dealing with saving). After a file is loaded the session is saved to a temp file, which saves the current session state, as well as the option to save any new modified data files. This scenario deals with saving the current session after modifying data and changing the tree view



3.4 Package Diagram



Shown above is the package diagram for the Phase II system. The only thing to note about the package diagram is the dependencies between packages. Each arrow begins at a source package and travels to the target package, conveying which source package is dependent on which target package.

3.4.1 Graphical User Interface

The GUI package contains all classes relevant to visual components, such as the dashboard display, graphs, dialogue boxes, widgets, and buttons to name a few. Because of their heavy use during user interaction the GUI package relies on the standard QT and C++ libraries in order to successfully function. Furthermore, MainWindow needs to create a new database and data structure for each CSV file loaded, making it dependent on the related packages.

3.4.2 Database Model

The database model package contains all classes relevant to the loading of files, and creation and use of the database which stores the data loaded in from CSVReader. Due to the simplicity of said class, it only really relies on the standard C++ library. However, RecordsManager is more complex in its implementation, and as a result it is reliant on both QT and C++ libraries, as well as the data structure model.

3.4.3 Data Structure Model

The data structure model package contains the TreeModel class and its related classes. TreeModel makes use of RecordsManager and, as a result, the package is dependent on the database model package. Naturally the package is also reliant on the standard libraries.

3.4.4 Standard Qt and C++ Libraries

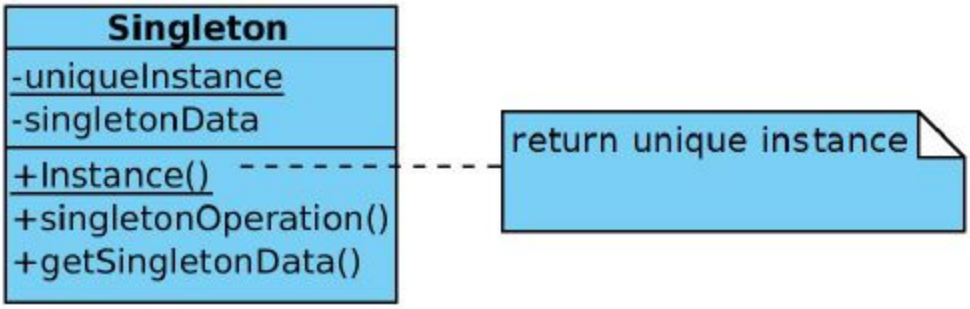
The documentation for both of these standard libraries is already present in the Qt IDE. They are used by every package in the system, and rely on none of them for functionality.

4 Design Patterns

The Libra 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.



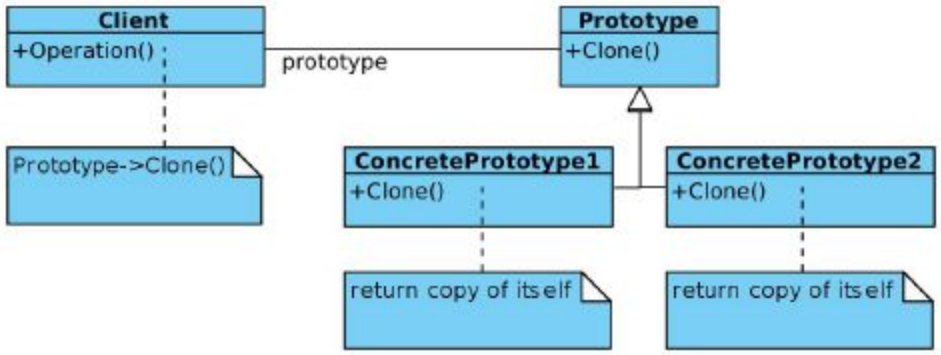
i. *Controlled access to sole instance:* because the Singleton class encapsulates its sole instance, it
can have strict control over how and when clients access it.
i. *Reduced name space:* the Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables 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 Implementation in C++

The code satisfies the design. As described above, the MainWindow class is an example of a singleton. Also described above, the TreeModel and TreeItem classes work in conjunction with the data, satisfying the prototype design. When making additions to the original phase 1 design, we have made sure to adhere to the design of the program, following the singleton pattern and creating the majority of code changes within MainWindow.

