Object Oriented Programming Final Assignment Report

R1: The application should contain all the basic functionality

Note that the application has been adapted to now have a mid-panel, which displays all buttons and sliders and controls most of the audio functionalities for both the right and left sides of the audio players.

R1A: can load audio files into audio players
This is handled by PlaylistComponent.cpp and DJAudioPlayer.cpp. The user can either click on the "Add To Library" button within the PlaylistComponent object to add one music file at a time or drag a collection of music files from another file to the playlist. The PlaylistComponent object holds a pointer that points to the relevant DeckGUI object. This pointer gets the DeckGuI object to load the relevant music file onto the appropriate DJAudioPlayer using the audio player's loadURL method.

R1B: can play two or more tracks
The application contains two audio players, which are two DJAudioPlayer objects created from the DJAudioPlayer class. These two players are player L on the left hand side and player R on the right hand side. Each of these players are pointed to by their respective "play" buttons created in the MidPanel interface (or object). This allows the buttons in the MidPanel to directly control the start() method in the DJAudioPlayer object. Because these two players are two different objects with their own separate "play" buttons, they are able to play music simultaneously or individually and are also independent from each other in terms of their functionalities.

R1C: can mix the tracks by varying each of their volumes
Similar to R1B, the two volume sliders in MidPanel object are each pointing to their respective DJAudioPlayer object. This allows the volume sliders access to the setGain() method in the DJAudioPlayer objects. On top of this, there is also a crossfader slider in MidPanel that controls both player L and R's volume sliders, allowing smooth mixing and fading of the tracks. This is explained in more detail in R2.

R1D: can speed up and slow down
Similar to R1C and R1B, the MidPanel contains "speed" buttons (explained in more detail in R2) that are each pointing at their respective DJAudioPlayer object. By using the pointers, these buttons can access the setSpeed() method in DJAudioPlayer.cpp. This gives each of the player object access to setSpeed(), which allows them to slow down or speed up their respective tracks.

R2: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/start

R2A: Component has custom graphics implemented in a paint function.
As mentioned earlier, a MidPanel component has been created to hold all buttons and sliders that control the two audio players. Note that the DJAudioPlayer component controls any the audio functionalities, while the DeckGUI is involved in the slow-stop control (The two black jog-wheels) and the waveform. Let's go into more detail into each of these button and slider types:

Play, Stop and Loop Buttons
The Play, Stop and Loop buttons are created by the CustomTextButton class. There are two constructors for this class; one takes in two button names as arguments and another takes in only one button name.

The Play button works in a way that, once it is clicked, its name changes from "Play" to "Pause" (this is similar to many audio player software interfaces). And if the button is clicked again, it changes from "Pause" to "Play". Therefore, it makes sense for the Play button to take in two button names. On the other hand, the Stop and Loop buttons' names stay the same regardless of how many times they are clicked, therefore they only take in one button name.

The Stop button stops the track and returns the play head back to the beginning of the track. The Loop button sets the track on repeat, so that when the track ends, the music goes back to the beginning and replay.

These three buttons take in a pointer to the DJAudioPlayer so that they can directly control the functionalities of their respective audio players. This is much neater than, say, having the MidPanel handle all of the functionalities as this would require messier code within MidPanel. Hence, it is in my opinion that passing in the pointers into these button objects is necessary.

Importantly, CustomTextButton.cpp also takes in the ButtonType data type as an argument. This is defined in enum class (written in the top section of CustomTextButton.h), specifying the type of the button to be created, namely, "play", "stop" or "repeat" (just to clarify that "repeat" refers to the Loop button type).

The **Paint()** function in CustomTextButton draws the labels of the buttons according to the ButtonType and, in the case of the Play button, how many times the button has been clicked. Once clicked, the whole button, including the text, turns from white to green. And if it is clicked again, it changes from green back to white. There is an exception for the stop button, which only turns green for as long as the mouse is pressed. Once the mouse is released, it turns back to white. This is because it does not make sense for the stop button to stay green when it is clicked once.

Speed Button

As the speed button has a different shape and quite distinct functionalities and behaviour compared to the Play, Stop and Loop buttons, it needs to have its own separate class, which is SpeedButton.cpp.

The Speed button is round in shape and sets the speed of the track in three levels, namely 1x (normal speed), 2x (twice the normal speed) and 3x (three times the normal speed). To get to each of these speed levels, the user only needs to keep clicking until the desired level is reached. Note that the text and button colour changes at each different level, white at 1x, green at 2x and red at 3x.

