C++ Program report

# Introduction

The program is a collection management system that allows the user to keep track of a collection of objects.

Many people collect things, however, they often have a problem keeping track of the different objects. Different methods of collection organisation have been attempted; these methods have had varying success, from very simple systems such as spreadsheets to complex collection management systems.

 This project aims to create a very simple object-oriented system that allows the user to store, retrieve and modify a list of objects. It aims to create a simple system that allows the user to store, retrieve and modify a list of objects in a way that other programs can use compared to other collection management systems.

# Methodology and analysis

This program is written in C++ to compile the program. C++ was chosen because of its object-oriented approach to programming; this OOP (object-oriented programming) approach was used, first because it allows the program to be compartmentalized into separate sections and secondly, it makes the program structure much clearer than it would be without the classes.

The classes and the objects that are created from them are a form of abstraction, where the different parts of the program

 the information inside the class is hidden from the outside and may be only accessed from the public methods that form an interface between the

This use of classes to encapsulate is not strictly necessary, however, because the different parts of the program can be placed into separate source files and the variables can be accessed using getter functions rather than directly accessing them.

However, when using classes to encapsulate the different parts of the program makes the separation of the different areas of the program formal, with the internal structure of the class being declared as private; these variables are only able to be accessed within the class.

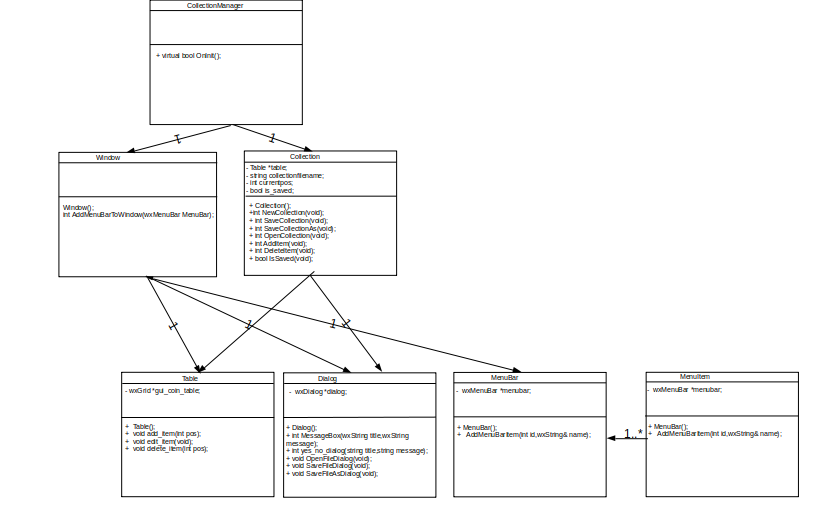
This project uses both methods to organize itself; the advantages of using a separate file for the class methods, which has a header to declare the class itself and any #define statements is that first, it is the obvious

Many public methods are used to access the data within the class and to provide different functions. An example is the Dialog class, which is responsible for creating the dialogs that the user will see; it has methods that create the message boxes, file open and save dialogs and a yes/no dialog box. The programmer only has to create a Dialog object and then call the methods in the class.

The use of classes in this method has the advantage that objects can be created when they are needed; the collection class uses temporary dialog objects to show the open, save and save as dialogs.

Another reason that the use of classes is an advantage is it prevents the structure of the program from becoming disorganised. If, for example, a program is written in C that does not use encapsulation and directly accesses the data, then it will be difficult to modify the program without causing problems to the other problems for the other parts of the program. Using a global variable to store state information can cause problems for the program; one function can modify the variable, while a second changes it again. The first function may have problems if it expects the value to not have changed.

Finally, using classes can be derived from a parent class. These classes can use the public functions of the derived class as though they were called from the derived class. This technique has the advantage of being able to create classes that use the methods and variables. An example is the main frame used by the program; it is derived from the wxFrame class and uses the methods that are exposed from it.

****

**Figure 1. Class structure of the program**

The program is divided into eight classes; Collection, CollectionManager, Dialog, Menu, MenuBar, Window and Table. Each of the classes implements a different part of the program:

* The collection manager class is derived from the wxApp class. It is the first class to be used and provides the main function….
* The collection class provides a bridge between the Table class, which stores the information the program is currently using and the Dialog class. It creates a unified interface layer between the Table class and the Dialog class.