The functions written by Team Libra in our implementation of both mandatory and stretch requirements can all be found within the related MainWindow classes. Take, for example, the User Selected List. The primary functions that were implemented are located in MainWindow.cpp, and the corresponding changes to the user interface are kept in the MainWindow.ui file.

When handling the addition of the new graphs, it was decided to keep the stacked bar chart functions in their own respective classes, but otherwise, they are called and used by the MainWindow classes, maintaining the singleton design.

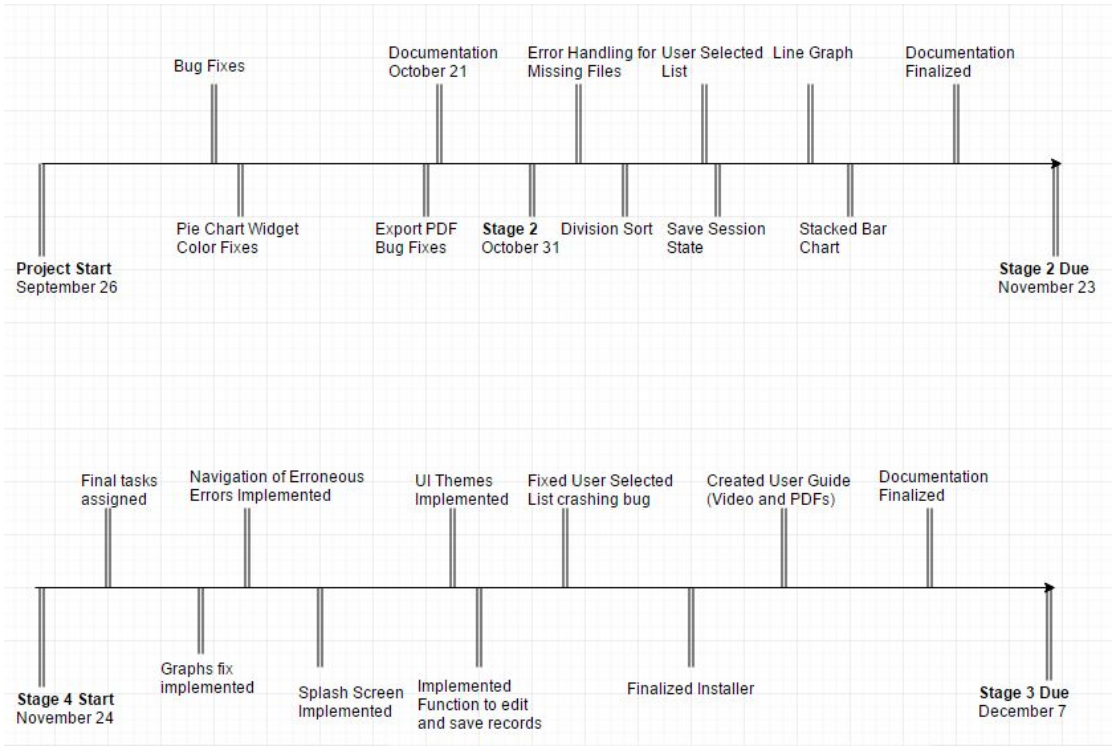
The state saving functions are, much like the other functions, in the Mainwindow classes, and are integrated into each tab in the project. Additionally, (as outlined in our stretch goals) the ability to close select tabs and save them is a part of these classes.

Overall, the majority of functional changes included with Team Libra's evolution of the Peachy Galaxy software are kept to the MainWindow classes, maintaining the singleton design.

