Stage 1 – Basic GUI

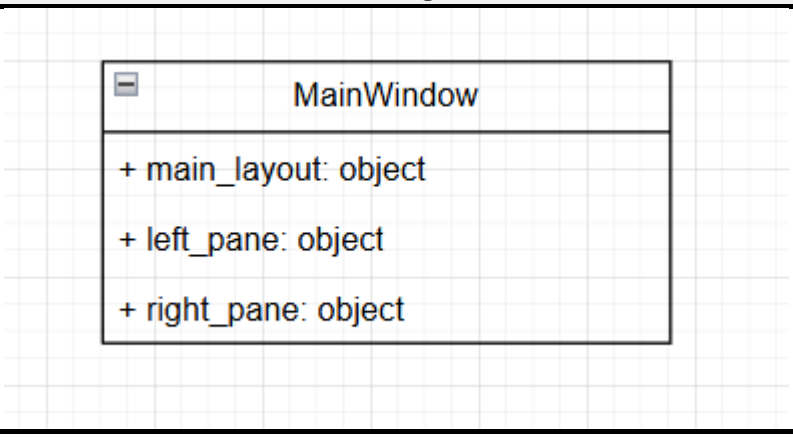
This stage will mainly include getting the basic GUI layout ready and setting it up to make sure it is easy to add new sections.

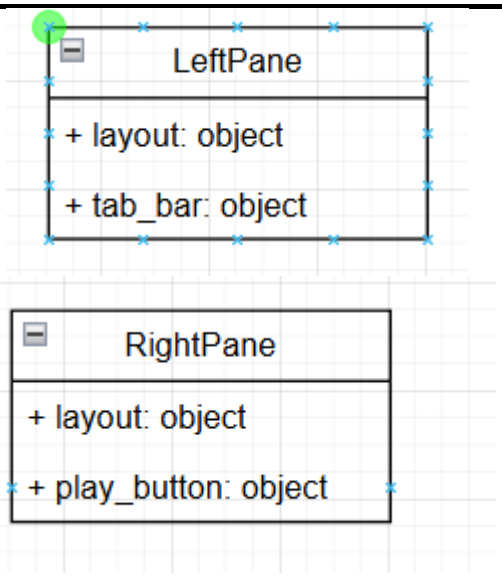
Design

Algorithms	
Algorithm	Explanation/Justification
MainWindow Horizontal layout Two panes LeftPane – tabbed layout RightPane – vertical layout	Main logic for the layout. The main window is split into two horizontal panes, where the left one will contain multiple sections that the user can switch between with a tab bar, and the right pane will always show a now playing window with widgets mostly added vertically.

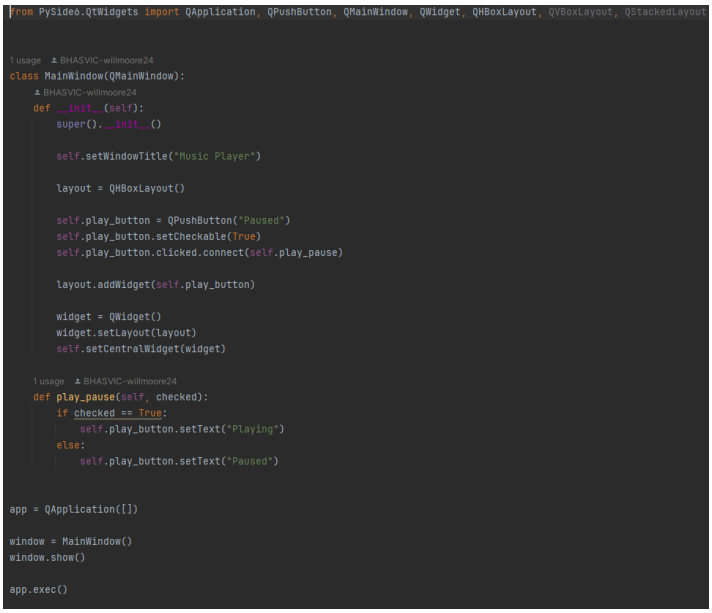
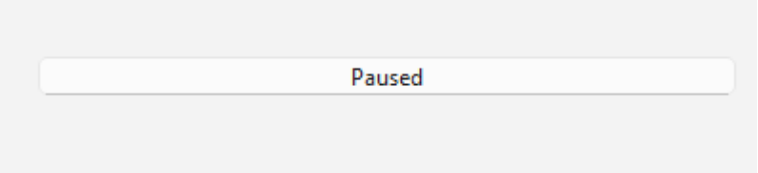
Data Dictionary		
Variable	Type	Explanation/Justification
No important variables at this stage as at this point I am mostly only working with objects included in the GUI library, PySide6.		

