Stage 1 – Basic GUI

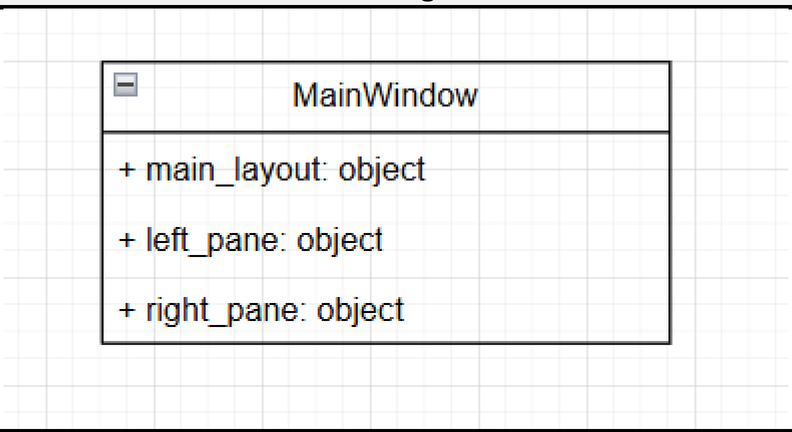
This stage will mainly include getting the basic GUI layout ready and setting it up to make sure it is easy to add new sections.

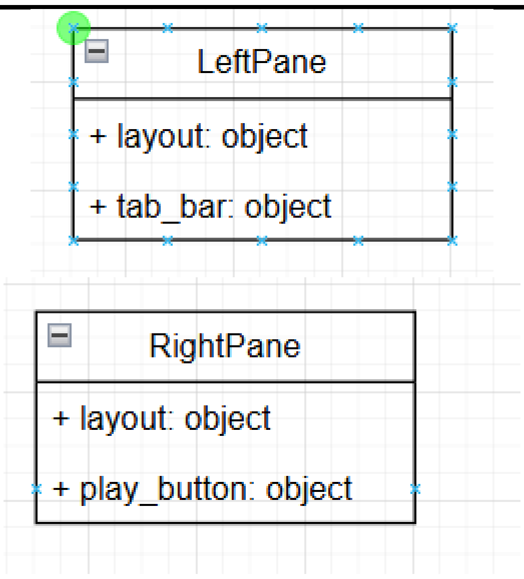
Design

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.

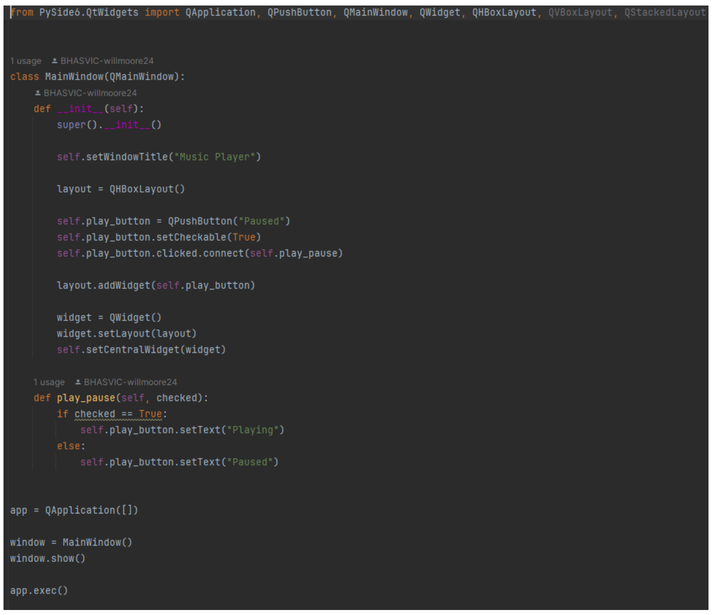
Data Dictionary		
Variable	**Type**	**Explanation/Justification**
No important variables at this stage as at this point I am mostly only working with objects included in the GUI library, PySide6.		

