

# OOP Final Report – OtoDecks

## Table of contents

OOP Final Report - OtoDecks.....	1
Introduction.....	2
Basic functionality (R1) .....	3
R1A: can load audio files into audio players .....	3
R1B: playing 2 tracks at the same time .....	5
R1C: Can mix the tracks by varying each of their volumes .....	6
R1D: Can speed up and slow down the tracks .....	7
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. ....	8
R2A: Components with custom graphics .....	8
R2B: Component enables the user to control the playback of a deck somehow .....	9
R3: Implementation of a music library component which allows the user to manage their music library .....	11
R3A: Component allows the user to add files to their library .....	11
R3B: Component parses and displays meta data such as filename and song length.....	13
R3C: Component allows the user to search for files.....	14
R3D: Component allows the user to load files from the library into a deck .....	16
R3E: The music library persists so that it is restored when the user exits then restarts the application .....	16
R4: Implementation of a complete custom GUI .....	18
R4A, B, C: Layout is different, it includes custom components, and music library .....	19

## Introduction

OtoDecks is a desktop DJ application written in C++ leveraging the JUCE Framework, which offers ready-made, customizable (GUI and audio related) components that are perfect for our use case.

The application allows to:

- Import tracks from the user's computer to the application playlist
- Save the playlist so that it is reloaded as saved even if the app is closed and reopened
- Load a track to the left deck and control it autonomously. Load a track to the right deck and control it autonomously.
- Play two tracks simultaneously and control their speed, volume, track position, and reverb properties through intuitive GUI components such as knobs and sliders.

This application was developed as final project for the class of Object-Oriented Programming, for which code-related best practices have been followed. For example, the different application components are divided into a series of header (interfaces) and .cpp files (implementation), and code documentation can be found for each function within the header files (following [these guidelines](#)).

This is how the application looks like:



Figure 1 - App appearance

In this report I will describe the structure of the application, highlighting how its features were implemented. I have implemented all requirements.

## Basic functionality (R1)

OtoDecks contains all the basic functionality shown in class: R1A, R1B, R1C, and R1D. Below is a detailed description for each requirement implementation.

### R1A: can load audio files into audio players

When the user clicks on the 'Import songs' button, the function 'buttonClicked' within file 'PlaylistComponent.cpp' is called.

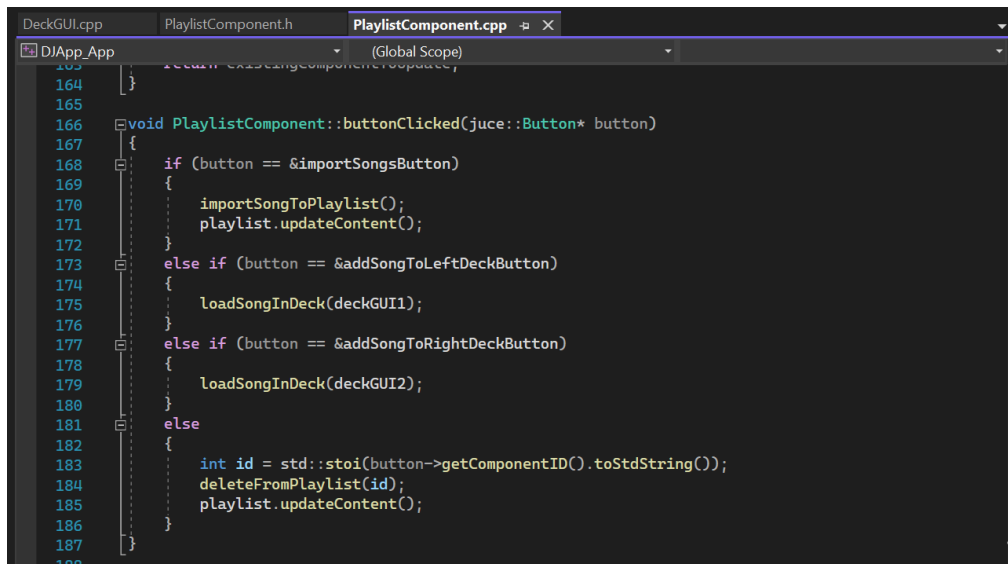


Figure 2 - PlaylistComponent::buttonClicked()

Here, we detect what button was clicked and call different logic accordingly. In this case, the condition at line 168 is met and function 'importSongToPlaylist()' is called. After this, the content of the playlist is updated to include the new song.

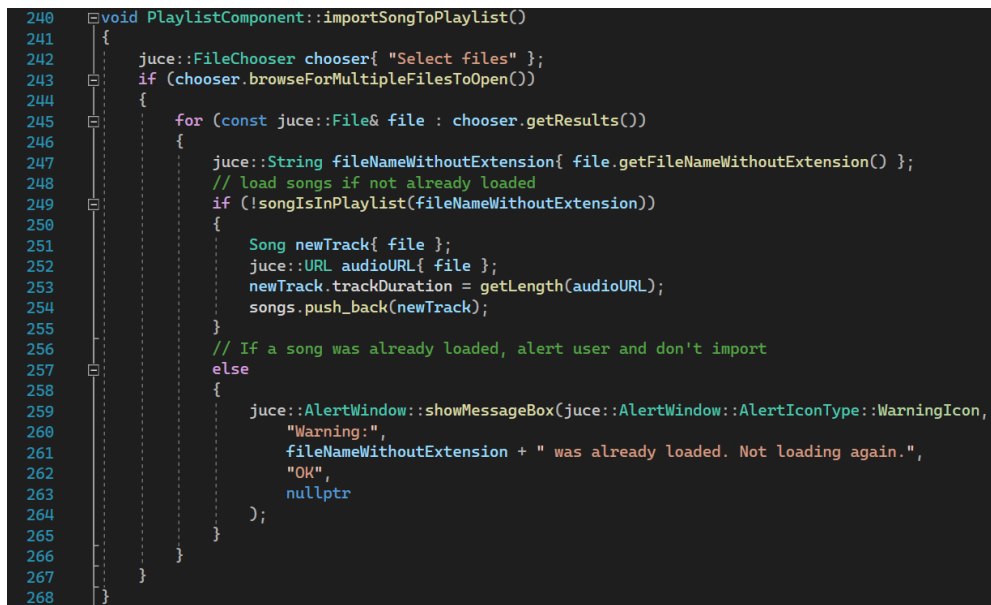
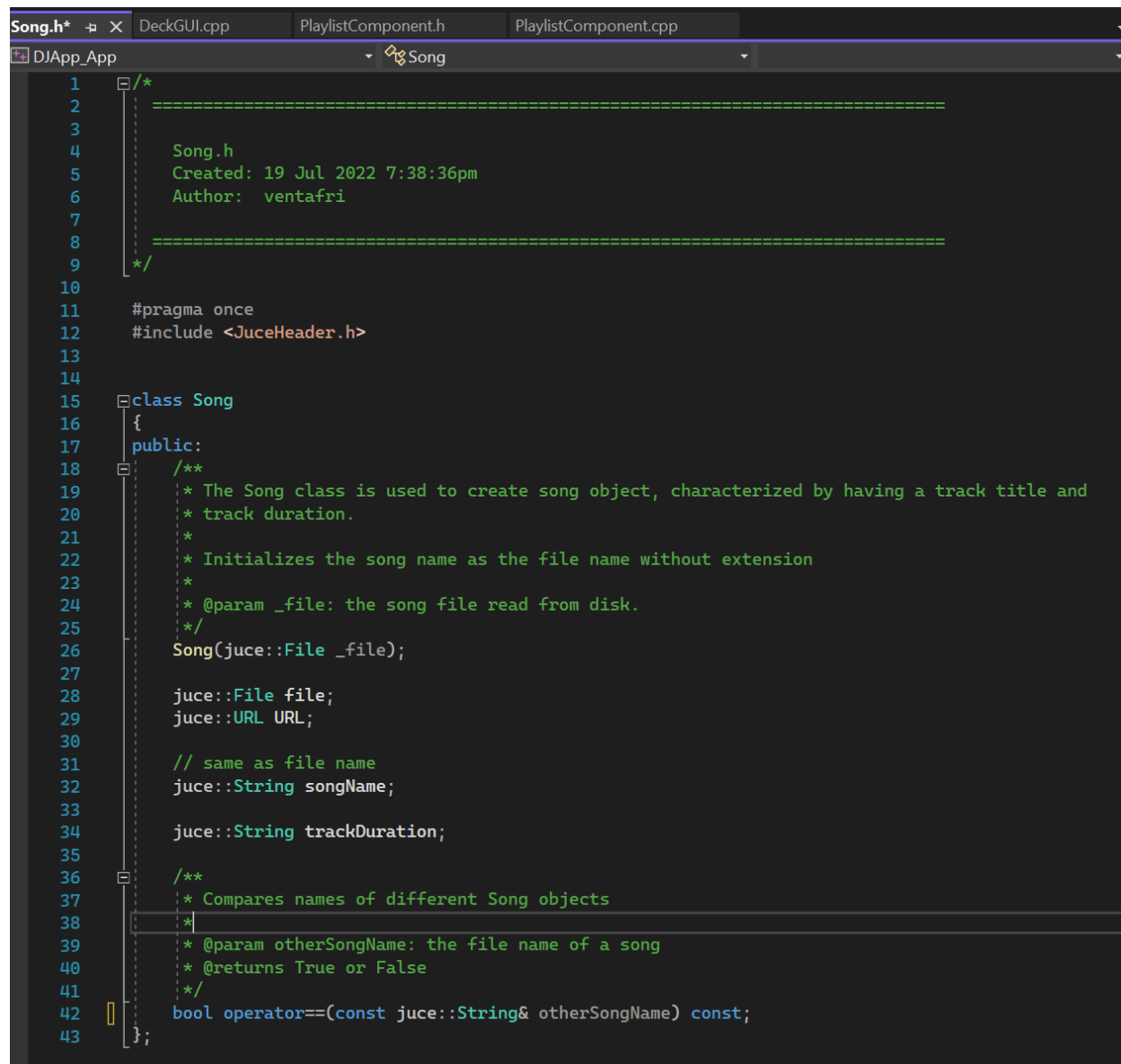