6 Development Plans

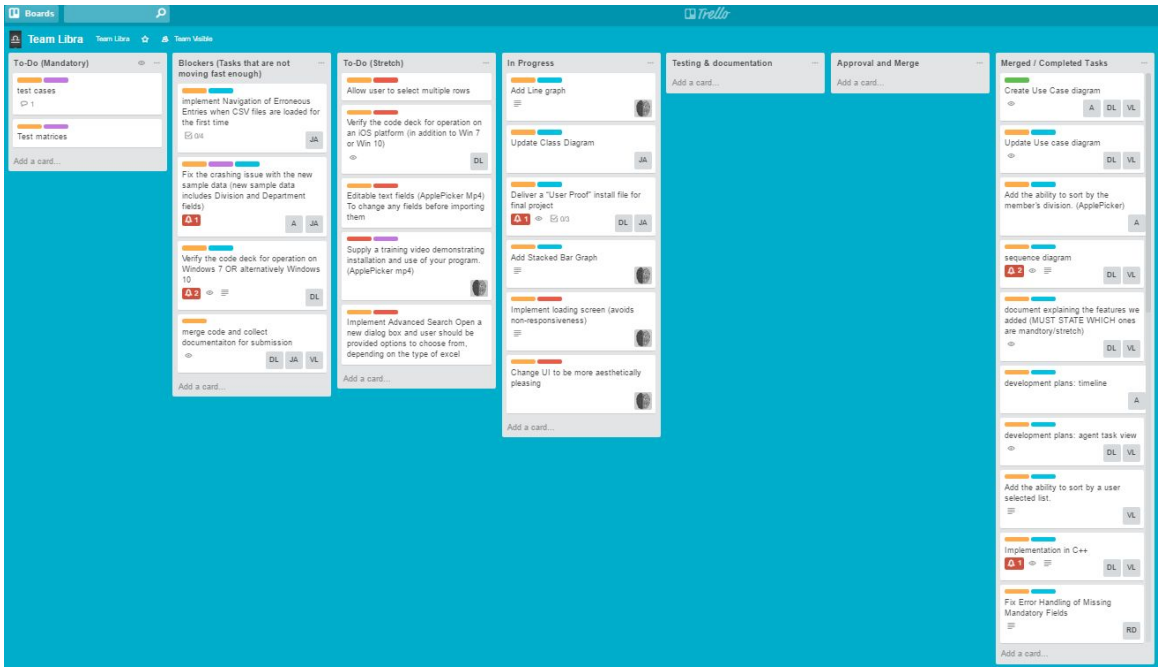
The Agent Task View is included separately in its own spread sheet.

6.1 Timeline



The Agent Task View files contain specific information on dates

Team Libra has used Trello as our live task tracking tool



Lessons learnt and retrospective analysis

One technical issue we faced was understanding how specific Qt components and functionalities worked. Many of us have never used Qt before, so many of its features were brand new to us. It would have been to the benefit of the group, and our projects expediency, to take a closer look at the documentation for the IDE and become familiar with it. Similarly, understanding the old code (each class and method) took awhile. A lot of the old code was uncommented and undocumented, so it was not always easy to understand the functionality of some of the function calls.

Implementing new features often took longer than we expected. We failed to take into account our other classes and real world commitments which meant we were often behind on our schedule. The main issue came from the goals we set, mainly their unrealistic expectations. Handling this class was like handling a part time job, on top of the other assignments and exams we all had to manage.

We had some difficulties combining all the Qt code written by different members using Github without obtaining conflicts. The main lesson to take from merging in cases like this must be that the ui files really enjoy starting conflicts. Ideally, a branch should be opened to place the added ui elements in, which should later have their slots filled by code from other branches.

The main issue in terms of the human interaction is one of communication. It was noticeable soon after our meetings that individuals were not communicating with the team. Ultimately, this lead to our second major problem as a group: the lack of balance in the work load. While these things are neatly documented in the agent task view, contextually, it's important to note that the assignment of tasks, and the actual completion of those tasks (and the members who completed them) are rarely congruent. Much of the work load had to be redistributed after the fact as a result of poor communication from the team. Ideally, in future team efforts, individuals will focus on responding, even if the response is to state inability to work, and prioritize communication. It is very hard to work together when members are not synchronized!

Through successes and tribulations, the experience has taught us a lot of valuable lessons.

- Work on a project that is planned to be used and evaluated by a customer and not only for the class teachers.
- Learn how to use a new programming software with its own components
- Use different tools to split and share our work within the other members
- Work with a team, for better or for worse, and participate in a simulation of a real world working environment.

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)

o yes o no o 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.

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

CSVReader and 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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Went through the code and headers for CSVReader and RecordsManager and compared with the class diagram.

Comment on your findings: The class diagram and the code are in sync.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: The CSV Reader has the functionality of Loading the CSV files and returning a vector of all the headers, the data and the dates. The RecordsManager is responsible of building the database of all the data send that to various tree models which show the view in application.
Comment on your findings: The CSVReader and the RecordsManager perform the intended functions.

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Examined the functions within the classes.
Comment on your findings: All the methods in CSVReader and RecordsManager are specific to their classes and are well encapsulated. Hence there is good-cohesion.

