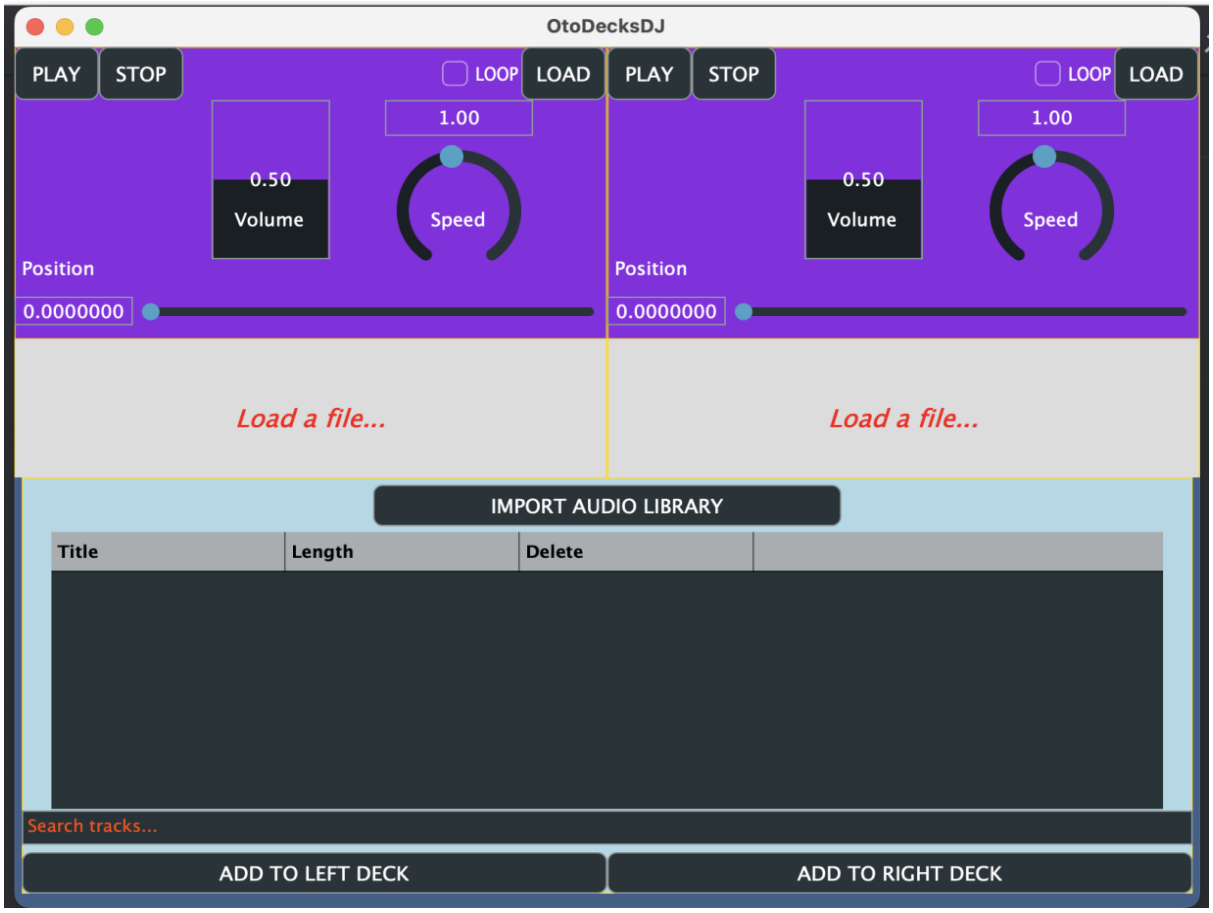
Object-oriented programming report: -

I made a simple DJ application called otodecks that can be used to edit and remix music recordings that have been imported into the application. It includes a playlist of music files that can be added to the user interface's deck and edited. I made this application using XCode (MacOS).