Class Diagrams



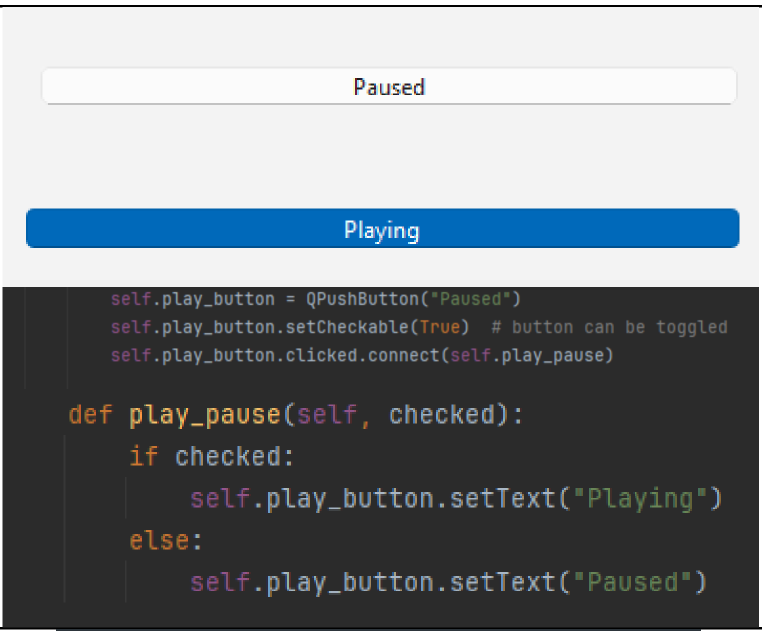


Develop

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[1] to find the necessary widgets.

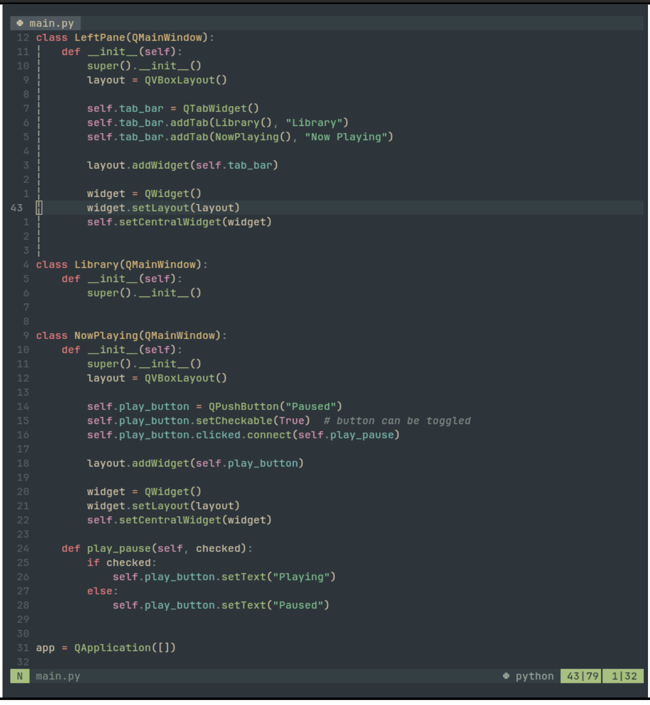
```
from PySide6.QtWidgets import QApplication, QPushButton, QMainWindow, QWidget, QHBoxLayout, QVBoxLayout, QStackedLayout


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()
```

[1] https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/index.html



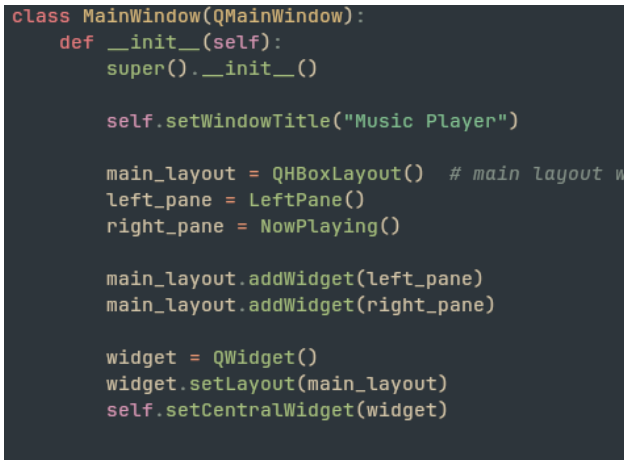
```python
        self.play_button = QPushButton("Paused")
        self.play_button.setCheckable(True)  # button can be toggled
        self.play_button.clicked.connect(self.play_pause)

    def play_pause(self, checked):
        if checked:
            self.play_button.setText("Playing")
        else:
            self.play_button.setText("Paused")
```

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 be beneficial to have the library coded first, so the rest of the now playing section will come later.