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

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Examined the implementation of functionalities with in CSVReader and RecordsManager. CSVReader doesn't have any dependencies on others. RecordsManager is dependent on CSVReader and TreeModel classes.
Comment on your findings: CSVReader is very less coupled. While the RecordsManager having a lot of functionality is tightly coupled.

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: The whole recquirements are broken into small components and each class is used to implement a component.
Comment on your findings: The CSVReader and the RecordsManager are used to implement Loading CSV Files and Loading data into database respectively.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files of classes for the access specifiers of data and methods.
Comment on your findings: The appropriate functions in RecordsManager are made public and others are kept private. The classes contain proper access specifiers.

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes
oDon't know

Comment on your analysis: Examined the code and functionality of the classes.
Comment on your findings: The CSVReader can be re-used in other applications but the RecordsManager can hardly be used because of its close interaction with TreeModel.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examining the code to evaluate its ease of understanding and ease of identifying the functionality. Evaluating the naming conventions used for classes and methods.
Comment on your findings: The CSVReader and the RecordsManager has functionalities which are easy to understand.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Scanned the code for these classes
Comment on your findings: Some parts do have proper comments but there are many sections where the comments can be improved for better maintainability and 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: The methods in CSVReader and RecordsManager are generic and can be updated
Comment on your findings: The CSVReader and RecordsManager can be easily updraged and enhanced without doing too many changes. (RecordsManager will require some changes.)

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

o Yes oNo oPartly (Can be improved) oDon't know

Comment on your analysis: There is a big delay in loading data into the database. The CSVReader doesn't add much to the inefficiency but the RecordsManager has many nested loops and many functions which add significantly to the loading time.
Comment on your findings: These classes can be improved to reduce the nested loops and the large number of methods can be reduced for increasing performance

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files for both the classes
Comment on your findings: The CSVReader and the RecordsManager don't inherit anything as such.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the headers for other classes
Comment on your findings: The CSVReader and RecordsManager doesn't have children. 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:
i) Title of scenario
ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be "touched" by the scenario and producing the anticipated outputs.

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

1. Loading data
2. Anticipated frequency of use : High
3. End-user trigger : User clicks the load file button and selects a file
4. Expected type of outputs: The data is loaded into the database
5. Main window calls-
   -- CSVReader to parse the file
   -- RecordsManager to load and manage the data from the file.

Comment on your findings, with specific references to the design/code elements/file names/etc.: CSVReader does the job of parsing through the data file properly as it has a small scope and limited functionality. RecordsManager works as per specification, but it has many methods and they interact with other classes which shows that the encapsulation and abstraction can be improved upon in the code. Both the classes make proper use of other Libraries for successful implementation of specifications.

(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)

o yes o no o 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

TreeModel and 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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Went through the code and headers for TreeModel and mainwindow and compared with the class diagram.

Comment on your findings: The class diagram and the code are in sync.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: TreeModel produces a valid TreeModel object accepted by the TreeModelView . The mainwindow is used to implement everything and this makes it overly complex and so thee are some problems in it while performing intending operations. Comment on your findings: The TreeModel implements the intended functionality but the mainwindow needs improvement.

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Examined the functions and code within the classes. Comment on your findings: All the methods in TreeModel are specific and are well encapsulated. Hence there is good-cohesion. The mainwindow tried to do it all and the functionality is at places not well defined and shows poor cohesion.

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

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Examined the implementation of functionalities with in TreeModel and TreeItems. TreeModel has some interactions with TreeItems class but the coupling is low. The mainwindow has complete dependence on all other classes.
Comment on your findings: TreeModel less coupled. While the mainwindow is tightly coupled.

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: The whole recquirements are broken into small components and each class is used to implement a component.
Comment on your findings: The TreeModel class is used to make treemodels for the dashboard to view. The mainwindow uses other classes to implement the dashboard view for the user.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files of classes for the access specifiers of data and methods.
Comment on your findings: The appropriate functions in TreeModel which deal with TreeView are made public and others are kept private. The classes contain proper access specifiers.