Here Is the output of my DJ application: -



Project requirements: -

R1A: Enables audio players to load audio files:-

In my DeckGUI.cpp and PlaylistComponent.cpp, which contains the code excerpt below. There are two ways to load music in my otodecks project. One is by pressing the load button and selecting the type of music the user wants.

```
117     else if (button == &loadButton) // button to load new audio file
118     {
119         DBG("Load button was clicked ");
120         FileChooser chooser{ "Select a file" };
121         if (chooser.browseForFileToOpen())
122         {
123             loadFile(URL{ chooser.getResult() });
124         }
125     }
126 }
```

The music file in the otodecks can be controlled by the play, stop and loop buttons and be adjusted by the volume, speed and position sliders. The user can also now drag and drop music files from their audio folders into the DeckGUI due to a function I had enabled.

```
bool DeckGUI::isInterestedInFileDrag(const StringArray& files)
{
    std::cout << "DeckGUI::isInterestedInFileDrag" << std::endl;
    return true;
}

void DeckGUI::filesDropped(const StringArray& files, int, int y)
{
    if (files.size() == 1)
    {
        player->loadURL(URL{File{files[0]} }); // player calls loadURL function from DJAudioPlayer
        waveformDisplay.loadURL(URL{ fChooser.getResult() });
    }
}
```

Another way of loading music to the otodecks is by importing the audio library to a playlist. After adding the music to the playlist, the user can choose where to load and play the file(left or right deck). The "Add to Left Deck" or "Add to Right Deck" buttons will load the songs to the audio player above.

```
141
142 void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
143 {
144     int selectedRow{ library.getSelectedRow() };
145     if (selectedRow != -1)
146     {
147         DBG("Adding: " << tracks[selectedRow].title << " to Player");
148         deckGUI->loadFile(tracks[selectedRow].URL);
149     }
150     else
151     {
152         AlertWindow::showMessageBox(AlertWindow::AlertIconType::InfoIcon,
153             "Add to Deck Information:",
154             "Please select a track to add to deck",
155             "OK",
156             nullptr
157         );
158     }
159 }
```

```
161 void PlaylistComponent::importToLibrary()
162 {
163     //initialize file chooser
164     FileChooser chooser{ "Select files" };
165     if (chooser.browseForMultipleFilesToOpen())
166     {
167         for (const File& file : chooser.getResults())
168         {
169             String fileNameWithoutExtension{ file.getFileNameWithoutExtension() };
170             if (!isInTracks(fileNameWithoutExtension)) // if not already loaded
171             {
172                 Track newTrack{ file };
173                 URL audioURL{ file };
174                 newTrack.length = getLength(audioURL);
175                 tracks.push_back(newTrack);
176                 DBG("loaded file: " << newTrack.title);
177             }
178             else // display info message
179             {
180                 AlertWindow::showMessageBox(AlertWindow::AlertIconType::InfoIcon,
181                     "Load information:",
182                     fileNameWithoutExtension + " already loaded",
183                     "OK",
184                     nullptr
185                 );
186             }
187         }
188     }
189 }
```

As shown in the screenshot below, a function in my DJAudioPlayer.cpp allows the URL of the audio track file to be loaded, played, and read.

```
162  void DeckGUI::loadFile(URL audioURL)
163  {
164      DBG("DeckGUI::loadFile called");
165      player->loadURL(audioURL);
166      waveformDisplay.loadURL(audioURL);
167  }
```

R1B: is able to play two or more tracks: -

By placing the audio track files on the right and left decks of the audio player, the user can play two audio tracks concurrently. The track files are queued after being loaded into either the left or right decks, and the user can then select "Play" to begin playing the audio tracks.

```
PlaylistComponent playlistComponent { &deckGUI1, &deckGUI2, &playerForParsingMetaData };
```

```
8  class DeckGUI  : public Component,
9                   public Button::Listener,
10                  public Slider::Listener,
11                  public FileDragAndDropTarget,
12                  public Timer
13
14 {
15 public:
16     DeckGUI(DJAudioPlayer* player, AudioFormatManager& formatManagerToUse, AudioThumbnailCache& cacheToUse);
17     // DeckGUI gets DJ audio player via the constructor, and use assignment list
18     ~DeckGUI() override;
19
```