This structure was chosen, first because it was convenient to encapsulate the common logic for displaying a dialog for the user to open a file and loading the information from the file to the table.

The second reason that this technique was used is that the methods in the Collection class are called from an event handler, which do not allow extra parameters to be passed to them; any method that is called must be able to get the filename from the user without accepting parameters. The method used was to create a dialogue object and call OpenFileDialog, SaveFileDialog or SaveFileAsDialog. These methods return a filename as a string; using these methods to get the filename from a dialog separate from the other methods is an important use of encapsulation. However, these methods are only called from the event handlers, but separating the dialog user interface code remains a useful way to abstract the interface code from the rest of the code.

* Dialog provides common dialogs that display the open, save and save as dialogs. Using the wxWidgets common dialogs has the advantage that creating the dialog was a simple matter of creating objects from the wxFileDialog class and passing the appropriate parameters.

This class implements the file open/save dialogs that were described before and it also implements a simple message box that displays a message and a yes/no message box that returns a boolean value depending on the button pressed.

* The table class stores the data in tabular form; this class has the advantage that it can store, retrieve and display the data using a single class.

The data is stored in a 8 x n sized grid, with columns the image, description,

year issued, grade,weight, height and width. This fixed-width grid was chosen because it has the advantage that the LoadTableFromFile method does not need to calculate the table size.

It only has to use the addRow method provided by wxGrid to append a row; it automatically creates the correct number of columns when a new row is added to the table.

Adding rows to the table and removing rows is a simple matter of using the wxGrid InsertRows and DeleteRows methods.

Loading data into the table is done using the LoadTableFromFile method. This reads data in CSV format into the table first by reading each line from the file into a string and then splitting them into tokens. The tokens are added to the table using SetCellValue. This method, although awkward to implement, make it easy to read the data into the table, because the method used to tokenize each line creates a string that can be added directly into the table.

Saving the data is only a matter of for each row, concatenating each column into a string with commands between each column data item and writing it to a file.

The data is stored in CSV (comma separated variable) format. This was chosen, because it is a portable format that is usable by other programs, such as Microsoft Excel.

* The MenuBar class creates the menu bar and the root menu items on it. It is a simple class that only initializes a wxMenuBar object.

The MenuBar class declared a GetItem function that returns the private wxMenuBar object stored in the class; this is a necessary, but rather ugly technique that creates a copy of the wxMenuBar object so it can be used by wxGrid. The alternative was to declare wxMenuBar to be a public variable, however it would break the class abstraction.

The MenuBar is added to the window using a method in the Window class. It is

structured in this way, first because it uses the SetMenuBar method contained in the Window class. Calling it from the Window class is a simple matter of calling SetMenuBar(window\_object).

* The menu class creates an unattached menu. Each of the menus has an event number and a description that is used to create the menu using the wxMenu class method AppendMenu.

This class and the MenuBar class are kept separate for two reasons, first is that like the other classes abstracting the Menu class will keep the internal mechanisms of the class hidden from the outside. The second reason is that it will allow the Menu class to be reused; an example is that a popup menu object could be created using the Menu class. This popup menu could be then implemented using the object by using an event handler to display the menu.

The program is partly event-driven and partly object-oriented; the event-driven aspects of the program allow it to flexibly respond to the actions of the user. This is demonstrated when the user clicks on a menu; an event handler is called whenever a menu is clicked on.

These event handlers, as was discussed in the Menu class paragraph are assigned a number that associates the object with an event and links it to a method that is called when the event is called:

  wxBEGIN\_EVENT\_TABLE(MainFrame,wxFrame)

    EVT\_MENU(NEW\_FILE,CollectionManager::FileNew)

    EVT\_MENU(OPEN\_FILE,CollectionManager::FileOpen)

    EVT\_MENU(SAVE\_FILE,CollectionManager::FileSave)

    EVT\_MENU(SAVE\_FILE\_AS,CollectionManager::FileSaveAs)