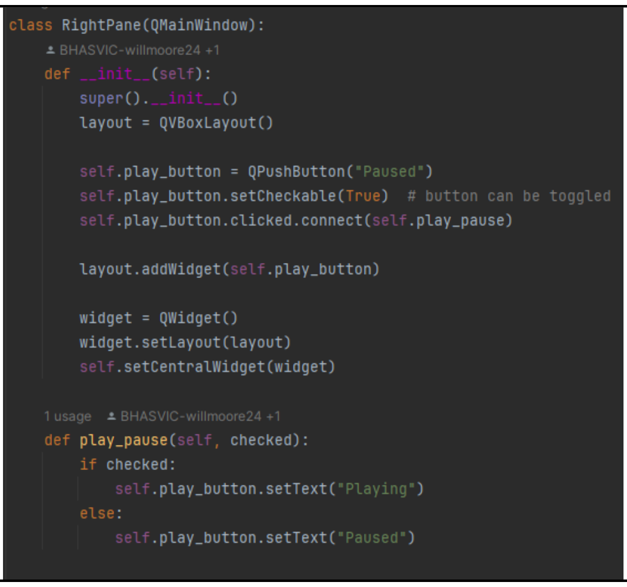
```python
class LeftPane(QMainWindow):
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        self.tab_bar = QTabWidget()
        self.tab_bar.addTab(Library(), "Library")
        self.tab_bar.addTab(NowPlaying(), "Now Playing")

        layout.addWidget(self.tab_bar)

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


class Library(QMainWindow):
    def __init__(self):
        super().__init__()


class NowPlaying(QMainWindow):
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        self.play_button = QPushButton("Paused")
        self.play_button.setCheckable(True)  # button can be toggled
        self.play_button.clicked.connect(self.play_pause)

        layout.addWidget(self.play_button)

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

    def play_pause(self, checked):
        if checked:
            self.play_button.setText("Playing")
        else:
            self.play_button.setText("Paused")


app = QApplication([])

N  main.py                                    python  43|79  1|32
```

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.



```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Music Player")

        main_layout = QHBoxLayout()  # main layout w
        left_pane = LeftPane()
        right_pane = NowPlaying()

        main_layout.addWidget(left_pane)
        main_layout.addWidget(right_pane)

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

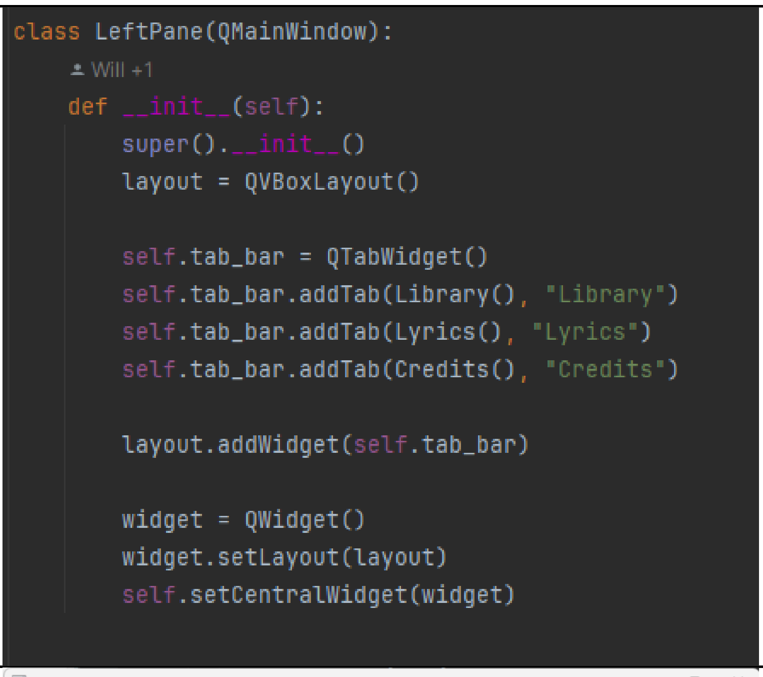
Using a horizontal layout (QHBoxLayout) allows for the two panes, where each individual pane will contain nested widgets.[2]