The code in SpeedButton's **Paint()** method ensures that the behaviour is correctly drawn. This involves keeping track of the number of times the user has clicked the button.

The SpeedButton class also takes in a pointer to the DJAudioPlayer as an argument, allowing it to directly control the speed of the track.

Cue Button

This is the most complex button in terms of functionality. Since the cue button's functionality differs greatly from the other buttons, it also needs to have its own separate class. This class also takes in a DJAudioPlayer pointer to control the respective player.

The Cue button allows the user to save a track's current position while the track is playing. Once the position of the track is saved, the button then turns green and its label changes from its original name to "saved" to indicate that the desired position has been saved. Once the button is green and says "saved", and if the user clicks on it while the track is at another position, the track will go back to the position that has been saved in the cue. This is a common functionality on a professional DJ device. However, if the user no longer wants this particular position to be saved in that cue button, it can free the memory and reset the cue button by **holding the CTRL key** while **left-clicking** on the relevant cue button. This will return the cue button to its original state, meaning it will change from green to white and its label will turn from "saved" to its original label.

The logic of the code is mainly written in the **mouseDown()** method in CueButton.cpp and the changes in appearance and text according to how the button is clicked. This is handled by its **paint()** method.

In the mouseDown() method, to accurately record the state of the button, four boolean properties are required, namely playCueCall, saveCueCall, buttonOn, as well as a boolean method isCtrlDown(). The **combination** of the states of these four boolean values determine what action the cue button should execute when the user clicks on the button:

When the playCueCall is true, it means the saved cue position is being used to set the player position. When saveCueCall is true, it means that the current position of the player will be saved by the cue button. if buttonOn is true, it means the button is currently green and says "saved" on the label. If ModifierKeys::currentModifiers.isCtrlDown() is true, it means the CTRL is being held down.

Volume Sliders
The appearance of the volume sliders have been significantly altered from the default Juce sliders. To do this, CompLookAndFeel.cpp has been created and this class inherits juce::LookAndFeel_V4, which allows custom graphics to be drawn. The custom graphics are manually drawn in the drawLinearSlider() method.

In contrast to the aforementioned custom buttons, the volume sliders' functionalities are handled in the sliderValueChanged method in MidPanel.cpp.

Crossfader Slider
The crossfader slider allows the user to cross fade from one track to another. The crossfader slider also moves the respective positions of the two volume sliders, giving a pleasing visual effect.
The custom graphics of the crossfader has also been manually drawn with the help of CompLookAndFeel.cpp and its functionality is handled in the sliderValueChanged method in MidPanel.cpp.

R2B: Component enables the user to control the playback of a deck somehow
This has already been clearly explained in R2A. Please see the functionalities of Play, Stop and Look buttons in R2A.

R3: Implementation of a music library component which allows the user to manage their music library

R3A: Component allows the user to add files to their library
The user has two ways to add files to the library. One is to drag a collection of music files into the play list area, or add one file at a time using the "Add To Library" button. Note that upon launching the application, if there is nothing saved in the playlist (if the playlist is empty), placeholder text saying "Empty track: Drag files or use Add button to load songs" will be displayed instead.

This functionality is handled in PlaylistComponent.cpp and with the help of two arrays: trackTitles (storing the std::string names of the tracks) and trackFiles(storing the file paths of the tracks). Much of this is implemented in the filesDropped() method (drag files functionality) and the buttonClicked() method (for the Add To Library button).

Note that the playlist is able to cope with **duplicated** files being added; the code checks the playlist each time a file is added to the playlist. If the file is already on the playlist, then that file will not be added.

Also note that apart from adding, the user an also remove the tracks from the playlist by clicking on the X buttons.

R3B: Component parses and displays meta data such as filename and song length
Once music files are added to the playlist (PlaylistComponent object), the track title length of the track and file size are displayed for each track.

Much of this functionality is handled in the paintCell() method in PlaylistComponent.cpp. The file size is retrieved using the static function File::descriptionOfSizeInBytes(), track length is retrieved using a pointer that points to the getLengthSeconds() method in DJAudioPlayer and the track title is obtained using the File method getFileNameWithoutExtension().

R3C: Component allows the user to search for files
The user can use the search bar on the PlaylistComponent interface to search for any song on the playlist.