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Examined the code and functionality of the classes.
Comment on your findings: The TreeModel may be used in other situations, but as it is passed the RecordsManager currently, it cannot be reused a lot. The Mainwindow most certainly cannot be used in other applications.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examining the code to evaluate its ease of understanding and ease of identifying the functionality. Evaluating the naming conventions used for classes and methods.
Comment on your findings: The TreeModel class is not very easy to understand because of the interdependence. The excessive functionality implemented within the mainwindow makes it very difficult to understand and results in poor readability

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Scanned the code for these classes
Comment on your findings: Some parts do have proper comments but there are many sections where the comments can be improved for better maintainability and 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: The methods in TreeModel are generic and can be updated. But the code in mainwindow is very difficult to maintain and upgrade.
Comment on your findings: The TreeModel and Mainwindow are difficult to maintain and upgrade.

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Examine the complexity for the various methods
Comment on your findings: TreeModel has nested loops. Mainwindow has a lot of condition statements, switch statements and loops which at some places introduce inefficiency.

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files for both the classes
Comment on your findings: The mainwindow has no inheritance. TreeModel has 1 level deep inheritance.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the headers for other classes
Comment on your findings: The TreeModel and Mainwindow doesn't have children. 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:
i) Title of scenario
ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be "touched" by the scenario and producing the anticipated outputs.

Follow the code (structured walkthrough) to ascertain whether this scenario is properly implemented both in terms of logic and design.
1) Load multiple CSV file
2) Anticipated frequency of use : Low
3) End-user trigger : Click load file button
4) Expected type of outputs: Correct dashboard tree view
5) MainWindow
--File selected by user
--CSVReader
-- ErrorEditDialog
– RecordsManager
—TreeModel and its derivatives are called depending on what files are loaded
--TreeModels are created using public member functions of the RecordsManager to access the data

Comment on your findings, with specific references to the design/code elements/file names/etc.: TreeModel uses inheritance to encapsulate the various types of tree model's that exist. Allows for polymorphism. 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.

(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)

o yes o no o 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.

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

UserSelectedList

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Went through the code and headers for UserSelectedList and mainwindow and compared with the class diagram.
Comment on your findings: The class diagram and the code are in sync.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the code in mainwindow especially the added functions for saving state. Examined the headers and code for UserSelectedList.
Comment on your findings: The UserSelectedList makes a vector of all the selected list items and has functionality to add, edit and display the list. The savestate functionality is added in the mainwindow and it has functionality of saving the current state for each tab and the application and to load from the previously saved state.

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Examined the functions and code within the classes.
Comment on your findings: All the methods in UserSelectedList are specific and are well encapsulated. Hence there is good-cohesion. The savestate inside mainwindow interacts with other classes to implement it's functionality and the level of cohesion is good.

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.)

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Examined the implementation of functionalities with in UserSelectedList. Mainwindow has complete dependence on all other classes and as he savestate functionality .
Comment on your findings: There is no interdependence between UserSelectedList and other classes. So the coupling is not there. There is a lot of interdependence between the savestate functionality in mainwindow but it is out of necessity.

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: The whole recquirements are broken into small components and each class is used to implement a component.
Comment on your findings: The UserSelectedList class is used to make a vector of the individuals the user has selected. The requirements set by user were properly encapsulated into the class. The save state functionality was implemented within the mainwindow and as such has connections with other concerns but that is the requirement.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files of classes for the access specifiers of data and methods.
Comment on your findings: The appropriate functions in UserSelectedList are made public which are later used in mainwindow , but the data members are private so there is proper use of access specifiers.

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Examined the code and functionality of the classes.
Comment on your findings: The savestate functionality is designed specifically for this application and as such cannot be used in other situations. The UserSelectedList can be used in other applications.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examining the code to evaluate its ease of understanding and ease of identifying the functionality. Evaluating the naming conventions used for classes and methods.
Comment on your findings: The UserSelectedList has functions that are easy to understand. The excessive functionality implemented within the mainwindow makes it very difficult to understand and results in poor readability

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Scanned the code for these classes
Comment on your findings: The UserSelectedList class as well as the code where the UserSelectedList and savestate functionality is used in mainwindow is properly 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: The methods in UserSelectedList are generic and can be updated. But the code in mainwindow is very difficult to maintain and upgrade.
Comment on your findings: The UserSelectedList class are easy to maintain and upgrade. The maintainability of code within the mainwindow is low and can be improved.

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Examine the complexity for the various methods
Comment on your findings: There are no unnecessary conditional statements or nested loops

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files for both the classes
Comment on your findings: These two classes have no inheritance.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the headers for other classes
Comment on your findings: The classes doesn't have children. 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:
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.