Figure 3 - PlaylistComponent::importSongToPlaylist()

Here, the logic allows the user to browse and select multiple files. If the selected song is not already loaded in the playlist, class `juce::URL` allows us to load the audio file, while at line 251 an object of class `Song` is created and then added to the playlist at line 254.

The `Song` class contains a constructor (l. 26), called every time a new song is added to the playlist public properties: `songName`, `trackDuration`, `file`, `URL`, and a Boolean operator used to compare song names.



```

1  /**
2  =====
3
4  Song.h
5  Created: 19 Jul 2022 7:38:36pm
6  Author:  ventafri
7
8  =====
9  */
10
11 #pragma once
12 #include <JuceHeader.h>
13
14
15 class Song
16 {
17 public:
18     /**
19     * The Song class is used to create song object, characterized by having a track title and
20     * track duration.
21     *
22     * Initializes the song name as the file name without extension
23     *
24     * @param _file: the song file read from disk.
25     */
26     Song(juce::File _file);
27
28     juce::File file;
29     juce::URL URL;
30
31     // same as file name
32     juce::String songName;
33
34     juce::String trackDuration;
35
36     /**
37     * Compares names of different Song objects
38     *
39     * @param otherSongName: the file name of a song
40     * @returns True or False
41     */
42     bool operator==(const juce::String& otherSongName) const;
43 };

```

Figure 4 - `Song` class .h

Once the song is in the playlist, the user can select it by clicking on it and then click on buttons 'Add to left' or 'Add to right' to add the song to the left or right deck respectively. Clicking on 'Add to left' or 'Add to right' triggers the call of function 'loadSongInDeck()' (Figure 2, line 175 or 179), which actually loads the song to the selected deck (left or right), at line 226.

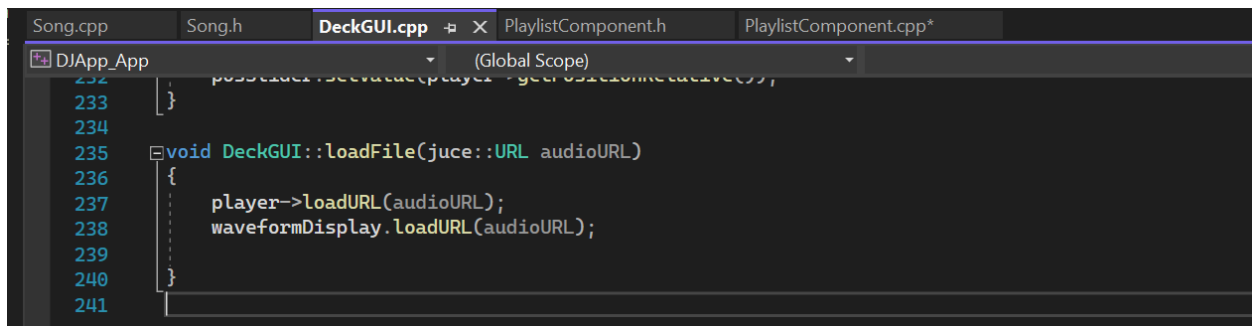
```

220
221 void PlaylistComponent::loadSongInDeck(DeckGUI* deckGUI)
222 {
223     int selectedRow{ playlist.getSelectedRow() };
224     if (selectedRow != -1)
225     {
226         deckGUI->loadFile(songs[selectedRow].URL);
227     }
228     else
229     {
230         juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
231             "Add to Deck Information:",
232             "Please select a track to add to deck",
233             "OK",
234             nullptr
235         );
236     }

```

Figure 5 - PlaylistComponent::loadSongInDeck(DeckGUI\* deckGUI)

Calling DeckGUI::loadFile has the effect of drawing the waveform of the song onto the selected deck. The song can now be played.



```

Song.cpp Song.h DeckGUI.cpp PlaylistComponent.h PlaylistComponent.cpp*
DJApp_App (Global Scope)
232
233 }
234
235 void DeckGUI::loadFile(juce::URL audioURL)
236 {
237     player->loadURL(audioURL);
238     waveformDisplay.loadURL(audioURL);
239 }
240
241

```

Figure 6 - DeckGUI::loadFile(juce::URL audioURL)

## R1B: playing 2 tracks at the same time

The DJAudioPlayer class is responsible for loading the song URL, setting the readerSource (l. 56, Figure 7) and transportSource (Figure 7, l. 55) equal to the newly created juce::AudioFormatReaderSource unique pointer (Figure 7, line 54).

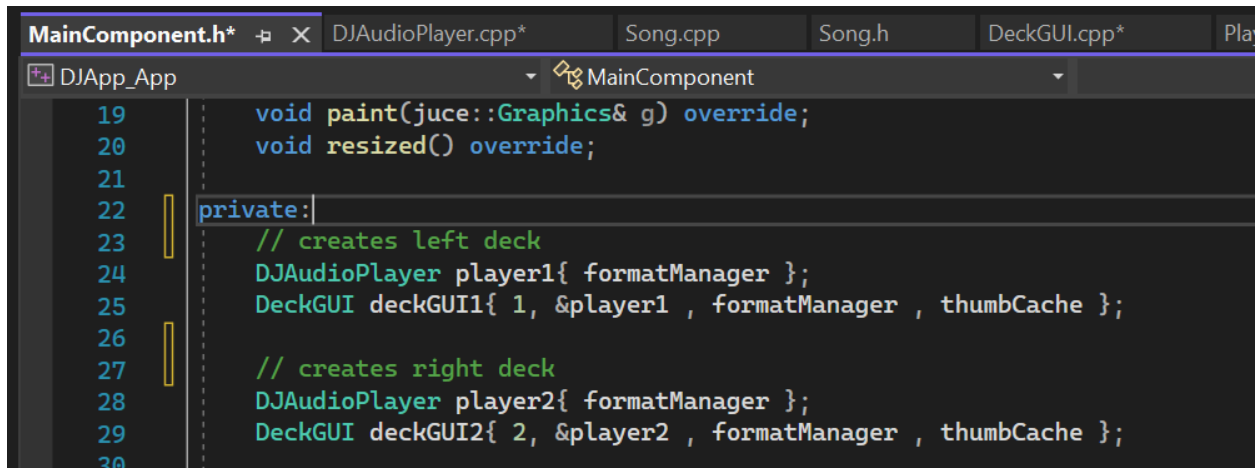
```

40
47 void DJAudioPlayer::loadURL(juce::URL audioURL)
48 {
49     // takes audio url input string, passes it to formatManager, and creates a Reader
50     auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
51     // check if it successfully created the reader aka the file is readable
52     if (reader != nullptr)
53     {
54         std::unique_ptr<juce::AudioFormatReaderSource> newSource(new juce::AudioFormatReaderSource
55             transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
56         readerSource.reset(newSource.release());
57     }
58 }
59
60