Class Diagrams

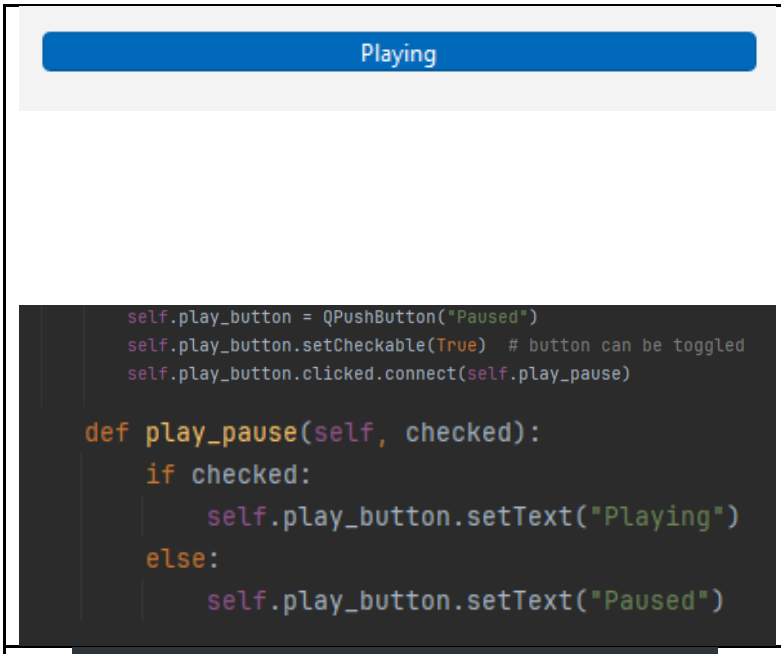




Develop

Screenshot	Explanation/Justification
	MainWindow class will have other widgets included in it. Originally planned to include most widgets in the MainWindow class, but this was changed later. Used the PySide6 docs to find the necessary widgets. https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/index.html
	Simple play button. I added this here as I was originally planning on creating the music playing functionality in this stage, but as I was coding this, I realised it would

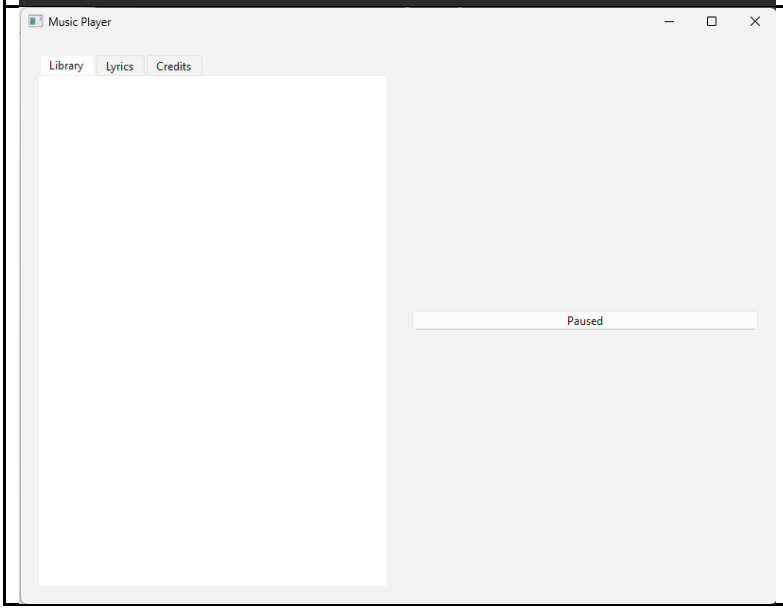
```python
from PySide6.QtWidgets import QApplication, QPushButton, QMainWindow, QWidget, QHBoxLayout, QVBoxLayout, QStackedLayout


1 usage  ▲ BHASVIC-willmoore24
class MainWindow(QMainWindow):
    ▲ BHASVIC-willmoore24
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Music Player")

        layout = QHBoxLayout()

        self.play_button = QPushButton("Paused")
        self.play_button.setCheckable(True)
        self.play_button.clicked.connect(self.play_pause)

        layout.addWidget(self.play_button)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

    1 usage  ▲ BHASVIC-willmoore24
    def play_pause(self, checked):
        if checked == True:
            self.play_button.setText("Playing")
        else:
            self.play_button.setText("Paused")


app = QApplication([])

window = MainWindow()
window.show()

app.exec()
```

