**CS 542 Group Project**

**File Viewer**

by Azadeh Iranmehr, Leila Vajed, Bin Wang

**Project Goal:**

This project focuses on creating an intelligent "tail" program for viewing live updates to log files and providing parsing tools. It allows a developer to view real-time updates to log files produced by debugging applications.

**How the Program Works:**

1. ***File Selection:***

The application allows the user to select a text file by the browser file system.

By running the program in QT , the user will see a window popped out called 'MainWindow.' It has one main menu called 'File' and two submenus called 'open a text file 'select a directory. (See 1.1) The submenu 'open a text file' allows the user to select text files (TXT, XML, Html, JSON, and Dat) in any directories. After selection, the address and the selected files' names will be displayed in the window(See 1.2). The application will automatically load the most recently modified file and continuously check that text files for a any update.

Graphical user interface, application

Description automatically generated A screenshot of a computer

Description automatically generated

1.1 1.2

Graphical user interface, application

Description automatically generated

1.3

1. ***Auto-Load:***

The other submenu, ‘select a directory,’ allows the user to select a directory containing text files instead of choosing a specific file. After the selection, the directory name and address will be listed in the listView of UI. If we add any new file, the application will automatically load the new file and continuously check the directory for any new file to open if we double click on the directory name in the listView.

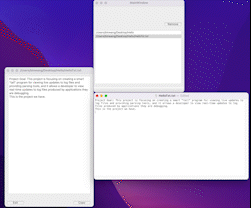
1. ***Tailing:***

the application continuously polls the target file for update and display to the user. First, the user needs to double-click on the file name to open the file. When the user double click on the directory name in the listView, all its files will be shown in its viewer window(See 1.4). All files under the directory will be opened and ready to view. (See 1.4). For text files, the application will open only the selected file itself. (See 1.5) Each file viewer window will have ‘Find’, 'Exit' and 'Copy' buttons to search for a list , close the file viewer and pop out the selected file's content.

When the user is editing the file outside the file viewer program (see 1.6), any changes in the original file will be updated and seen in the file viewer while the new file has been updated. This real-time updating functionality also applies to creating a new file under one directory or creating a new directory under an existing directory.

Graphical user interface, text, application, Word

Description automatically generated Graphical user interface, text, application

Description automatically generated 

1.4 1.5 1.6

1. ***Regular Expressions:***

The application has three .ui files under folder ‘Forms’ (See 1.7). The ‘mainWindow.ui’ generates the main window of the application. It has the buttons such as menu and submenu, and it also lists the name and address of the file after the user selects. The ‘logviwerform.ui’ generates a pop-out viewer window after the user double clicks the file chosen. It has 3 function buttons for users to click: ‘find’, ‘exit’ and ‘copy,’ and it has a text browser to show content of the text file and the text line to insert list of search separated by ‘,’. The find function helps the user to find/highlight important messages by typing list of the words in the search line and clicking find. (See 1.8)

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

1.7 1.8 Regular Expressions

1. ***Capture:***

The application also allows the user to capture some text and pop out into another window called ‘Popup’ at the click of the ‘Copy’ button. The ‘PopupWindow.ui’ generates the below pop-out window (See 1.9) by selecting the text wanted and clicking copy, and the Popup window will show the text chosen by the user.

**Button ‘Copy’:**

When we click on the copy button in the log viewer, a popup menu containing the selected text that the user copied should be open. We created a slot on-click function on the copy button for this functionality. First, we call the function copy to copy the text's content in the clipboard. We then created an object of the class clipboard to paste what we have in the clipboard, then create an object of the popUpWindow from that text and add that popUpWindow to the list of popUpWindow(popups) and show this popup in UI. To enable the function of opening several popups, we made a list of popups to push back new popups. To create these PopupWindows, we initialized its text and its viewer to show it in the UI in its constructor. (See 2.0)

**Button ‘Close’:**

* When we click on the close button of popup windows, closeEvent will be triggered automatically.
* Popups should be closed.
* We call viewer->dropWindow(this) in close\_event that delete those popups from the popups list and delete it from memory.

dropWindow() is defined as one of the virtual functions in the logViewrForm class. Actually, in PopupWindow::closeEvent(QCloseEvent \*event), we override closeEvent() function of the window class because we want to delete that popup windows from the list of popups in class logViewrForm and delete it from memory. (See 2.1)

Graphical user interface, text, application

Description automatically generated Text

Description automatically generated

1.9 2.0 LogViewerForm.cpp

Text

Description automatically generated with medium confidence

2.1 LogViewerForm.cpp

**‘Search’ and ‘Find’ Functionality:**

We add lineEdit to the logviewerForm in the UI, which functions as a text editor to insert long text in one line). We also added a button, 'Find.' When the user clicks on the button, all text in lineEdit should be copied in a variable called 'searchLine' first. Each textBrowser in the logviewer (that shows text file in UI) has a document class. It is like a word document that maintains the text document and all properties like color, font, etc. To search for what we have in the variable 'searchLine', we must split 'searchLine' by ','. We insert a list of searches separated by "," in editLine. Then check each string for each string in the searchList, and we define two cursors if it is not empty: one default cursor and the other highlighting cursor for the document. It searches through each string using the function find () in the while loop until we reach the end of the document. The highlight cursor acts as a cursor to navigate in the search loop. It selects the string and changes its color from black to red when it finds the specific string. (use QT documentation to implement search function.) (See 2.2)

Text

Description automatically generated

2.2 LogViewerForm.cpp

**Timer and Mulltithreading :**

We also have multithreading in our application by setting a timer in connect command in constructor on mainWindow class so that all the dataModels will be updated after the timer's time out.