R1C: can adjust each of the song's volumes to mix the tracks.: -

For volume control, I've implemented a linear bar vertical slider, to adjust the audio volume while the audio file is playing.A slider for visualizing the volume slider is present in my DeckGUI.cpp file, and it contains the location of the volume slider in an "if" condition. My volume slider's gain is set using "if" conditions.

```
void DeckGUI::sliderValueChanged(Slider* slider)
{
    if (slider == &volumeSlider)
    {
        DBG("Volume slider moved: " << slider->getValue());
        player->setGain(slider->getValue()); // player calls setGain function from DJAudioPlayer
    }
```

I have added slider listeners and customizations for my sliders in DeckGUI.cpp, such as the volume slider

```
32      volumeSlider.addListener(this);
33
34      //volumeSlider customisation
35      volumeSlider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
36      volumeSlider.setRange(0.0, 1.0);
37      volumeSlider.setValue(0.5); //default volume half of max vol
38      volumeSlider.setNumDecimalPlacesToDisplay(2);
39      volumeSlider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
40      volumeSlider.setTextBoxStyle(Slider::TextBoxAbove, false, 100,25);
41      addAndMakeVisible(volumeLabel);
42      volumeLabel.setText("Volume", dontSendNotification);
43      volumeLabel.attachToComponent(&volumeSlider, false);
44      volumeLabel.setJustificationType(Justification::centred);
```

R1D: Can change the speed of the tracks

for track speed, I've implemented a rotating slider with labels that are displayed in the DeckGUI class layout.

```
  speedSlider.setSliderStyle(Slider::Rotary);
  speedSlider.addListener(this); // The object on the GUI must be informed by the entity that wants to receive
      events that it wants to register for them.
  speedSlider.setRange(0.1, 2.0);
  speedSlider.setValue(1.0);
  speedSlider.setNumDecimalPlacesToDisplay(2);
  speedSlider.setTextBoxStyle(Slider::TextBoxAbove, false, 100, 25);
  addAndMakeVisible(speedSliderLabel);
  speedSliderLabel.setText("Speed", dontSendNotification);
  speedSliderLabel.attachToComponent(&speedSlider, false);
```

R2A: The component's paint function uses customized graphics

By moving the rows and columns, I was able to reposition the playlist elements as well as the user controls for my application interface. The following code snippet is from my DeckGUI.cpp file, and its resized positions in the user's customized controls. I used a linear bar slider for the volume and a rotatory slider for the speed.

```
void DeckGUI::resized()
{
    // This method is where you should set the bounds of any child components that your component contains..

    double rowH = getHeight() / 8;
    playButton.setBounds(0, 0, getWidth()/7, rowH);
    stopButton.setBounds(getWidth() / 7, 0, getWidth() / 7, rowH);
    loopButton.setBounds((getWidth() / 7)*5, 0, getWidth() / 7, rowH);
    loadButton.setBounds((getWidth() / 7)*6, 0, getWidth()/7, rowH);

    volumeSlider.setBounds(getWidth() / 3, rowH, getWidth()/5, rowH*3);
    speedSlider.setBounds(getWidth()/2, rowH, getWidth()/2, rowH*3.5);

    volumeLabel.setCentreRelative(0.43f, 0.4f);
    speedSliderLabel.setCentreRelative(0.94f, 0.4f);
    positionSlider.setBounds(0, rowH * 4.5, getWidth(), rowH);
    waveformDisplay.setBounds(0, rowH * 5.5, getWidth(), rowH*3);
}
```

The locations of the search bar and the playlist containing the audio tracks that are currently loaded into the interface and are awaiting loading into the DJ audio player above have been customized in my PlaylistComponent.cpp file.

```
    searchField.setTextToShowWhenEmpty("Search tracks...", juce::Colours::orangered);
    searchField.onReturnKey = [this] { searchLibrary(searchField.getText()); };

    // setup table and load library from file
    library.getHeader().addColumn("Title", 1, 1);
    library.getHeader().addColumn("Length", 2, 1);
    library.getHeader().addColumn("Delete", 3, 1);
    library.setModel(this);
    loadLibrary();
```

R2B: Component gives the user some sort of playback management over a deck.