```
class RightPane(QMainWindow):
    ± BHASVIC-willmoore24 +1
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        self.play_button = QPushButton("Paused")
        self.play_button.setCheckable(True)  # button can be toggled
        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 +1
    def play_pause(self, checked):
        if checked:
            self.play_button.setText("Playing")
        else:
            self.play_button.setText("Paused")
```

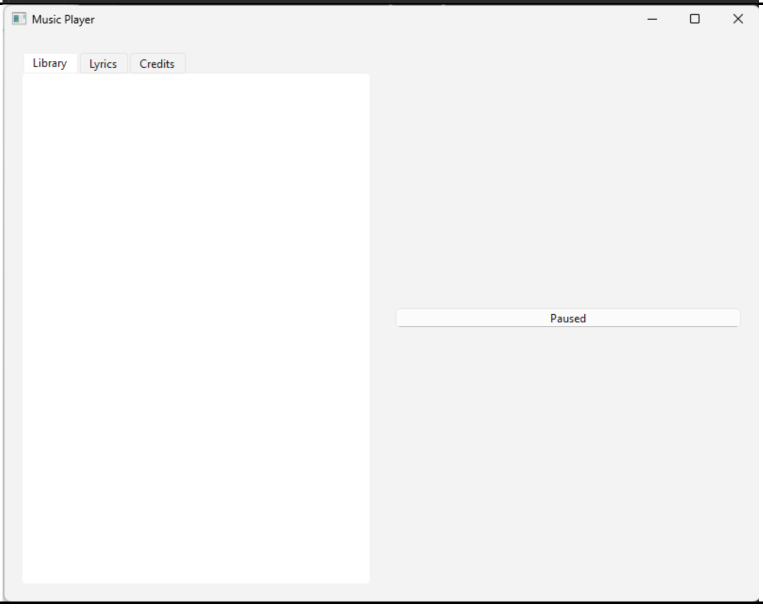
RightPane class for showing currently playing. At the moment, this section will not contain too many different widgets so should be easy to fit all into this class, but in the future I may move widgets such as 'play_button' into their own class.

[2] https://www.pythonguis.com/tutorials/pyside6-layouts/



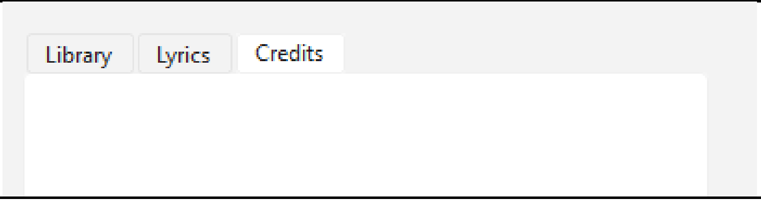
```python
class LeftPane(QMainWindow):
    ≛ Will +1
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        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)
```

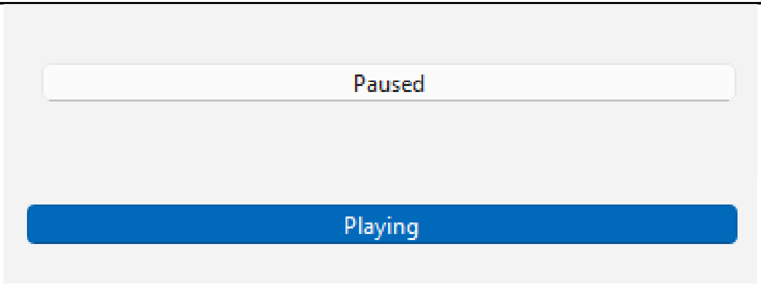
For the left pane I have used the tab bar included in Qt which allows the user to switch between widgets using tabs at the top of the window. This is easy to add new widgets, as I just write the section as a new class and add the tab with self.tab_bar.addTab(<object>, name).