	be beneficial to have the library coded first, so the rest of the now playing section will come later. https://www.pythonguis.com/pyside6-tutorial/
	Decided to use OOP classes for each of the main widgets as this allows me to separate out the variables they use (encapsulation) and makes it easy to add new widgets in the future, by just creating a new class.

	Using a horizontal layout (QHBoxLayout) allows for the two panes, where each individual pane will contain nested widgets. https://www.pythonguis.com/tutorials/pyside6-layouts/
	RightPane class for showing currently playing. At the moment, this section will not contain too many different widgets so should be easy to fit all into this class, but in the future I may move widgets such as ‘play_button’ into their own class.

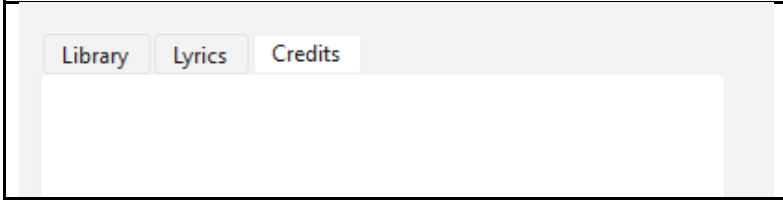
```python
class LeftPane(QMainWindow):
    Will +1
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        self.tab_bar = QTabWidget()
        self.tab_bar.addTab(Library(), "Library")
        self.tab_bar.addTab(Lyrics(), "Lyrics")
        self.tab_bar.addTab(Credits(), "Credits")

        layout.addWidget(self.tab_bar)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

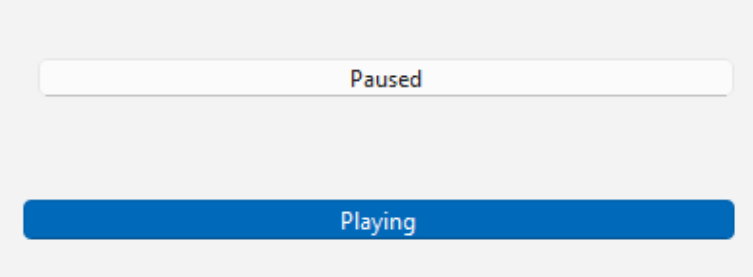
For the left pane I have used the tab bar included in Qt which allows the user to switch between widgets using tabs at the top of the window. This is easy to add new widgets, as I just write the section as a new class and add the tab with self.tab_bar.addTab(<object>, name).



This is what the program looks like at the end of stage one.
On the left, a simple area that each section can be added to easily and switched between with the tab bar.
On the right, space for showing currently playing, and a play/pause button.
This is a basic frame of what the GUI will look like by the end, but the layout is mostly correct, and it will be easy to add to as new features are added.

Testing

Test Description	Evidence
At the moment, tests are quite simple; ensure the layout is correct visually, and all the buttons do what they are supposed to.	
	Tab buttons work fine, as I add widgets this will allow me to switch between

	Play button text updates correctly when clicked. Toggles on and off

Review
So far in stage 1, I have created the main layout for the rest of the program. I have used classes to ensure it is easy to add new sections/features without majorly restructuring code, and sections can be reused if needed. In the future, should be easy to add colours/styling but for now the default white is fine. The simple layout should be easy for the user to use.