In my DeckGUI, I've also included a position slider that enables the playback of the audio files.cpp

```
    positionSlider.addListener(this); // The entity that wants to receive events must tell the object on the GUI
        that it wants to register for them.
    positionSlider.setRange(0.0, 1.0);
    addAndMakeVisible(positionSliderLabel);
    positionSliderLabel.setText("Position", dontSendNotification);
    positionSliderLabel.attachToComponent(&positionSlider, false);
```

The conditions under which the position slider for audio playback should be applied are listed in my DJAudioPlayer.cpp file.

```
void DJAudioPlayer::setPosition(double posInSecs)
{
    transportSource.setPosition(posInSecs);
}

void DJAudioPlayer::setPositionRelative(double pos) // between 0 and 1; i.e. 0 - 100% of any length audio file
{
    if (pos < 0 || pos > 1.0)
    {
        std::cout<<"DJAudioPlayer::setPositionRelative pos should be between 0 and 1"<< std::endl;
    }
    else
    {
        double posInSecs = transportSource.getLengthInSeconds() * pos; // pos of 0 - 1 applied to length of audio
            file in seconds
        setPosition(posInSecs);
    }
}
```

R3A: Through the playlist component, the component enables users to upload files to their libraries

By reading the file's URL and pathway, the user can upload audio track files to the Otodecks playlist library using my PlaylistComponent.cpp code.

```
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow{ library.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Adding: " << tracks[selectedRow].title << " to Player");
        deckGUI->loadFile(tracks[selectedRow].URL);
    }
```

R3B: The component analyzes and shows metadata, including track length and filename

The FilenameWithoutExtension function will cause the file name of the loaded file to be presented.

```
161 void PlaylistComponent::importToLibrary()
162 {
163     //initialize file chooser
164     FileChooser chooser{ "Select files" };
165     if (chooser.browseForMultipleFilesToOpen())
166     {
167         for (const File& file : chooser.getResults())
168         {
169             String fileNameWithoutExtension{ file.getFileNameWithoutExtension() };
170             if (!isInTracks(fileNameWithoutExtension)) // if not already loaded
171             {
172                 Track newTrack{ file };
173                 URL audioURL{ file };
174                 newTrack.length = getLength(audioURL);
175                 tracks.push_back(newTrack);
176                 DBG("loaded file: " << newTrack.title);
177             }
178             else // display info message
179             {
180                 AlertWindow::showMessageBox(AlertWindow::AlertIconType::InfoIcon,
181                     "Load information:",
182                     fileNameWithoutExtension + " already loaded",
183                     "OK",
184                     nullptr
185                 );
186             }
187         }
188     }
189 }
```

The call to a function called getLength allows us to reach the load URL and getLengthInSeconds functions in the DJAudioplayer by using the pointer to playerforparsingmetadata.

```
201 String PlaylistComponent::getLength(URL audioURL)
202 {
203     playerForParsingMetaData->loadURL(audioURL);
204     double seconds{ playerForParsingMetaData->getLengthInSeconds() };
205     String minutes{ secondsToMinutes(seconds) };
206     return minutes;
207 }
```

As a result, we can measure the length of the music in seconds and use the seconds to minutes function to convert the outcome to minutes.

```
209  String PlaylistComponent::secondsToMinutes(double seconds)
210  {
211      //find seconds and minutes and make into string
212      int secondsRounded{ int(std::round(seconds)) };
213      String min{ std::to_string(secondsRounded / 60) };
214      String sec{ std::to_string(secondsRounded % 60) };
215
216      if (sec.length() < 2) // if seconds is 1 digit or less
217      {
218          //add '0' to seconds until seconds is length 2
219          sec = sec.paddedLeft('0', 2);
220      }
221      return String{ min + ":" + sec };
222  }
```

Finally, the Isintrack method iterates through the vector from start to finish to determine if the same name appears anywhere in the vector.

```
bool PlaylistComponent::isInTracks(String fileNameWithoutExtension)
{
    return (std::find(tracks.begin(), tracks.end(), fileNameWithoutExtension) != tracks.end());
}
```

R3C: Component enables the user to search for files

I created a listener for the search box and labeled it in my PlaylistComponent.cpp file to enable users to look for specific audio tracks among the audio tracks loaded in the playlist.

```
searchField.addListener(this);
addToPlayer1Button.addListener(this);
addToPlayer2Button.addListener(this);

searchField.setTextToShowWhenEmpty("Search tracks...", juce::Colours::orangered);
searchField.onReturnKey = [this] { searchLibrary(searchField.getText()); };
```