1. Adding items to User Selected List
2. Anticipated frequency of use; High
3. End-user trigger: User clicks the Add button in the main window
4. Expected type of outputs: The selected index is added to the list
5. When the button is clicked--the USL is checked for duplicates--if not then the index is added--the clear and undo buttons are enabled.

1 . Displaying UserSelectedList

2. Anticipated frequency of use; High
3. End-user trigger: User clicks the Show button in the main window
4. Expected type of outputs: The present UserSelectedList is displayed.
5. On click--first view is cleared--then the vector containing the indices of selected elements is iterated and those elements are displayed in the list.

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

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)

o yes o no o 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.

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

SaveState

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Went through the code of mainwindow and compared with the class diagram.
Comment on your findings: The class diagram and the code are in sync.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the code in mainwindow especially the added functions for saving state.
Comment on your findings: The savestate functionality is added in the mainwindow and it has functionality of saving the current state for each tab and the application and to load from the previously saved state. The code performs the intended operations.

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Examined the functions and code within the classes.
Comment on your findings: The savestate inside mainwindow interacts with other classes to implement it's functionality and there is normal level of cohesion.

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

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Mainwindow has complete dependence on all other classes as does savestate functionality .
Comment on your findings: There is a lot of interdependence between the savestate functionality in mainwindow but it is out of necessity.

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: The whole requirements are broken into small components and each class is used to implement a component.

Comment on your findings: The save state functionality was implemented within the mainwindow and as such has connections with other concerns but that is the requirement. The requirements set by user were properly encapsulated into the class.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files of classes for the access specifiers of data and methods.
Comment on your findings: The appropriate functions in UserSelectedList are made public which are later used in mainwindow , but the data members are private so there is proper use of access specifiers.

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon’t know

Comment on your analysis: Examined the code and functionality of the classes.
Comment on your findings: The savestate functionality is designed specifically for this application and as such cannot be used in other situations.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examining the code to evaluate its ease of understanding and ease of identifying the functionality. Evaluating the naming conventions used for classes and methods.
Comment on your findings: The excessive functionality implemented within the mainwindow makes it very difficult to understand and results in poor readability

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Scanned the code for these classes
Comment on your findings: The UserSelectedList class as well as the code where the UserSelectedList and savestate functionality is used in mainwindow is properly 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: The code in mainwindow is very difficult to maintain and upgrade.
Comment on your findings: The maintainability of code within the mainwindow is low and can be improved.

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Examine the complexity for the various methods
Comment on your findings: There are no unnecessary conditional statements or nested loops

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the header files for both the classes
Comment on your findings: These two classes have no inheritance.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examined the headers for other classes
Comment on your findings: The classes doesn't have children. 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:
i) Title of scenario
ii) Anticipated frequency of use (high, normal, low)
iii) End-user trigger (starting point) for the scenario.
iv) Expected type of outputs.
v) List of bullet points linking end-user inputs and identifying all the key features of the system expected to be "touched" by the scenario and producing the anticipated outputs.

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

1) Save State
2) Anticipated frequency of use : High
3) End-user trigger : Click Close button on tabs or Window
4) Expected type of outputs: The state is saved to a temporary file.
5) MainWindow User trigger
-- using RecordsManager and CSVReader to get Headers and save them
-- getting managers and saving to file.
Comment on your findings, with specific references to the design/code elements/file names/etc.:
Whenever the user closes a tab, or the entire app, the session state is saved to a temp file. Whenever the program is reopened the user will be prompted to continue from their last session, which will load these temporary files. If they decline, a new session state will be started
(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)

  o yes o no o 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.

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

ErrorEditDialog

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Compare class and class diagram
Comment on your findings: ErrorEditDialog and the class diagram are matching

Functionality:

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

o Yes oNo oPartly (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: ErrorEditDialog is doing what expected, it opens a window and allows the user to fill the missing fields

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Re-evaluate class and members
Comment on your findings: This class has its proper members and functions

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.)

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Examine structure of class
Comment on your findings: this class doesn't depend on any other class; it is instantiated by the mainwindow class (which instantiate all the used 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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examines class
Comment on your findings: ErrorEditDialog has it own concern

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examine header class
Comment on your findings: Acces specifications of this class are correct.

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Analyse code and header
Comment on your findings: This class is proper to the CSV file

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyze the method names
Comment on your findings: Each method functionality can be easily known by reading their names.