The timer is one of the threads running behind the scenes that we set to reset every 300 milliseconds. When the timer times out every 300 milliseconds, all the dataModels get updated to show the most recent models(textFiles). Another thread is our main application. So, we have two threads or more, and we have one thread for each opened window. (See 2.3)

Text

Description automatically generated

Text

Description automatically generated

2.3 MainWindow.cpp

**Design Pattern:**

Overall, we have combined some patterns in the same class, hence some classes have been implemented more than one pattern in QT creator. First, the ‘mainwindow.cpp’, ‘mainwindow.UI’ and ‘main.cpp’ are the three default files that was automatically generated by clicking on ‘main.window’ when we create the new project File Viewer.

1. ***Observer Pattern #1:***

Diagram

Description automatically generated

To achieve the goal of the tailing functionality that the application can continuously poll the target file for Update and display to the user, we decided to use the observer pattern. The observer logic is implemented in the ‘textFile’, the subject class is ‘textFile’, the ‘ILogviewer’ is the observer/subscriber class, and the ‘LogViewerForm’ is acting as the concrete observer that inherited from the ‘ILogViewer.’ The observer can be used when one object changes its state. In this case, the text file’s content has changed, all dependent objects should be notified and updated automatically by calling one of the functions called Update (). Hence, any changes in the text file should be updated and shown in the viewer. When the user is editing the file, any changes in the original file will be updated and seen in the file viewer when the new file is saved. So, the viewer can update any file in real-time if any updates happen to the text file.

1. ***Observer Pattern #2:***

***Diagram

Description automatically generated***

There is an Observer pattern in QT connects the QListView to the QstringListModel. The QstringListModel acts as the subject/model, and the QListView act as the views. Any update in the model will be notified in the ListView, and the ListView should be updated automatically. We use this pattern in two functions: selectDirectory() and openTextFile(). This list should be updated when we want to add a selected text file or directory or if we want to remove a row from the list of files and directories in the listView. QListView should be updated when any modifications occur in the Model (QtringListModel).

1. ***Adapter Pattern:***

Diagram

Description automatically generated

Adapter Pattern allows two incompatible interfaces to work together, and it converts the interface of a class into another interface clients expect. QlistView cannot work with strings directly, but it can work well with QStringListModel instead. If you frequently need to modify the list of strings and have connected views that need to be updated, String should be wrapped by QStringListModel to prevent tedious work. So we use adaptor pattern here and set the ‘String’ acts as the Adaptee, and the ‘QStringListModel’ functions as the Adaptor. This Adapter locates between QlistView of UI and String, and its role is to wrap String to QStringModel.

The adapter pattern has been used in the following two functions: openTextFile() and selectDirectory().



dataModel is our adaptor that wraps strinf name(file->name()) that is our adaptee .

but in the mainWindow constructor, we have to create our adaptor and set the ui QlistView objet(selectedFileView) model to this dataModel.



1. ***Command Pattern:***

Diagram

Description automatically generated

The command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.

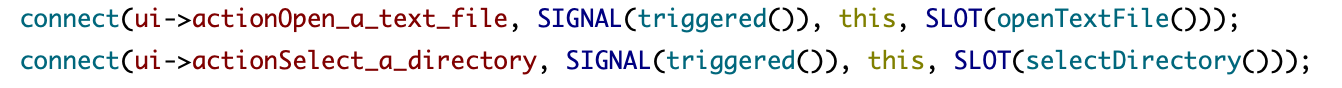
We can connect the Menu and the mainWindow by using the connect function of QObject class. The Menu acts as the sender, and mainWindow is considered as the receiver. It determines which function of the receiver class should be executed in any of the following signals: trigger, click, dbclick, etc. Clicking on the menu(QAction object) executes a function in the receiver object(mainWindow). Hence, the object of the mainWindow encapsulates an action that needs to be taken when the menu item is triggered or when the menu item is clicked. Thus, click on Menu--> executed function in mainWindow. The function 'connect' does an excellent job for us because it creates a command design pattern by specifying sender and receiver objects and the function executed when the Menu is triggered.

We define slot functions in the ‘mainwindow.h’, “open a text file” and “select a directory”. These are the functions that should be execute by clicking on “open a text file” and “select a directory” menus.

Text

Description automatically generated with low confidence

There are our connect functions to create 2 command patterns:



Signal is the operation that we need to run that command triggered (). SLOT is our function in the receiver object that should be run when the sender object is triggered.

1. ***Composite Pattern:***

Diagram

Description automatically generated

The Composite pattern is used when we need to treat a group of objects similarly as a single object. It is a tree structure that one object can have multiple objects, and that object can have multiple objects. Whatever the operation you can perform on the leaf node, the same operation should also be performed on the composite object.

In this case, the class ‘ifile’ acts as the Component, the class ‘Directory’ acts as the Composite and the class ‘textfile’ acts as the leaf. So, the Directory can have multiple text files and many different directories, and there can be various text files and directories under these directories.

1. ***Factory Method Pattern:***

Diagram

Description automatically generated

The client can instantiate Directory or TextFile objects indirectly just by using fileCreator.

For the factory method, it is essential to say that when we use this pattern, we don't need to instantiate the file and directory by using new in the client (mainWindow). We can just include and use the createFactory class to create both objects in the client.

The create factory () has two parameters:

* the first parameter determines which class we want to make (file or directory)
* the second parameter is the address of the file and directory

**Reference**

for the search code: <https://doc.qt.io/qt-5/qtuitools-textfinder-example.html>