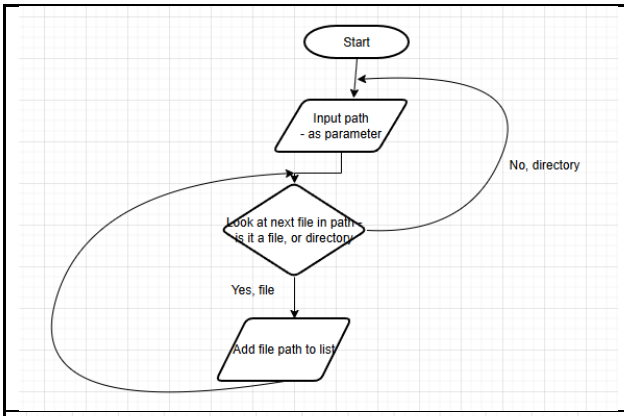
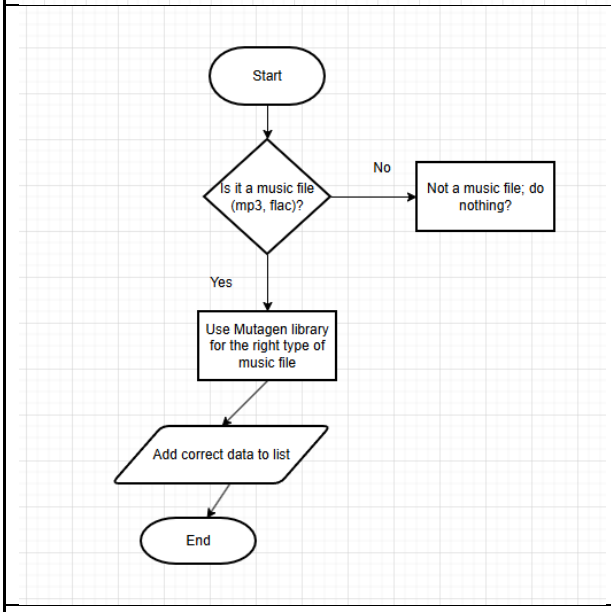
Stage 2.1 – Scan folders

Originally, stage 2 was to create the library management system. On starting this, I realised that my original plan for stage 2 was too large as there was a lot of functionality that needed to be coded first, that the library and later stages are based on.

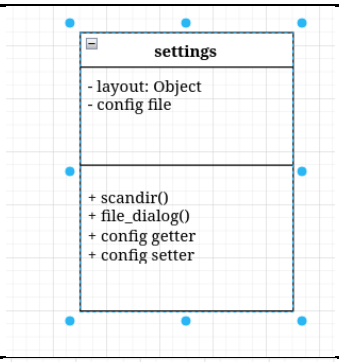
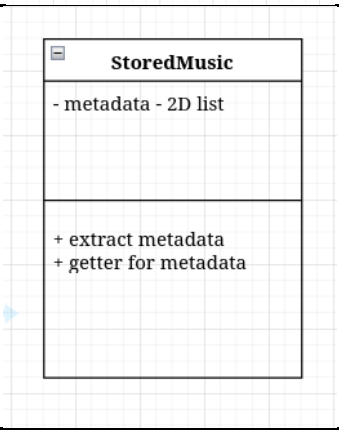
Because of this, I have split stage 2 into two sections, where the first section will be for the necessary algorithms to scan a selected directory and read track metadata, which will be used in most of the other stages later. The second section will be focussed on the library specifically.

Design

Algorithms	
Algorithm	Explanation

	Scanning directory algorithm Initial path entered Looks at each entry in the directory If it is a file, add the path name to a list – this list can then be analysed later to get the metadata, but for the moment just the path works If it is a folder, run the same algorithm, using the path of this folder End when no more items left
	After scanning music directory, need to extract metadata from each file. Need to ensure it is only music files (e.g. an image, video or text file would not work) If it is not a music file – at the moment I will likely just skip over the file, as the program does not need to worry about other files. I could add a warning to the user that the file is not a music file

Data Dictionary & Class Diagrams		
Variable	Type	Explanation
Stored music metadata 2D list structure [[path, artist, album, title, track number, date, genre, length]]	2D list	Structure for the 2D list for storing metadata Each individual list contains all the information relating to one music file
Class Diagrams		

		Settings class diagram – main methods include scanning directory, file picker window and editing the config file
		StoredMusic class diagram Unsure of every method that I will include at the moment, but the main purpose of this class is to extract metadata from a file and store it in a 2D list

Develop

Screenshot	Explanation/Justification
 	Trying to write an algorithm to retrieve the paths of all music files in a directory, and all subdirectories within. I used the 'os' library. Initially, I was using 'os.listdir(<path>)' which returns a string of each item in the directory. This was causing problems as the 'is_dir' function does not work on strings, which listdir() returns, so I switched to 'os.scandir()', where I can check if the return value is a directory or not.