This is implemented within PlaylistComponent.cpp with the help of two additional arrays: store_trackTitles (storing the “backup” std::string names of the tracks) and store_trackFiles(storing the “backup” file paths of the tracks). Note that these two arrays are in addition to the aforementioned trackFiles and trackTitles arrays. There are a total of four arrays working in conjunction with each other to carry out the search and filter functionality. While it is entirely possible to only use two arrays rather than four, it is in my opinion that having four clearly laid out arrays allows coders to visualise the workings more clearly, although care has to be taken in the sense that, when deleting the trackTitles array, the coder needs to remember to delete the trackFiles array at the same time, too. Another important factor is, this playlist contains placeholder texts at the start; having four arrays makes dealing with the situation where placeholder texts are present easier.

Much of the search functionality is carried out in the textEditorTextChanged() method in PlaylistComponent.cpp. If the user types in something in the search bar, the code starts to save that std::string into a variable, then makes a copy of the contents in the two arrays trackFiles and trackTitles and store them into store_trackFiles and store_trackTitles, respectively. Then, the code proceeds to delete all contents in the original trackFiles and trackTitles arrays. All relevant search results are pushed into these two original arrays, which will then display the results onto the playlist. The search functionality is very dynamic in the sense that, if the searched word is altered, the search results automatically and immediately change to correspond to the different word being searched.

Once the user decides to reset the search bar and remove the searched word, the code is able to detect this and restore all the original track list onto the playlist. It does this by deleting all the new contents in trackFiles and trackTitles, then

pushing back all of the contents in store_trackFiles and trackTitles back into trackFiles and trackTitles. This only happens when the code detects that the search bar no longer contains a word or character.

R3D: Component allows the user to load files from the library into a deck
This is implemented in PlaylistComponent.cpp's buttonClicked method. The playlist has two load buttons, namely left (for loading onto the left audio player) and right (for loading onto the right audio player).

Because PlaylistComponent takes in pointers to DeckGUIs that have access to both the left and right audio players, it has access to DeckGUI's loadFile() method, which will help it to load the relevant files onto the appropriate audio player and display the relevant waveform onto the DeckGUI.

R3E: The music library persists so that it is restored when the user exits then restarts the application
Important note to marker: If you are using XCode on Macbook, for this to work properly, you must ensure that the code editor is working in the right folder. You have to go to Product, then select Scheme then Edit Scheme. Then tick "Use Custom Working Directory" and select the Source folder path (the folder that stores all the .cpp and .h files, as well as the .csv file).

To implement this functionality, a SaveList.cpp class is created. The object of this class is created and placed within the PlaylistComponent object. This is how it works: when the user closes the application, the PlaylistComponent's destructor function is called. Within this function, the SaveList object is called to save all of the contents of the current trackFiles array then outputs them in std::string onto a csv file, which is saved in the Source folder.

When the application is reopened, within the PlaylistComponent constructor function, the code checks the .csv file to see if it is empty. If the csv file is not empty, it means the playlist was not empty when it was previously closed. Therefore, the code loads the contents of the .csv file onto the trackTitles and trackFiles, store_trackTitles and store_trackFiles, which will display all the previously saved track files onto the playlist and ensure that the search function continues to work properly. However, if the .csv file is empty, then the code will just display the placeholder text indicating that the playlist is empty.

R4: Implementation of a complete custom GUI

R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls

Both the left and right DeckGUIs no longer contain all the functional buttons and volume sliders. As mentioned earlier, all the previously available buttons and sliders, as well as extra controls like the cue buttons and crossfader, have been moved to the mid panel. Rather, the DeckGUI now holds a large jog-wheel that implements a slow-stop effect. This slow-stop functionality works by allowing the

user to drag the white needle of the jogwheel and spin it **anti-clockwise** to make the track stop in a slow-motion manner. This is a common technique used by many professional DJs and is present in many pop songs, including Britney Spears' You Drive Me Crazy (https://www.youtube.com/watch?v=Q4VK9_CfOLQ **see the 2:10 minute mark how this is done**)
Much of the jogwheel's functionality is implemented in DeckGUI::mouseDrag(). The code includes turning the angle of the needle to determine how slow the speed of the track should be.

R4B: GUI Layout includes the custom Component from R2

The MainComponent object now contains both left and right DeckGUIs (includes the jogwheels and waveform displays), the MidPanel object (contains most of the buttons and sliders).

R4C: GUI Layout includes the music library component from R3

The MainComponent now contains the music library, which includes the "Add To Library" button, the track list and the relevant metadata alongside other relevant buttons and also the search bar for the user to search for music in the library. The entire application now looks very different from what was given in the lecture.