The search library function along with the search text string searches through files that could be in the playlist by their titles.

```
void PlaylistComponent::searchLibrary(String searchText)
{
    DBG("Searching titles for: " << searchText);
    if (searchText != "")
    {
        int rowNumber = whereInTracks(searchText);
        library.selectRow(rowNumber);
    }
    else
    {
        library.deselectAllRows();
    }
}

int PlaylistComponent::whereInTracks(String searchText)
{
    // finds index where track title contains searchText
    auto it = find_if(tracks.begin(), tracks.end(),
        [&searchText](const Track& obj) {return obj.title.contains(searchText); });
    int i = -1;

    if (it != tracks.end())
    {
        i = std::distance(tracks.begin(), it);
    }

    return i;
}
```

R3D: enabling the user to add audio tracks from a playlist to a deck

I made a loadinplayer function that will be executed when the user presses on the button to load tracks to either the left audio player or the right audio player in order to load the audio track files from the playlist to the deck audio player.

```
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow{ library.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Adding: " << tracks[selectedRow].title << " to Player");
        deckGUI->loadFile(tracks[selectedRow].URL);
    }
    else
    {
        AlertWindow::showMessageBox(AlertWindow::AlertIconType::InfoIcon,
            "Add to Deck Information:",
            "Please select a track to add to deck",
            "OK",
            nullptr
        );
    }
}
```

The addtoplayer1 button loads the track on DeckGUI1 and the addtoplayer2 button loads the track on DeckGUI 2.

```
void PlaylistComponent::buttonClicked(Button* button)
{
    if (button == &importButton)
    {
        DBG("Load button clicked");
        importToLibrary();
        library.updateContent();
    }
    else if (button == &addToPlayer1Button)
    {
        DBG("Add to Player 1 clicked");
        loadInPlayer(deckGUI1);
    }
    else if (button == &addToPlayer2Button)
    {
        DBG("Add to Player 2 clicked");
        loadInPlayer(deckGUI2);
    }
    else
    {
        int id = std::stoi(button->getComponentID().toStdString());
        DBG(tracks[id].title + " removed from Library");
        deleteFromTracks(id);
        library.updateContent();
    }
}
```

R3E: The music library is persistent, so restarting the program after quitting will bring it back

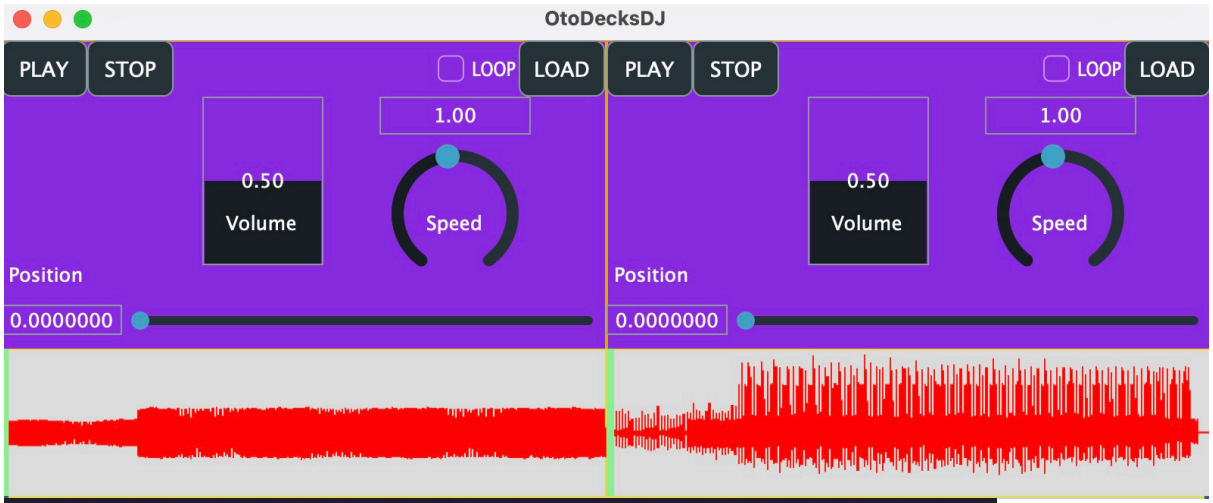
I used the load library and save library functions to save the playlist in the library.

```
253 void PlaylistComponent::saveLibrary()
254 {
255     // create .csv to save library
256     std::ofstream myLibrary("audioLibrary.csv");
257
258     // save library to file
259     for (Track& t : tracks)
260     {
261         myLibrary << t.file.getFullPathName() << "," << t.length << "\n";
262     }
263 }
264
265 void PlaylistComponent::loadLibrary()
266 {
267     // create input stream from saved library
268     std::ifstream myLibrary("audioLibrary.csv");
269     std::string filePath;
270     std::string length;
271
272     // Read data, line by line
273     if (myLibrary.is_open())
274     {
275         while (getline(myLibrary, filePath, ',')) {
276             File file{ filePath };
277             Track newTrack{ file };
278
279             getline(myLibrary, length);
280             newTrack.length = length;
281             tracks.push_back(newTrack);
282         }
283     }
284     myLibrary.close();
285 }
```