```

Figure 7 - DJAudioPlayer::loadURL(juce::URL audioURL)

The application creates two different DeckGUI objects, each with an autonomous DJAudioPlayer, as can be seen in Figure 8. This allows two tracks to be loaded at the same time: one on the left deck and one on the right deck.



```

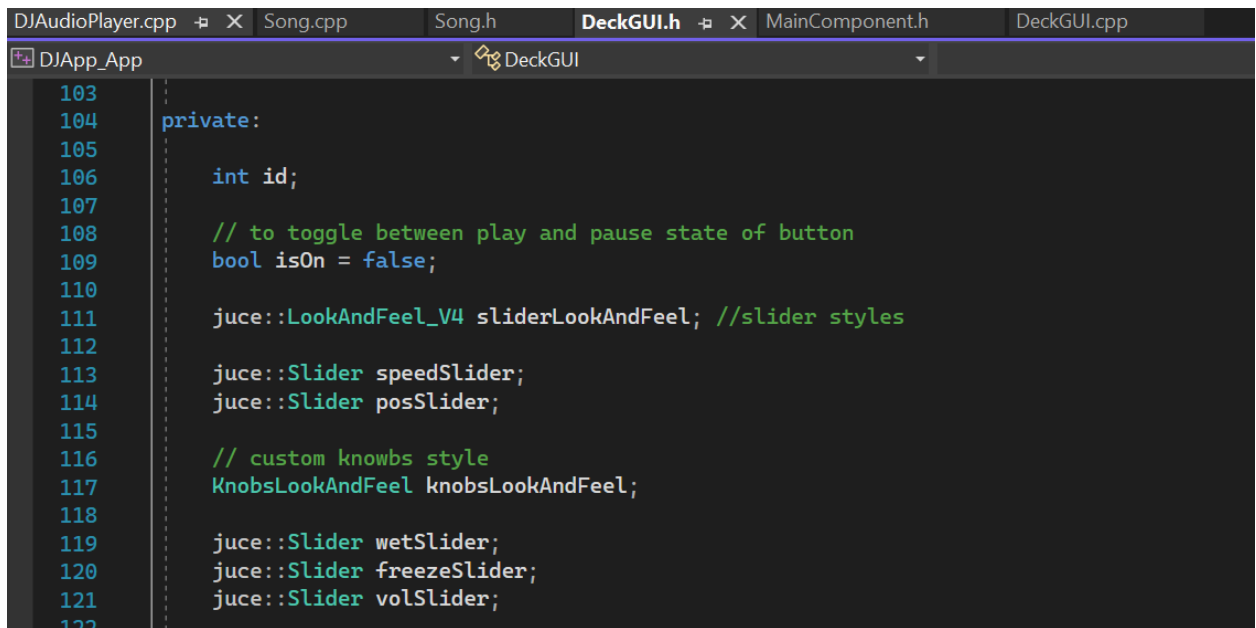
MainComponent.h*  DJAudioPlayer.cpp*  Song.cpp  Song.h  DeckGUI.cpp*  Play
DJApp_App  MainComponent
19  void paint(juce::Graphics& g) override;
20  void resized() override;
21
22  private:
23      // creates left deck
24      DJAudioPlayer player1{ formatManager };
25      DeckGUI deckGUI1{ 1, &player1 , formatManager , thumbCache };
26
27      // creates right deck
28      DJAudioPlayer player2{ formatManager };
29      DeckGUI deckGUI2{ 2, &player2 , formatManager , thumbCache };
30

```

Figure 8 - MainComponent.h creates two DeckGUI objects

## R1C: Can mix the tracks by varying each of their volumes

From figure 1, we can see that each deck has a separate volume knob. Each knob is implemented as an object of class Slider, within file 'DeckGUI.h', line 121 (Figure 9).



```

DJAudioPlayer.cpp  Song.cpp  Song.h  DeckGUI.h  MainComponent.h  DeckGUI.cpp
DJApp_App  DeckGUI
103
104  private:
105
106      int id;
107
108      // to toggle between play and pause state of button
109      bool isOn = false;
110
111      juce::LookAndFeel_V4 sliderLookAndFeel; //slider styles
112
113      juce::Slider speedSlider;
114      juce::Slider posSlider;
115
116      // custom knobs style
117      KnobsLookAndFeel knobsLookAndFeel;
118
119      juce::Slider wetSlider;
120      juce::Slider freezeSlider;
121      juce::Slider volSlider;
122

```

Figure 9 - DeckGUI.h, volSlider

As we have seen in figure 8, each deck is instantiated separately so as to have their own separate audio controls. Hence, the two volSlider objects (left and right deck) are separate.

Whenever the user turns the left or right volume knob, function 'DeckGUI::sliderValueChanged' is called. This can be found at line 184 of DeckGUI.cpp (figure 10).

```

184 void DeckGUI::sliderValueChanged(juce::Slider* slider)
185 {
186     if (slider == &volSlider)
187     {
188         player->setGain(gain: slider->getValue());
189     }
190     if (slider == &speedSlider)
191     {
192         player->setSpeed(ratio: slider->getValue());
193     }
194     if (slider == &posSlider)
195     {
196         player->setPositionRelative(pos: slider->getValue());
197     }
198     if (slider == &wetSlider)
199     {
200         player->setWetLevel(wetLevel: slider->getValue());
201     }
202     if (slider == &freezeSlider)
203     {
204         player->setFreeze(freezeAmt: slider->getValue());
205     }
206 }
207

```

Figure 10 - DeckGUI::sliderValueChanged

This function detects which slider control was changed by the user and call a specific function accordingly. In this case, it is calling the setGain function of the deck's player (line 188, fig. 10). Player is here the deck's DJAudioPlayer object.

Figure 11 displays the content of setGain function. In here, method 'setGain' is called on the transportSource object, effectively increasing the volume of the deck's track after checking that the gain value argument is in the inclusive range 0 to 1.

```

59 };
60
61 void DJAudioPlayer::setGain(double gain)
62 {
63     if (gain <= 0 || gain > 1.0) {
64         if (gain < 0)
65         {
66             // Set minimum to avoid error
67             gain = 1;
68         }
69     }
70     else {
71         transportSource.setGain(newGain: gain);
72     }
73 }

```

Figure 11 - DJAudioPlayer::setGain(double gain)

## R1D: Can speed up and slow down the tracks

From figure 1, we can see that each deck has a separate speed knob. Each knob is implemented as an object of class Slider, within file 'DeckGUI.h'. The speed slider is instantiated at line 113 of figure 9.

When the user turns the speed knob, function 'DeckGUI::sliderValueChanged' (fig. 10) is called, entering the logic at line 192 and calling the player method's 'setSpeed()'.

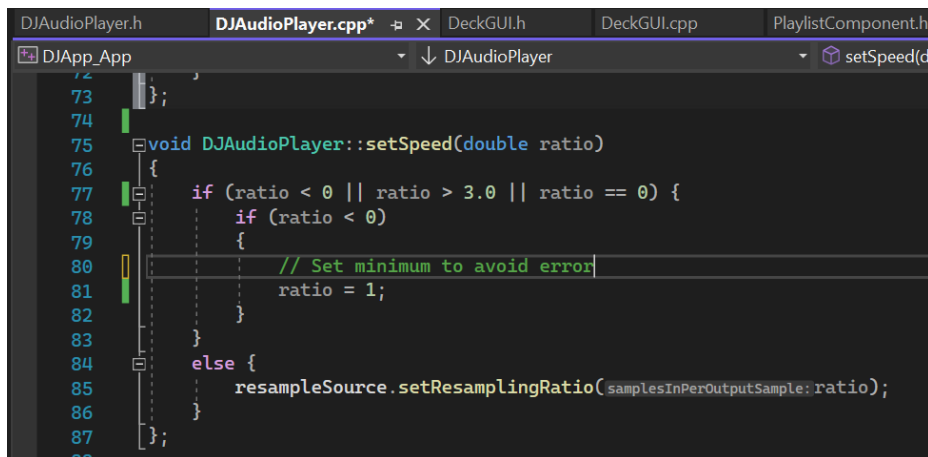


Figure 12 - DJAudioPlayer::setSpeed(double ratio)

DJAudioPlayer::setSpeed(double ratio) checks if the argument is within the accepted range 0 and 1 and if so, passes this value to the 'setResamplingRatio' method of the transportSource. This has the effect of speeding up or down the playback speed of the song playing on the deck, on which the speed knob is turned.