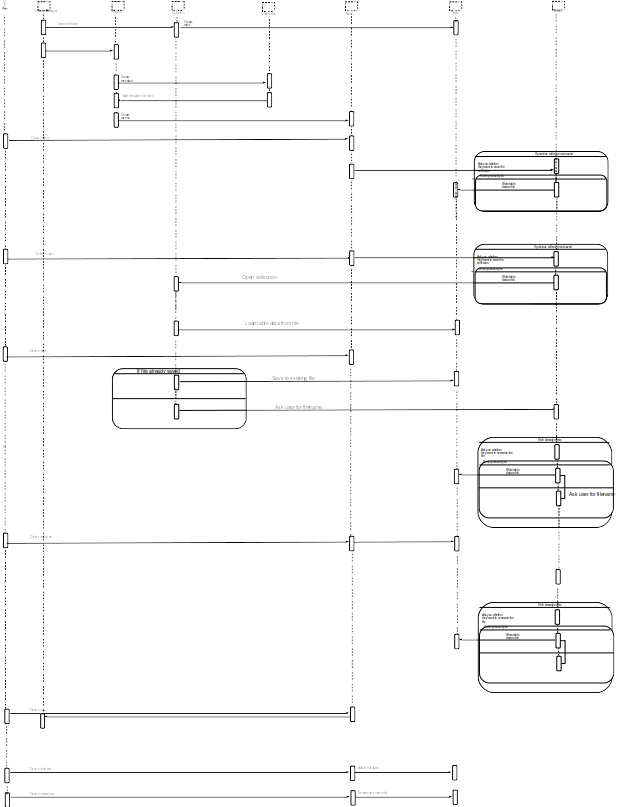
    EVT\_MENU(ADD\_ITEM,CollectionManager::ItemAdd)

    EVT\_MENU(DELETE\_ITEM,CollectionManager::ItemDelete)

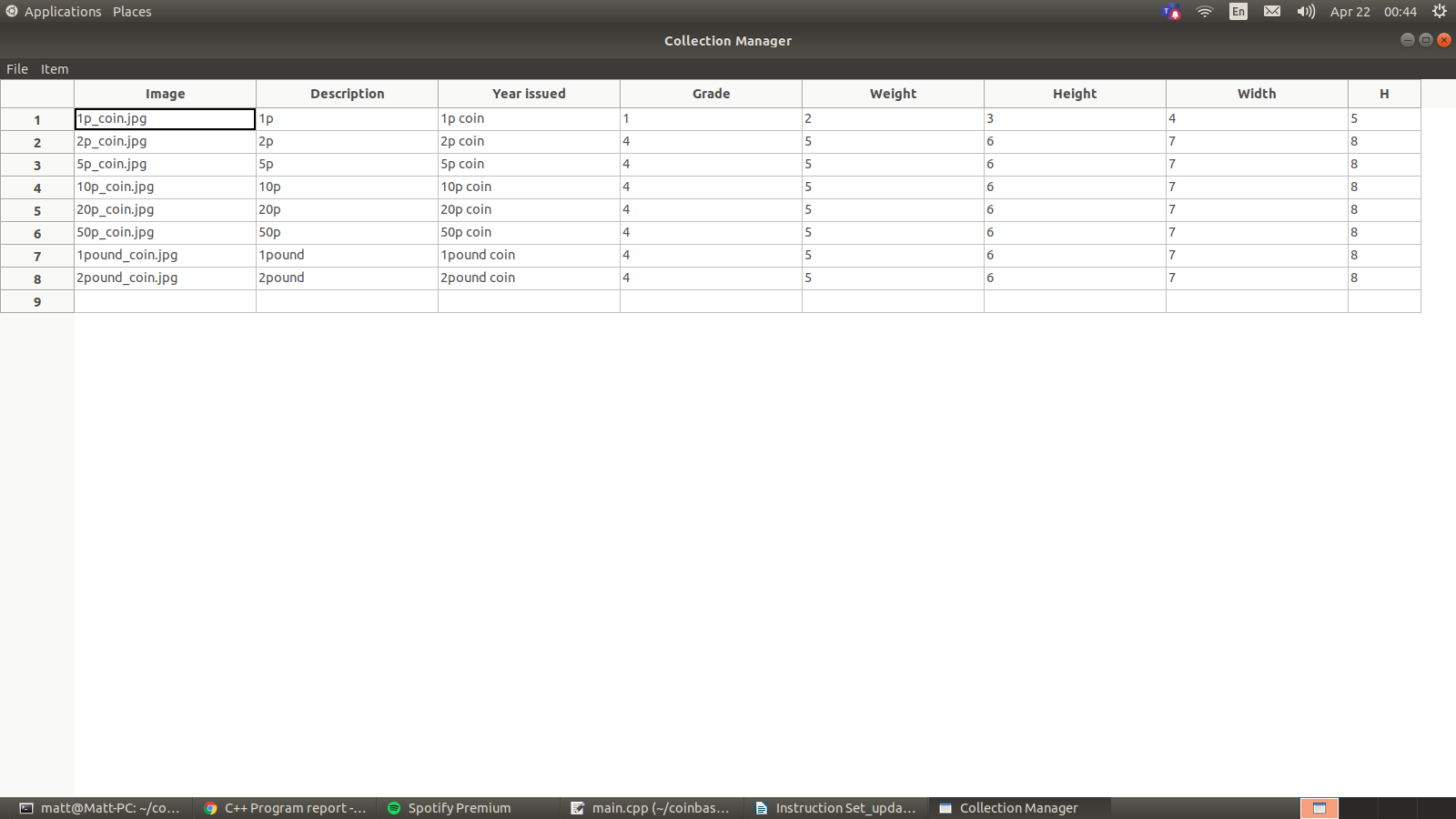
  wxEND\_EVENT\_TABLE()