R4A and R4B: Customising the GUI layout

In contrast to the program taught in class, I fully customized the GUI layout for my Otodecks application while maintaining its simplicity and user-friendliness.

Each component—DeckGUI, PlaylistComponent, and DJAudioPlayer—has its own code that allows for customizations. The majority of the customizations involve "painting" and "resizing" the various components. In order to make it more noticeable to the user, the buttons and colors have also been altered.




```
void PlaylistComponent::paintRowBackground(Graphics& g, int rowNumber, int width, int height, bool rowIsSelected)
{
    if (rowIsSelected)
    {
        g.fillAll(juce::Colours::yellow);
    }
    else
    {
        g.fillAll(juce::Colours::plum);
    }
}

void PlaylistComponent::paintCell(Graphics& g, int rowNumber, int columnId, int width, int height, bool
    rowIsSelected)
{
    if (rowNumber < getNumRows())
        if (columnId == 1)
        {
            g.drawText(tracks[rowNumber].title, 2, 0, width - 4, height, Justification::centredLeft, true);
        }
        if (columnId == 2)
        {
            g.drawText(tracks[rowNumber].length, 2, 0, width - 4, height, Justification::centredLeft, true);
        }
}
```

I made rotatory sliders, and linear bar sliders and resized the buttons to customize the layout.

```
//volumeSlider customisation
volumeSlider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
volumeSlider.setRange(0.0, 1.0);
volumeSlider.setValue(0.5); //default volume half of max vol
volumeSlider.setNumDecimalPlacesToDisplay(2);
volumeSlider.setSliderStyle(Slider::SliderStyle::LinearBarVertical);
volumeSlider.setTextBoxStyle(Slider::TextBoxAbove, false, 100,25);
addAndMakeVisible(volumeLabel);
volumeLabel.setText("Volume", dontSendNotification);
volumeLabel.attachToComponent(&volumeSlider, false);
volumeLabel.setJustificationType(Justification::centred);


speedSlider.setSliderStyle(Slider::Rotary);
speedSlider.addListener(this); // The entity that wants to receive events must tell the object on the GUI
    that it wants to register for them.
speedSlider.setRange(0.1, 2.0);
speedSlider.setValue(1.0);
speedSlider.setNumDecimalPlacesToDisplay(2);
speedSlider.setTextBoxStyle(Slider::TextBoxAbove, false, 100, 25);
addAndMakeVisible(speedSliderLabel);
speedSliderLabel.setText("Speed", dontSendNotification);
speedSliderLabel.attachToComponent(&speedSlider, false);
```