## 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.

I have added two custom ways in which the user can control the deck playback:

- 1) Right below the wave form of the track, a horizontal slider allows to play a specific relative position of the loaded track. The slider has a customized look.
- 2) Although not related to changing the timeline of the track, I have implemented custom graphics for the knobs on the GUI and added two extra functionalities: a knob to control wet level of the track's reverb, and a knob to control the freeze property of the track's reverb.
- 3) In addition to the play/pause button, I have added a rewind and fast forward button, with customized graphics (ImageButton).

### R2A: Components with custom graphics

- 1) The posSlider initialized as seen in Figure 9, line 114, is then customized in DeckGUI.cpp:

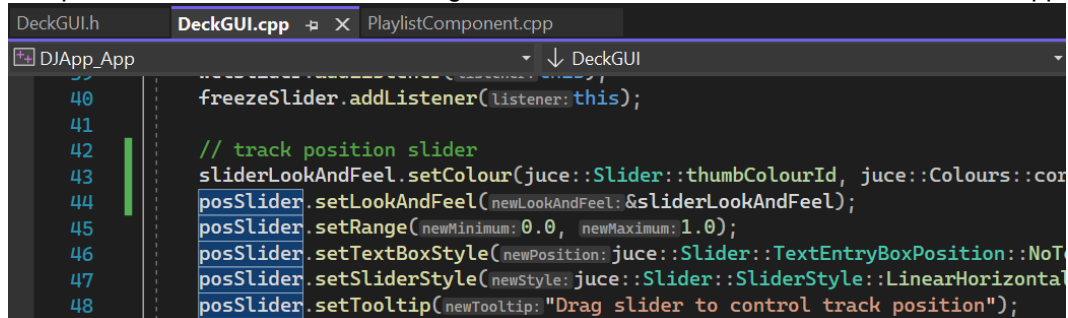
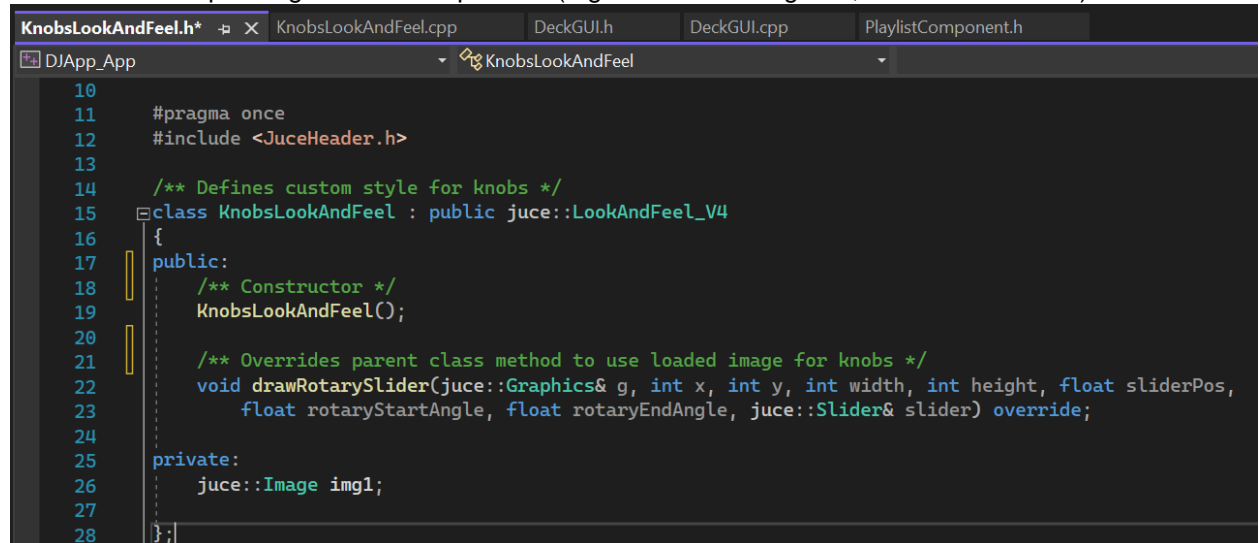


Figure 13 - posSlider custom graphics



In particular, we're changing the slider control color to coral (fig 13, line 43).

- 2) The custom knobs graphic has been achieved within the KnobsLookAndFeel class, which inherits from `juce::LookAndFeel_V4` and overrides the parent class method 'drawRotarySlider' with custom logic to display knobs images turning based on user interaction, as well as displaying a different color depending on the knob position (e.g. min volume = green, max volume = red).



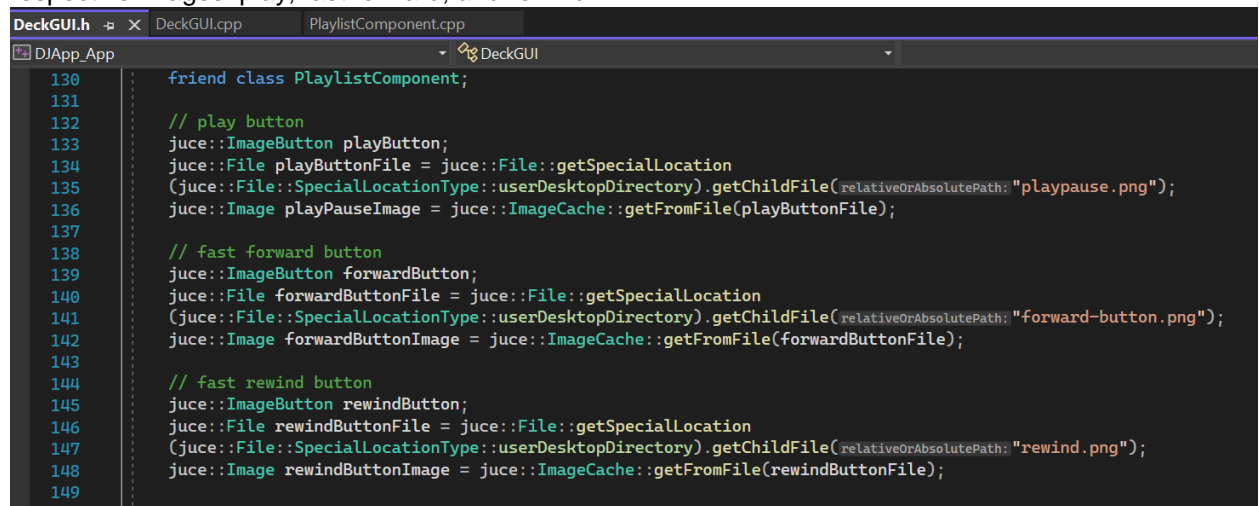
```

KnobsLookAndFeel.h  KnobsLookAndFeel.cpp  DeckGUI.h  DeckGUI.cpp  PlaylistComponent.h
DApp_App  KnobsLookAndFeel
10
11  #pragma once
12  #include <JuceHeader.h>
13
14  /** Defines custom style for knobs */
15  class KnobsLookAndFeel : public juce::LookAndFeel_V4
16  {
17  public:
18      /** Constructor */
19      KnobsLookAndFeel();
20
21      /** Overrides parent class method to use loaded image for knobs */
22      void drawRotarySlider(juce::Graphics& g, int x, int y, int width, int height, float sliderPos,
23                          float rotaryStartAngle, float rotaryEndAngle, juce::Slider& slider) override;
24
25  private:
26      juce::Image img1;
27
28  };

```

Figure 14 – KnobsLookAndFeel

- 3) In the private section of DeckGUI.h, three image buttons are instantiated and filled with the respective images: play, fast forward, and rewind.



```

DeckGUI.h  DeckGUI.cpp  PlaylistComponent.cpp
DApp_App  DeckGUI
130  friend class PlaylistComponent;
131
132  // play button
133  juce::ImageButton playButton;
134  juce::File playButtonFile = juce::File::getSpecialLocation
135  (juce::File::SpecialLocationType::userDesktopDirectory).getChildFile(relativeOrAbsolutePath: "playpause.png");
136  juce::Image playPauseImage = juce::ImageCache::getFromFile(playButtonFile);
137
138  // fast forward button
139  juce::ImageButton forwardButton;
140  juce::File forwardButtonFile = juce::File::getSpecialLocation
141  (juce::File::SpecialLocationType::userDesktopDirectory).getChildFile(relativeOrAbsolutePath: "forward-button.png");
142  juce::Image forwardButtonImage = juce::ImageCache::getFromFile(forwardButtonFile);
143
144  // fast rewind button
145  juce::ImageButton rewindButton;
146  juce::File rewindButtonFile = juce::File::getSpecialLocation
147  (juce::File::SpecialLocationType::userDesktopDirectory).getChildFile(relativeOrAbsolutePath: "rewind.png");
148  juce::Image rewindButtonImage = juce::ImageCache::getFromFile(rewindButtonFile);
149
150

```

Figure 15 - track controls – ImageButtons

## R2B: Component enables the user to control the playback of a deck somehow

- 1) The fast forward and rewind buttons, when clicked upon, trigger the execution of lines 168-170 and 176-179 respectively (figure 16). Here, the relative position of the track is incremented / decremented by 0.05, having the effect of jumping to a future track point or previous point respectively.