```python
def scandir(path):
    current_dir = os.scandir(path)
    for i in current_dir:
        if i.is_dir():
            scandir(i.path)
        else:
            song_paths.append(f"{i.path}")


directory = "Music/"

song_paths = []

scandir(directory)

print(song_paths[0])
```

```
Music/Elliott Smith\Either_Or\01 - speed trials.flac
```

‘scandir(path)’ procedure recursively scans every file in a directory, and if it is a file adds a string of the path to the song_paths list. If it scans a directory, it will then apply itself to all items in that directory. Will continue until all items in main directory scanned.

```python
from mutagen import (
    flac,
)
test = flac.FLAC(song_paths[0])
print(test["ARTIST"])
```

```
['Elliott Smith']
```

Using the ‘Mutagen’ Python library makes it easy to read metadata tags from files.
This will be used for displaying in the program, as part of the library and now playing sections, and will also be helpful for API calls in the future.

https://www.geeksforgeeks.org/python/extract-and-add-flac-audio-metadata-using-the-mutagen-module-in-python/

```python
def file_dialog(self):
    directory = QFileDialog.getExistingDirectory(self, "Choose directory")
    if directory:
        self.current_dir_text.setText(f"{directory}")
        self.scandir(directory)
        print(self.song_paths)
```

```python
choose_dir = QPushButton("Select Music Folder")
choose_dir.clicked.connect(self.file_dialog)
self.current_dir_text = QLineEdit()
```

I needed to add a way for the user to select the directory of their stored music. Initially, I planned to just have the user type the path into a text box, but on reading the pyqt docs, I found that PySide6 includes a file picker widget.

https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/QFileDialog.html

settings.py

```python
import configparser


class Settings:
    def __init__(self):
        self.config = configparser.ConfigParser()


    def get(self, section, setting):
        self.config.read("config.ini")
        value = self.config.get(section, setting)
        return value

    def set(self, section, setting, value):
        self.config.set(section, setting, value)