I made extra controls like the loop function, in which if you check the loop box the song will keep on looping until the loop button is not ticked.

```
166
167 void DeckGUI::timerCallback() { //Every 100ms, the timer loops, checking to see if the "loop" button is turned on
        or off and updating the wave form display's current location.
168     if (std::to_string(loopButton.getToggleState()) == "1") { //If loop button is checked
169         if (player->getPositionRelative() >= 1) {// and the position more than 1, it means it is the end of the
                audio file
170             player->setPositionRelative(0); // Sets position back to the start of the audio file
171             player->start(); // Plays the audio file
172         }
173     }
174
175     if (std::to_string(loopButton.getToggleState()) == "0") { //If loop button is not checked
176         if (player->getPositionRelative() >= 1) { //and the position more than 1, it means it is the end of the
                audio file
177             player->setPositionRelative(0); //Sets position back to the start of the audio file
178             player->stop(); // Stops the audio file
179         }
180     }
181     waveformDisplay.setPositionRelative(player->getPositionRelative()); //Updates the waveform representation
            every 100 ms to reflect the audio file's current timestamp.a
182 }
```

There is also the replay function. The user can restart the track by pressing the start button at any position/time of the song.

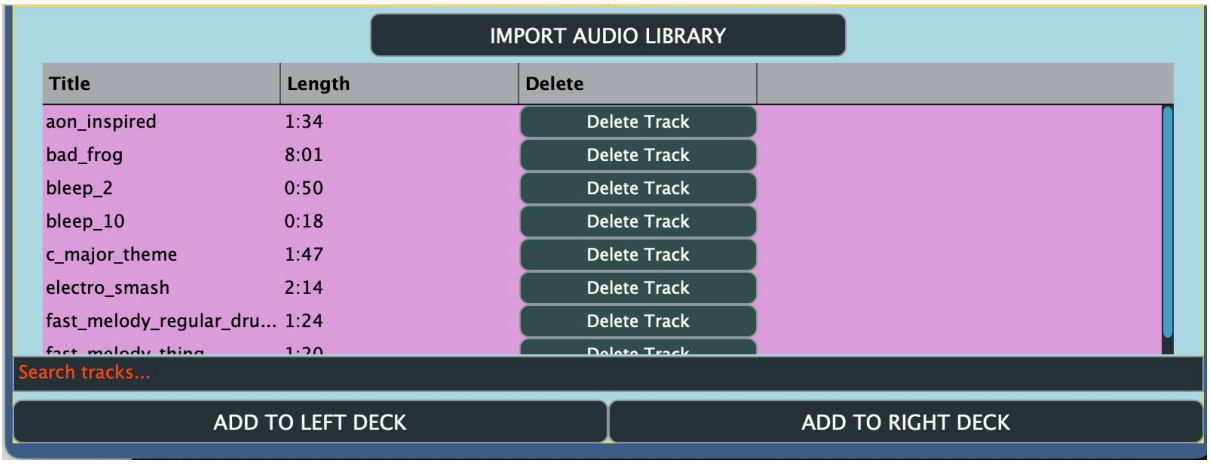
```
void DeckGUI::buttonClicked(Button* button) // pointer to button; memory address
{
    if (button == &playButton) // button to start audio file play
    {
        DBG("Play button has been clicked!");
        Logger::outputDebugString("Play button!");
        player->setPosition(0); // player calls setPosition(0) function from DJAudioPlayer to start playing from
            the beginning
        player->start(); // player calls start function from DJAudioPlayer


    }
```

<u>R4C: The R3 music library component is incorporated into the layout</u>

The import tracks button serves as the foundation for the music library, which also includes three headers for track titles, music length, and a delete button for library management. Users will initially see an empty library, followed by a full library once the library is filled with songs. The table list box's footer still includes the add to left

deck and add to right deck buttons, and the search field section is placed immediately below the file list.




```
addAndMakeVisible(tableComponent);

addAndMakeVisible(importButton);
addAndMakeVisible(searchField);
addAndMakeVisible(library);
addAndMakeVisible(addToPlayer1Button);
addAndMakeVisible(addToPlayer2Button);

importButton.addListener(this);
searchField.addListener(this);
addToPlayer1Button.addListener(this);
addToPlayer2Button.addListener(this);

searchField.setTextToShowWhenEmpty("Search tracks...", juce::Colours::orangered);
searchField.onReturnKey = [this] { searchLibrary(searchField.getText()); };

// setup table and load library from file
library.getHeader().addColumn("Title", 1, 1);
library.getHeader().addColumn("Length", 2, 1);
library.getHeader().addColumn("Delete", 3, 1);
library.setModel(this);
loadLibrary();
```