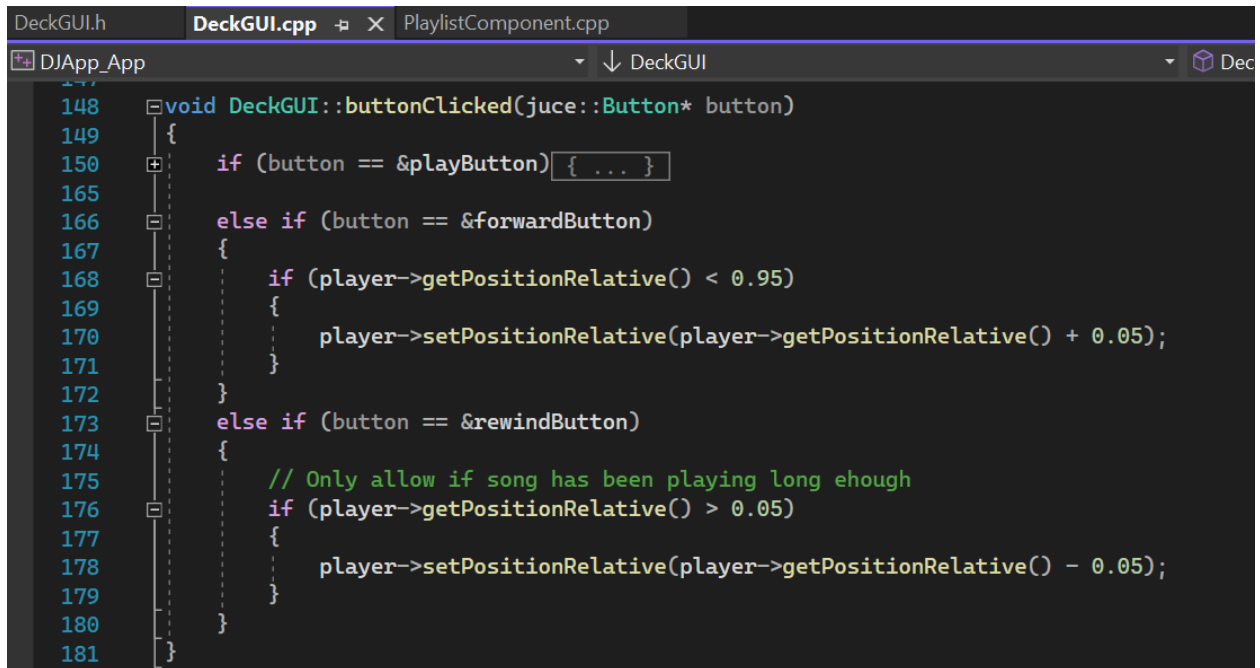


Figure 16 - buttonClicked - fast forward and rewind

2) Reverb modifications. Wet level and freeze knobs.

At lines 198 and 202 of Figure 10 we can see that when a user turns the wet knob, the player's method 'setWetLevel' is called, while method 'setFreeze' is called by turning the freeze knob. In DJAudioPlayer.cpp we can find the implementation of these functions:

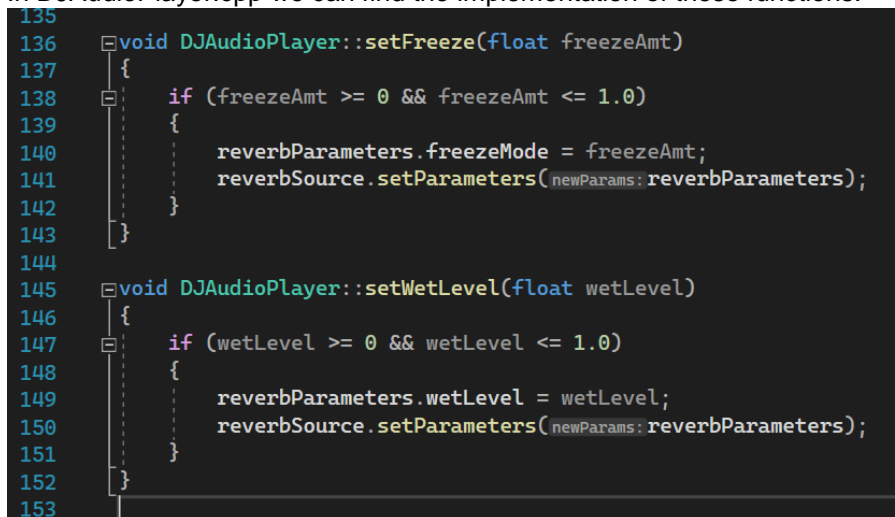


Figure 17 - reverb modifications

These functions alter the track reverb and thus modify the audio output. A high 'wet level' makes the song sound very 'metallic', while the freeze knob will keep looping the song at the moment it was turned (even when the song is paused or stopped), allowing for nice track mixing effects.

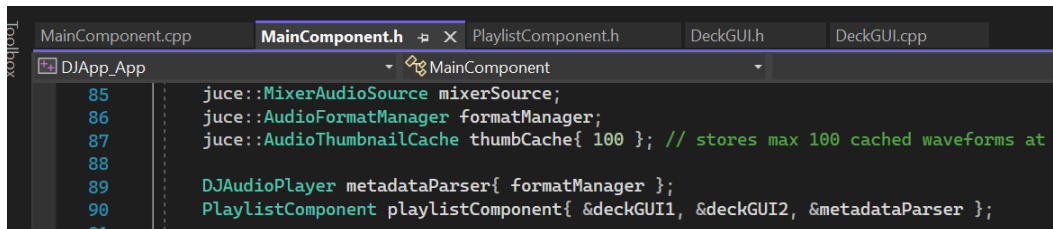
3) The posSlider (horizontal slider below the track's waveform display) can be controlled by changing the relative position of the track that is playing. Figure 10, line 196 displays that whenever the slider's position is changed, function 'setPositionRelative' is called, which changes the track's relative position.

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

A music library component is added to the app via the class PlaylistComponent, which interface can be found in the file PlaylistComponent.h and which implementation can be found in the file PlaylistComponent.cpp.

### R3A: Component allows the user to add files to their library

All the functionality of the playlist is handled by the playlistComponent class, including the visual components. An object of the playlist component is created as a private attribute within the MainComponent.h file (figure 18, line 90).

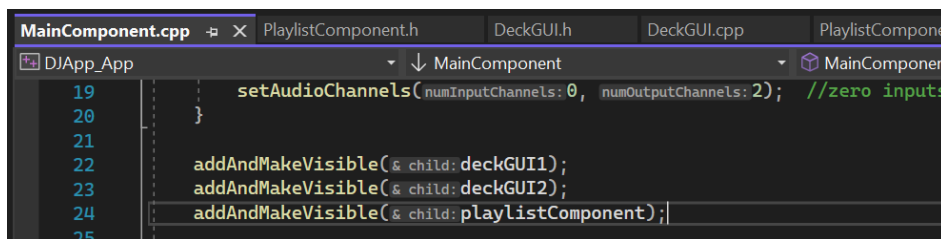


```

MainComponent.h
85     juce::MixerAudioSource mixerSource;
86     juce::AudioFormatManager formatManager;
87     juce::AudioThumbnailCache thumbCache{ 100 }; // stores max 100 cached waveforms at a
88
89     DJAudioPlayer metadataParser{ formatManager };
90     PlaylistComponent playlistComponent{ &deckGUI1, &deckGUI2, &metadataParser };
91
  
```

Figure 18 - PlaylistComponent object creation

The playlist component is then rendered within MainComponent.cpp (l. 24, figure 19).