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Examine code
Comment on your findings: This class doesn't have a lot of 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Analyse code
Comment on your findings: This class can easily be extended

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Look at the code
Comment on your findings: The code implements functionalities that are used and it's neat

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the header file of the class
Comment on your findings: This class doesn't inherit from any other class

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the header file of the class
Comment on your findings: The class is 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:

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

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

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

_____

_____

(Note: expand here as necessary for each scenario)

Scenario 1:

i) Load teaching data and fix errors
ii) High
iii) Open Libra Galaxy

iv) List of teachings

v) - The user loads a file to the current dashboard
- He answers Edit to message box asking to fill mandatory fields
- ErrorDialogEdit class is called with the data from the file
- All the missing fields are filled, so the new data is saved to the table widget

names/etc.: the file is loaded to the application with all the rows

Scenario 2:

vi) Load teaching data without fixing errors

vii) Normal

viii) Open Libra Galaxy

ix) List of teachings

x) - The user loads a file to the current dashboard
- He answers Edit to message box asking to fill mandatory fields
- ErrorDialogEdit class is called with the data from the file
- No data saved as he clicked on cancel

Comment on your findings, with specific references to the design/code elements/file names/etc.: the file is loaded without the rows with missing fields

Scenario 3:

xi) Load teaching data and fix some errors

xii) High

xiii) Open Libra Galaxy

xiv) List of teachings

xv) - The user loads a file to the current dashboard
- He answers Edit to message box asking to fill mandatory fields
- ErrorDialogEdit class is called with the data from the file
- He fixes only a few errors, so when the method on_save_clicked is called; the action is not proceed

Comment on your findings, with specific references to the design/code elements/file names/etc.: if the user want to fill some missing fields, he has to complete each one of the missing

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)

  o yes o no o 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.

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

QSortListO and TreeItem

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Compare the class diagram with the implemented code
Comment on your findings: the class diagram and the TreeItem class match, it's a match between the QSortListO class and the diagram

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

o Yes oNo oPartly (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: QSortListO sorts the data as expected and TreItem contains items of the tree

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Analyse the methods of each class
Comment on your findings: QSortList has the function to create lists to sort and TreeItem's role is to contain the items of the tree

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.)

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Analyse the header files for "include"
Comment on your findings: The two classes don't depend on any other class of the program (only librairies)

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Observe different cases
Comment on your findings: TreeItem contains items of TreeModel; QSortListO is sorting the data

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look at the header files
Comment on your findings: Both classes use proper access specifications

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Consider other situations where the classes can be used with minor changes
Comment on your findings: TreeItem can easily be resused for other situations; QSortListO can be reused but will need more changes to adapt

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look at the class and method names
Comment on your findings: TreeItem class name can be understood easily as well as its methods, QSortListO is understandable for someone who knows what the program is doing

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look at the code to see if comments are present
Comment on your findings: QSortListO has some comments to explain the code lines and TreeItem has comments for each 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Look at the code to see if enhancements are possible
Comment on your findings: Both classes can be updated to be used for other dashboards.

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Look at the code and the presence of loops
Comment on your findings: QSortListO doesn't have loop and doesn't call a lot of functions; same for TreeItem

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look at the include in the header
Comment on your findings: TreeItem inherits only from TreeModel; QSortListO only inherits from Qt libraries

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: look at the classed using those classes
Comment on your findings: the only class depending on those classes is mainwindow

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

Each scenario is described as follows:

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

iv) Expected type of outputs.

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

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

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

_____

_____

(Note: expand here as necessary for each scenario)

TreeItem is not used by the user.

Scenario 1:

i) Load CSV file and sort

ii) High

iii) Open Libra Galaxy and load a file

iv) Vectors with data sorted by division

v)
- the user loads a file to one dashboard
- He creates a new sort order
- The fields are sent to QSortListO
- The new sort order is added to the list

Comment on your findings, with specific references to the design/code elements/file names/etc.: the treeModel displays the data sorted by division first

Scenario 2:

vi) Sort using a vector of length 0

vii) Low

viii) Open Libra Galaxy and load file

ix) Error dialog

x)
- the user loads file in one tab
- He creates a new sort order
- No new sort order is saved

Comment on your findings, with specific references to the design/code elements/file names/etc.: after clicking on save an error message is displayed and the sort order is not saved

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)

  o yes o no o 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.