Fig 2. wxWidgets uses macros to generate the event handler table.

Behaviour diagram for the program



# Design of the user interface



The graphical interface was designed to be as simple as possible, with a minimal number of elements. This was chosen to implement the interface in code as simple as possible; the number of classes written to create it has been reduced to a bare minimum of five from the large number of classes that would be implemented if more features were added.

The layout of the interface, like the underlying design, is modular and simple, with each part being interdependent with each other and creating a whole interface that can be extended easily by adding new features.

The graphical interface was created using the wxWidgets library. This library was used because of convenience, firstly because it is simple to create a graphical interface using the functions that are provided; the classes were easy to use and well documented which made it simple to implement the graphical user interface elements. The wxWidgets library hides the details of creating the user interface; if the internals of wxWidgets changes, then the program that uses it will not have to change. This stability allows Collection Manager to be compiled on any future version.

 The second reason that wxWidgets was that it is portable; the program was developed on Linux, but it would be simple to compile the program on a Windows system

# Testing of the program

|  |  |  |  |
| --- | --- | --- | --- |
| Test number | What to test | Expected result | y/n |
| 1 | Window is created | Window is created | y |
| 2 | Menu   bar and menus are created and are visible | Menu   bar and menus are created and are visible | y |
| 3 | Can get menu variable from object | Can get menu variable from object | y |
| 4 | Menus work as expected, each function is called when the menu is clicked on | Menus work as expected, each function is called when the menu is clicked on | y |
| 5 | Asks user to save before loading new file or quitting | Asks user to save before loading new file or quitting | y |
| 6 | New collection is created on program start | New collection is created on program start | y |
| 7 | Table is created when new collection is created | Table is created when new collection is created | y |
| 8 | Table has single row when created | Table has single row when created | y |
| 9 | Table header display correct names | Table header display correct names | y |
| 10 | Table header has correct number of columns | Table header has correct number of columns | y |
| 11 | Table shows image | Table shows image | n |
| 12 | New collection can be created | New collection can be created | y |
| 13 | Collection can be loaded | Collection can be loaded | y |
| 14 | Collection can be saved | Collection can be saved | y |
| 15 | Collection can be saved | Collection can be saved | y |
| 16 | Program terminates | Program terminates | y |
| 17 | Item can be added to the table | Item can be added to the table | y |
| 18 | Item can be removed from table | Item can be removed from table | y |
| 19 | Data is loaded into table correctly from CSV file | Data is loaded into table correctly from CSV file | y |
| 20 | Data is saved to CSV file from table | Data is saved to CSV file from table | y |
| 21 | Message box shows title and message | Message box shows title and message | y |
| 22 | Open, save and save as dialogs are displayed | Open, save and save as dialogs are displayed | y |
| 23 | OpenFileDialog, SaveFileDialog and SaveFileAsDialog returns filename | OpenFileDialog, SaveFileDialog and SaveFileAsDialog returns filename | y |
| 24 | Yes/No dialog displays title and message | Yes/No dialog displays title and message | y |
| 25 | Yes/No dialog returns true or false if Yes or No buttons are clicked.= | Yes/No dialog returns true or false if Yes or No buttons are clicked.= | y |

# Conclusion

A collection manager has been created that allows the user to manage a collection of objects.

# References