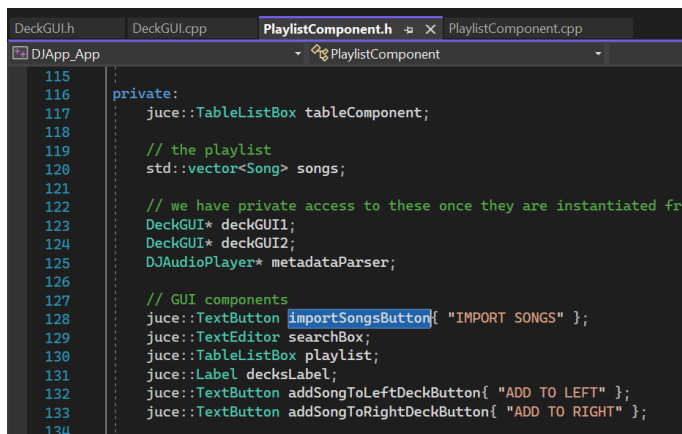
```

MainComponent.cpp
19     setAudioChannels(numInputChannels: 0, numOutputChannels: 2); //zero inputs
20 }
21
22     addAndMakeVisible(& child: deckGUI1);
23     addAndMakeVisible(& child: deckGUI2);
24     addAndMakeVisible(& child: playlistComponent);
25
  
```

Figure 19 - rendered playlistComponent

Within MainComponent::resized(), the playlist's bounds are also defined.

Within PlaylistComponent.h, object importSongsButton is created as an instance of the class TextButton.

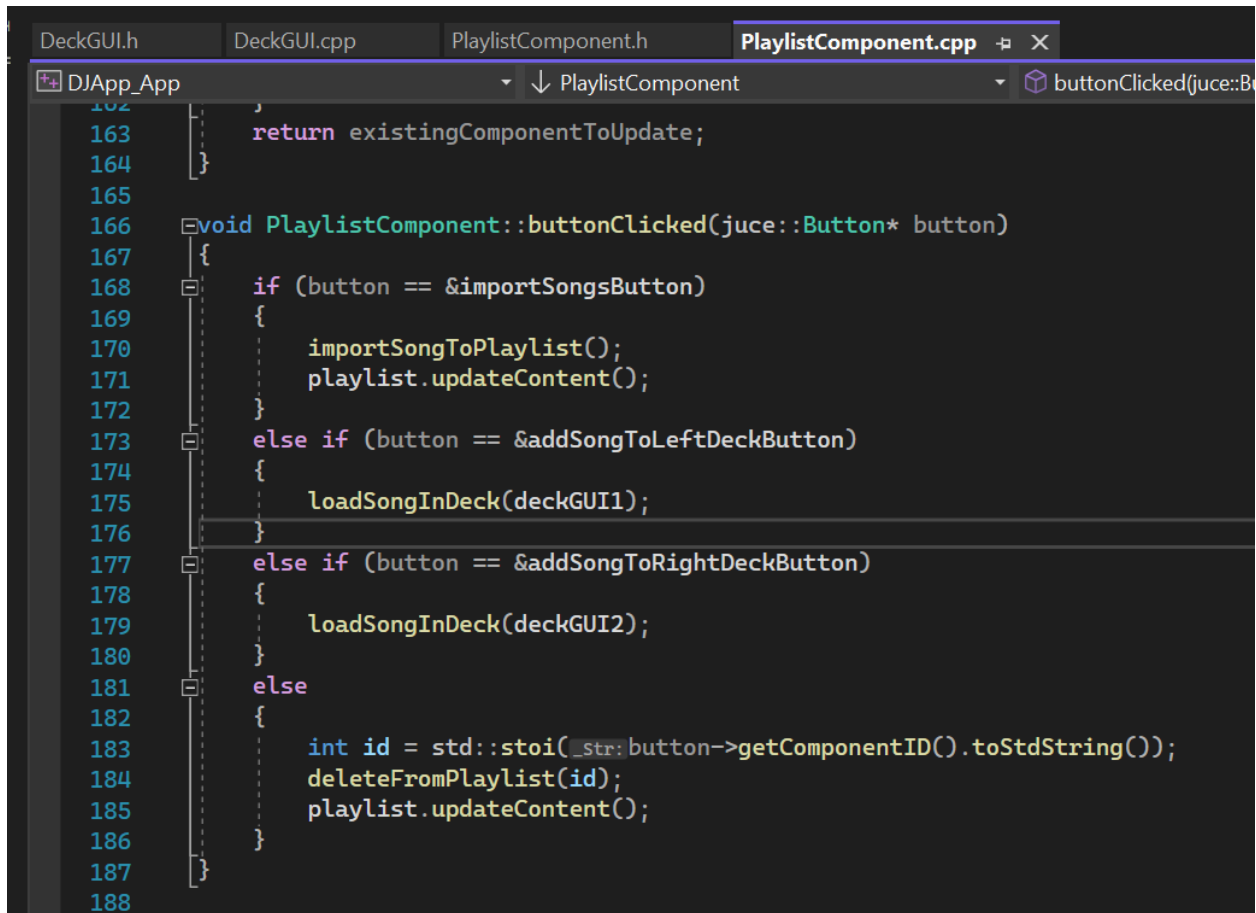


```

PlaylistComponent.h
115 private:
116     juce::TableListBox tableComponent;
117
118     // the playlist
119     std::vector<Song> songs;
120
121     // we have private access to these once they are instantiated from
122     DeckGUI* deckGUI1;
123     DeckGUI* deckGUI2;
124     DJAudioPlayer* metadataParser;
125
126     // GUI components
127     juce::TextButton importSongsButton{ "IMPORT SONGS" };
128     juce::TextEditor searchBox;
129     juce::TableListBox playlist;
130     juce::Label decksLabel;
131     juce::TextButton addSongToLeftDeckButton{ "ADD TO LEFT" };
132     juce::TextButton addSongToRightDeckButton{ "ADD TO RIGHT" };
133
134
  
```

Figure 20 - creation of import songs button

The button's position, settings are then defined within PlaylistComponent.h, in which a listener is also attached to the button. Thanks to this listener, when the user clicks on the 'Import songs' button, the function buttonClicked is called, executing in this case the logic at lines 170 and 171 (fig. 21).



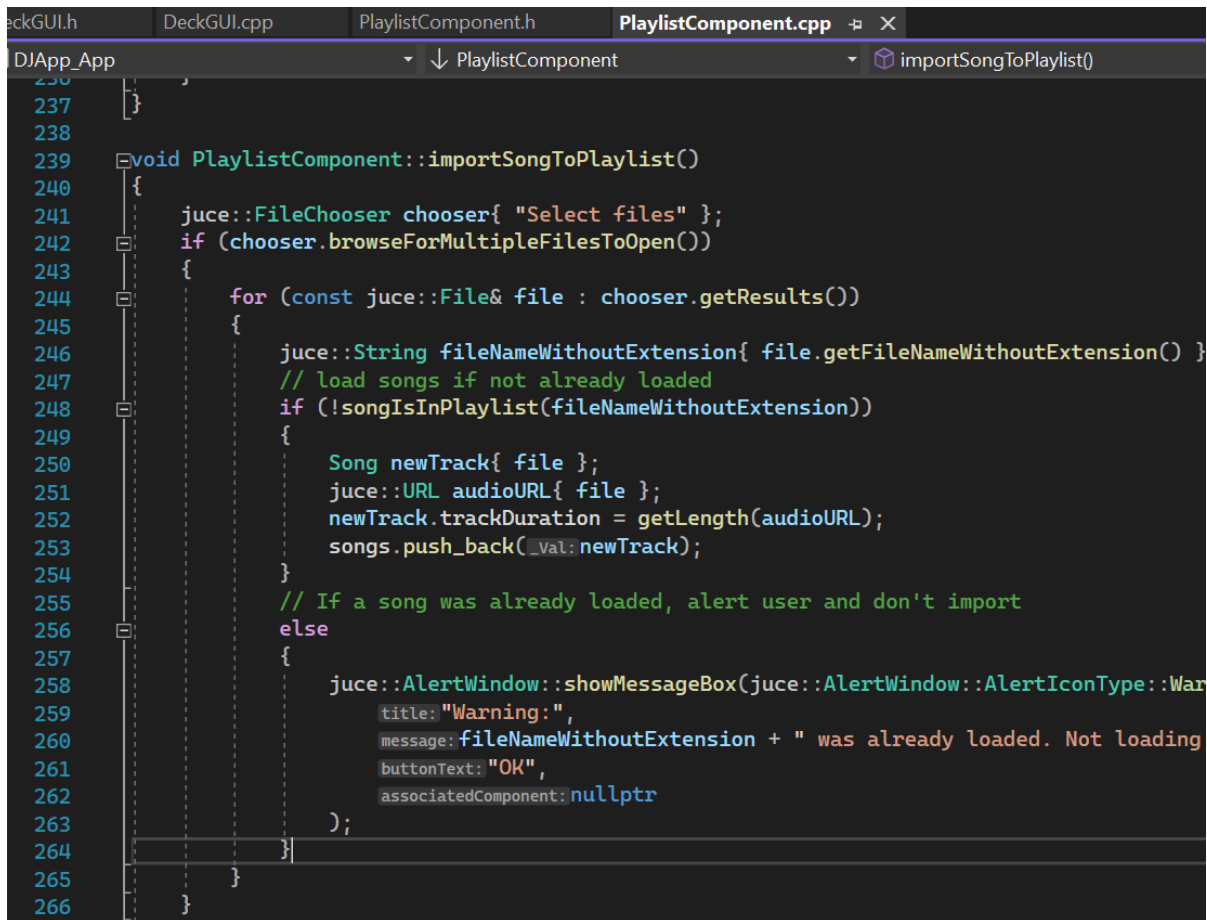
```

162
163     return existingComponentToUpdate;
164 }
165
166 void PlaylistComponent::buttonClicked(juce::Button* button)
167 {
168     if (button == &importSongsButton)
169     {
170         importSongToPlaylist();
171         playlist.updateContent();
172     }
173     else if (button == &addSongToLeftDeckButton)
174     {
175         loadSongInDeck(deckGUI1);
176     }
177     else if (button == &addSongToRightDeckButton)
178     {
179         loadSongInDeck(deckGUI2);
180     }
181     else
182     {
183         int id = std::stoi(_str:button->getComponentID().toStdString());
184         deleteFromPlaylist(id);
185         playlist.updateContent();
186     }
187 }
188