    def add(self, section, setting, value):
        self.config[section] = {setting: value}

```

After adding the file picker, I realised that the user would have to select a directory every time they ran the program, so I thought it would be beneficial to create a way to store this info and any future settings (i.e. light mode vs dark mode). Created a class to handle the config. Decided to use .ini format, as it is built into the configparser standard library, and is very simple.

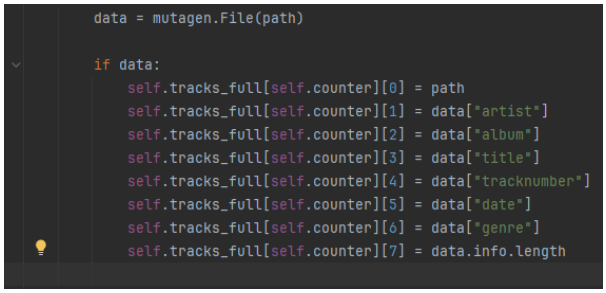
https://www.w3schools.io/ini-read-write-python/

I decided to add a GUI for the settings class, which will be shown in the tab bar, and move the file picker, and directory scanning methods to this class. Also created class for storing data.

Class for storing metadata. Originally, the file picker was part of the library class, and stored data would be stored there, but when I realised that multiple classes would all require this data, having it part of the library would be inefficient.

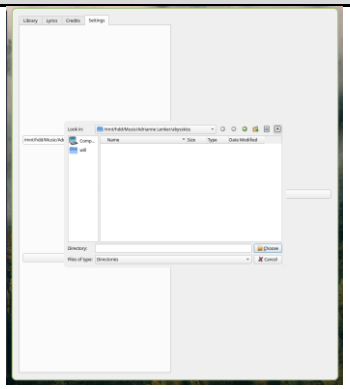
```
self.tracks_full = [[]]
```

Trying to decide how to store metadata. Some options were:

	• Simple 2D Array – simplest, could be inefficient? • Nested dictionaries – still simple, more readable than a 2D array • Separate objects for each track, album or artist – more complex, unclear benefits • Database – most complex but also has the most benefits. A database would be the most efficient, as I would not need to re-run the algorithms over the entire directory every time the program is run. Also allows me to track things that I wouldn't want to be wiped when the program is closed e.g. play count In the end, I decided to stick with a 2D array and not to use a database as while it had the most benefits, I am aware of scope creep, and it would take too long/be too difficult to implement. Using a 2D list should work perfectly fine and is much easier to implement. However, it could make running the program a lot slower for larger libraries. This is because the algorithms to scan a directory and read metadata will need to be run every time the program is run. This would be fine if the user only has a few albums that need to be read, but if their library contains thousands of albums this will likely be very slow.
 `data = mutagen.File(path)` `if data:` `  self.tracks_full[self.counter][0] = path` `  self.tracks_full[self.counter][1] = data["artist"]` `  self.tracks_full[self.counter][2] = data["album"]` `  self.tracks_full[self.counter][3] = data["title"]` `  self.tracks_full[self.counter][4] = data["tracknumber"]` `  self.tracks_full[self.counter][5] = data["date"]` `  self.tracks_full[self.counter][6] = data["genre"]` `  self.tracks_full[self.counter][7] = data.info.length`	Using the Mutagen library to extract metadata from file and add it to the 2D list. mutagen.File(path) automatically scans the file to work out the filetype (.flac, .mp3 etc.) If it cannot find type, or not a supported format, returns None. The if statement ensures that only files supported by Mutagen get their metadata read, otherwise it does nothing

`` if data: temp_list = [0] * 8 temp_list[0] = path temp_list[1] = data["artist"] temp_list[2] = data["album"] temp_list[3] = data["title"] temp_list[4] = data["tracknumber"] temp_list[5] = data["date"] temp_list[6] = data["genre"] temp_list[7] = data.info.length self.tracks_full.append(temp_list) ```	However, there was a problem initially with the 2D list. I thought that the .append function wouldn't work with a 2D list, so I was using a counter variable which introduced errors with the list size. I realised that I could use the .append method, by appending an entire list, and this was a lot easier. https://mutagen.readthedocs.io/en/latest/api/flac.html
`` def get_metadata(self, index, data): return self.tracks_full[index][data] ```	Simple getter method to return chosen metadata. Parameter 'index' will be used to select track, and 'data' for the specific data (i.e. path, artist etc.)

Test

Test Description		Evidence
Test file dialog window – Ensure it opens correctly and allows for picking a directory		 Opens correctly
Select folder, ensure metadata is correct and complete		 * printing the contents of tracks_full shows that metadata is correct and complete

Review
Completed the necessary functions for scanning library and extracting metadata. Everything works correctly, with the only major issues being wanting to add more features/more complex features (e.g. a database instead of a simple list), that I decided against due to time constraints. For a large library, this could make running the program extremely slow, especially as Python is a high-level interpreted language so it will naturally run slower.