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 – Library Management

The aim of this stage is to first write the algorithms to scan a selected folder for music files and record what albums, artists and tracks the user has on their system.

I will then create the library section to display the names of tracks stored, grouped into album and artist.

I will also add the ability to sort the library based on specific criteria i.e. alphabetically, release year, rating, for example.

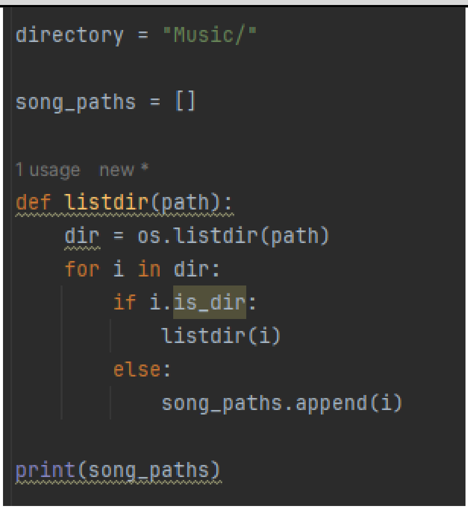
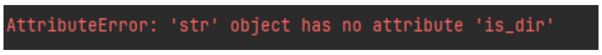
Design

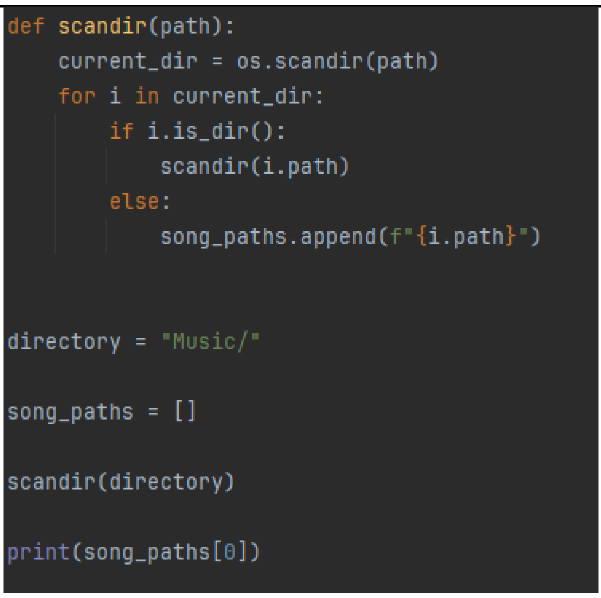
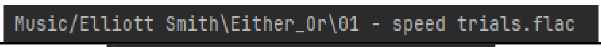
Algorithms	
Algorithm	Explanation

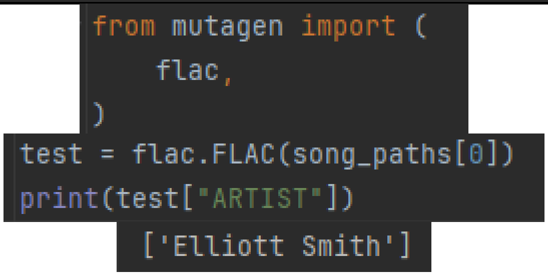
Data Dictionary & Class Diagrams		
Variable	Type	Explanation

Develop

Screenshot	Explanation/Justification

 `directory = "Music/"`
 	
`song_paths = []`
 	
`1 usage   new *`
`def listdir(path):`
`    dir = os.listdir(path)`
`    for i in dir:`
`        if i.is_dir:`
`            listdir(i)`
`        else:`
`            song_paths.append(i)`
 	
`print(song_paths)`
 `AttributeError: 'str' object has no attribute 'is_dir'`
	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.

 `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.

 `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.

Test		
Test Description		Evidence

Review