```

Figure 21 - playlist button clicked

Figure 22 displays function PlaylistComponent::importSongToPlaylist(). The function instantiates a Juce FileChooser object (line 241) called 'chooser'. We can then call the method 'browseForMultipleFilesToOpen()' on this object, which allows the user to select multiple files from their local drive. For each file chosen, we get the file name without extension and check if the song was already saved to the playlist. If this is the case, a warning message is displayed to the user. Otherwise, the song is loaded to the playlist. This is achieved by creating an object of class Song, passing the file as argument to the constructor, then creating an audioURL object also passing the file to the class constructor (audioURL object is used to assign the track duration to the song object). Finally, we add the new song object to the playlist, which is effectively an array of Song objects called songs (line 253, fig. 22).



```

230 }
231 }
232 }
233
234 void PlaylistComponent::importSongToPlaylist()
235 {
236     juce::FileChooser chooser{ "Select files" };
237     if (chooser.browseForMultipleFilesToOpen())
238     {
239         for (const juce::File& file : chooser.getResults())
240         {
241             juce::String fileNameWithoutExtension{ file.getFileNameWithoutExtension() };
242             // load songs if not already loaded
243             if (!songIsInPlaylist(fileNameWithoutExtension))
244             {
245                 Song newTrack{ file };
246                 juce::URL audioURL{ file };
247                 newTrack.trackDuration = getLength(audioURL);
248                 songs.push_back(_val: newTrack);
249             }
250             // If a song was already loaded, alert user and don't import
251             else
252             {
253                 juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::Warning,
254                     title: "Warning:",
255                     message: fileNameWithoutExtension + " was already loaded. Not loading",
256                     buttonText: "OK",
257                     associatedComponent: nullptr
258                 );
259             }
260         }
261     }
262 }

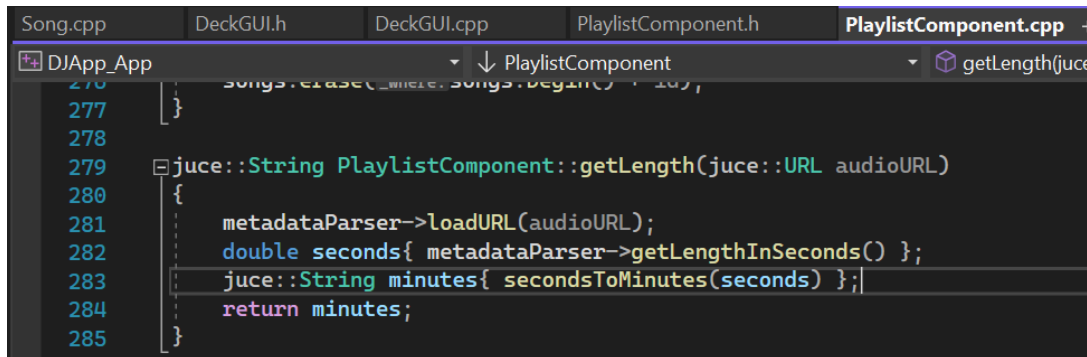
```

Figure 22 - PlaylistComponent::importSongToPlaylist()

### R3B: Component parses and displays meta data such as filename and song length

As we have seen above, each song is loaded to the playlist as a song object, which has the following properties: songName and trackDuration (figure 4). Attribute trackDuration is set at line 252 in fig 22, while the song name is saved when the Song constructor is run with the file as argument.

Function PlaylistComponent::getLength (fig. 23) parses the file metadata making use of the audioURL.



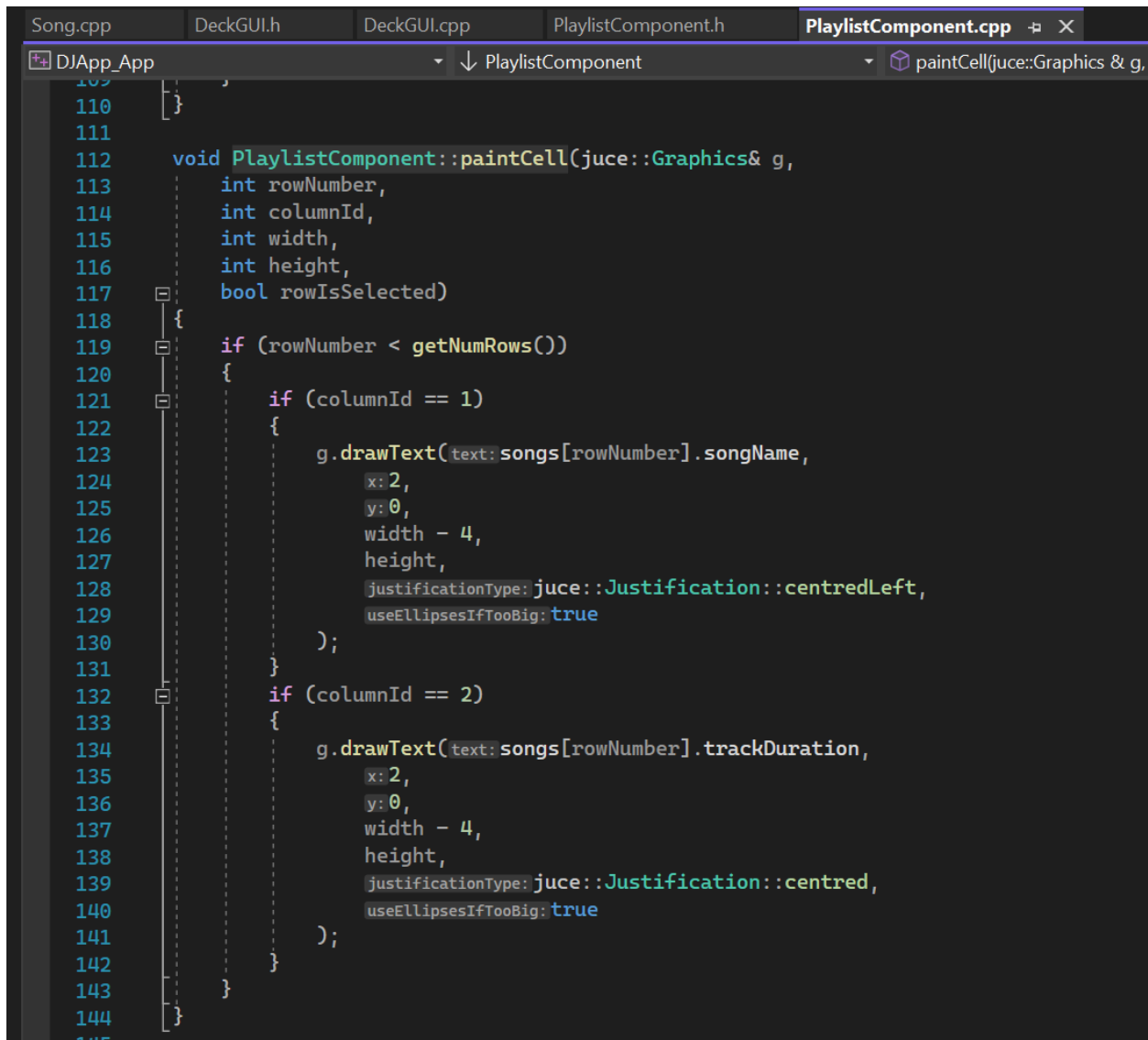
```

270 songs.erase(_val: songs.begin() + id);
271 }
272
273 juce::String PlaylistComponent::getLength(juce::URL audioURL)
274 {
275     metadataParser->loadURL(audioURL);
276     double seconds{ metadataParser->getLengthInSeconds() };
277     juce::String minutes{ secondsToMinutes(seconds) };
278     return minutes;
279 }