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

CustomSort

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Compare diagram with header file
Comment on your findings: Class and diagram match

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Test the program
Comment on your findings: CustomSort classes implement the dialog box to create a new sort order and save it

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Inspect methods
Comment on your findings: CustomSort methods are accessing the same data members

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.)

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Check the dependences with the other classes
Comment on your findings: this class doesn't depend on other class; it uses Qt libraries

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the purpose of the class
Comment on your findings: The class doesn't use any subclass for the different dashboard

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyze the header file of the class
Comment on your findings: The class uses proper access specifications, only two methods used in other classes declared as public

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Consider other situations where the class can be reused with minor changes
Comment on your findings: This class is specific to this project, it's not really reusable

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyze the class and methods names
Comment on your findings: the class name is very clear; it uses a few methods with understandable names

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look for comments in the code
Comment on your findings: The class has a few comments, but there is not really complicated portions of code

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Analyze the possible modifications to the code
Comment on your findings: for most of the enhancements it will not required a lot of changes

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Analyze the code of the class
Comment on your findings: This class doesn't have any nested loop

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the dependence of this class with others
Comment on your findings: this class doesn't depend on any other class

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: analyze the classes depending on this class
Comment on your findings: the only class using CustomSort and its methods is mainwindow

Behavioural analysis:

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

Each scenario is described as follows:

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

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

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

_____

_____

(Note: expand here as necessary for each scenario)

Scenario 1:

i) Edit date range
ii) H
iii) Open Libra Galaxy and load a file
iv) Updated data and graphs
v) - the user changes the start/end date
    - Mainwindow get the changed date
    - Mainwindow accesses RecordsManager to request data for the new range
    - Mainwindow builds a new tree model

Comment on your findings, with specific references to the design/code elements/file names/etc.: if the date range got reduced, less data is displayed

END.

Design Inspection Instrument

Instructions:

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

  o yes o no o 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.

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

PieChartWidget

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Compare diagram with header file
Comment on your findings: Class and diagram match

Functionality:

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Test the program
Comment on your findings: PieChartWIdget is a part of the UI, to test it, the user needs to select pie chart for display

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)

o Yes oNo oPartly (Can be increased)

Comment on your analysis: Inspect methods
Comment on your findings: PieChartWidget methods have a specific function that is to enhance the implementation of the pie chart graph

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.)

o Yes oNo oPartly (Can be reduced)

Comment on your analysis: Check the dependences with the other classes
Comment on your findings: PieChartWidget doesn't use other classes variables and has a minimal number of includes

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?

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the purpose of the class
Comment on your findings: The class doesn't use any subclass for the different dashboard

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyze the header file of the class
Comment on your findings: The class uses the proper access specifications for its variables and methods

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

o Yes, most of the classes oNo, none of the classes oPartly, some of the classes oDon't know

Comment on your analysis: Consider other situations where the class can be reused with minor changes
Comment on your findings: PieChartWidget can be reused with a few changes

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyze the class and methods names
Comment on your findings: The names of the class/methods are understandable but not very clear

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Look for comments in the code
Comment on your findings: PieChartWidget doesn't have any 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)

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Analyze the possible modifications to the code
Comment on your findings: for most of the enhancements it will not required a lot of changes

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

o Yes oNo oPartly (Can be improved)
oDon't know

Comment on your analysis: Analyze the code of the class

Comment on your findings: PieCHartWidget uses a lopp but is quite efficient

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).

o Yes oNo oPartly (Can be improved)

Comment on your analysis: Analyse the dependence of this class with others
Comment on your findings: PieChartWidget doesn't inherit from anything

Children:

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

o Yes oNo oPartly (Can be improved)

Comment on your analysis: analyze the classes depending on this class
Comment on your findings: this class doesn't have 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:

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

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

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

_____

(Note: expand here as necessary for each scenario)

Scenario 1:

i) Show a pie chart
ii) Anticipated frequency of use: High
iii) Click field in a tree view model
iv) Pie chart graph displayed
v) - the user select pie chart graph for the selected member
    - The field string is sent to RecordManager
    - The correct data is passed to qcustomplot and piechartwidget

Comment on your findings, with specific references to the design/code elements/file names/etc.: the graph with the data regarding the member is displayed using the pie chart graph, to be updated (select another member) the user has to click on the member name and not on the total (or other integers)

END.