```

Figure 23 - get track length

Song length and track names are then displayed on the GUI thanks to function `PlaylistComponent::paintCell` (figure 24).



```

Song.cpp    DeckGUI.h    DeckGUI.cpp    PlaylistComponent.h    PlaylistComponent.cpp
DJApp_App  PlaylistComponent  paintCell(juce::Graphics & g,
109
110 }
111
112 void PlaylistComponent::paintCell(juce::Graphics& g,
113     int rowNum,
114     int columnId,
115     int width,
116     int height,
117     bool rowIsSelected)
118 {
119     if (rowNum < getNumRows())
120     {
121         if (columnId == 1)
122         {
123             g.drawText(songs[rowNum].songName,
124                 x: 2,
125                 y: 0,
126                 width - 4,
127                 height,
128                 juce::Justification::centredLeft,
129                 useEllipsesIfTooBig: true
130             );
131         }
132         if (columnId == 2)
133         {
134             g.drawText(songs[rowNum].trackDuration,
135                 x: 2,
136                 y: 0,
137                 width - 4,
138                 height,
139                 juce::Justification::centred,
140                 useEllipsesIfTooBig: true
141             );
142         }
143     }
144 }
145

```

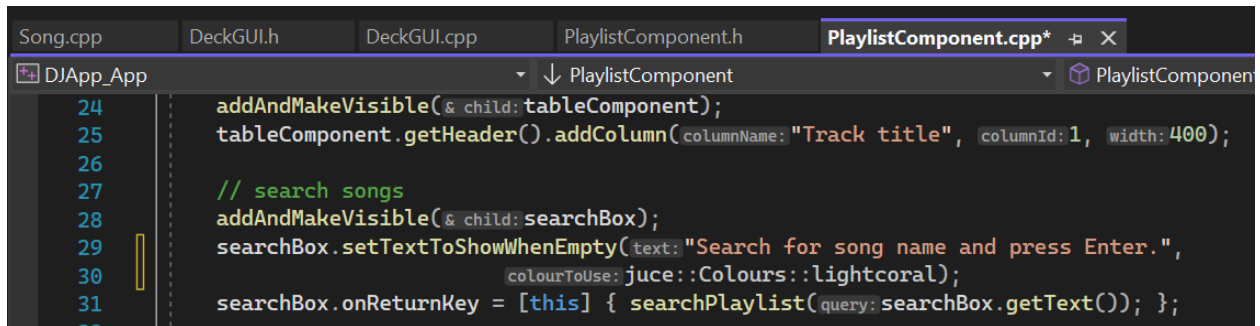
Figure 24 - `PlaylistComponent::paintCell`

`paintCell()` paints a row for each song within the playlist at the center of the GUI. For each song, it displays the track name and duration.

### R3C: Component allows the user to search for files

Figure 20, line 129 displays the creation of the `searchBox` object, which is an object of class `juce::TextEditor`.

Within `PlaylistComponent.cpp`, we set function 'searchPlaylist' to be called every time the return/ enter button is pressed (figure 25, line 31). So when the user types something in the search bar at the top of the playlist, the text typed by the user is passed as argument to the `searchPlaylist` function.



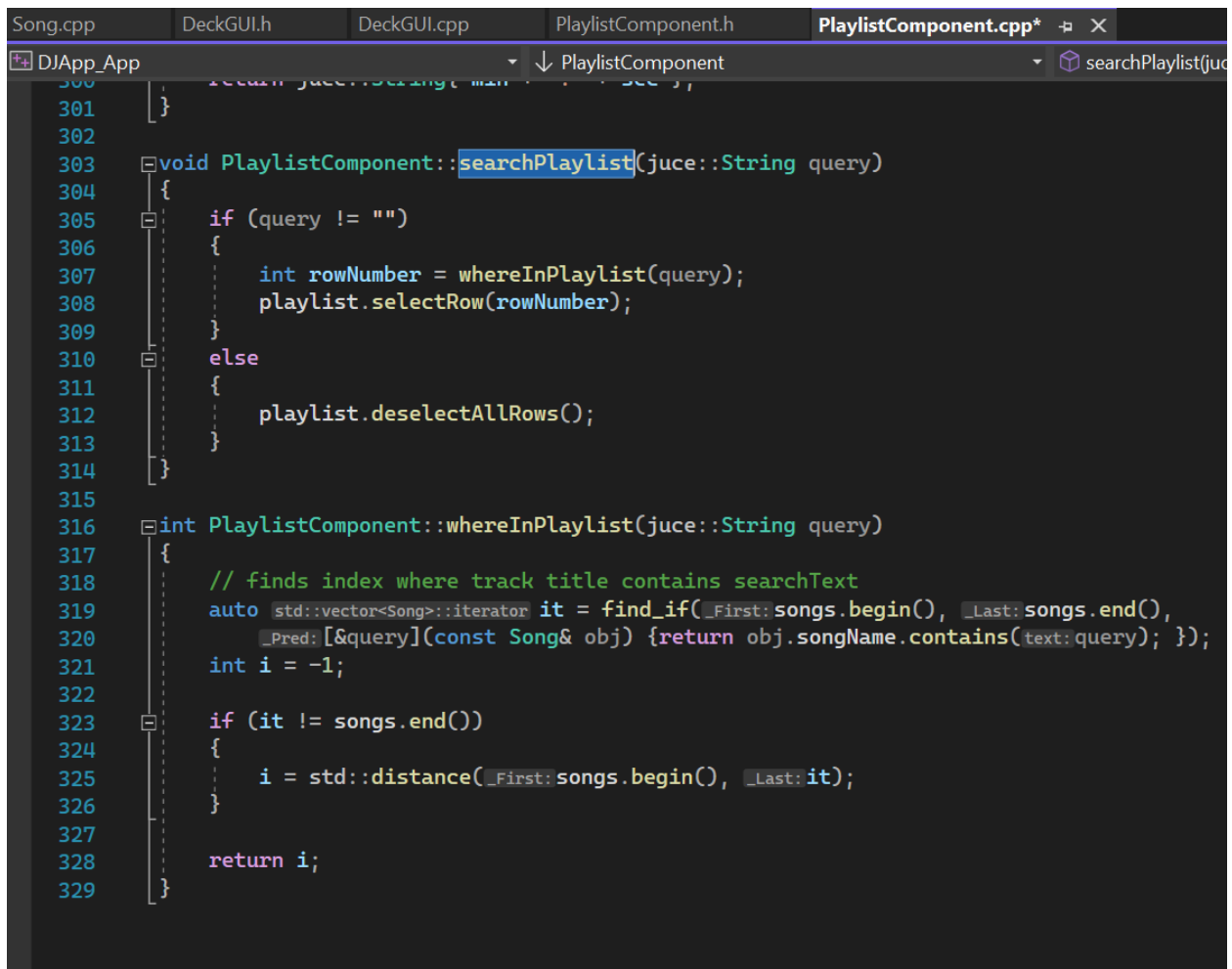
```

Song.cpp  DeckGUI.h  DeckGUI.cpp  PlaylistComponent.h  PlaylistComponent.cpp*
DJApp_App
24      addAndMakeVisible(& child: tableComponent);
25      tableComponent.getHeader().addColumn(columnName: "Track title", columnId: 1, width: 400);
26
27      // search songs
28      addAndMakeVisible(& child: searchBox);
29      searchBox.setTextToShowWhenEmpty(text: "Search for song name and press Enter.",
30                                     colourToUse: juce::Colours::lightcoral);
31      searchBox.onReturnKey = [this] { searchPlaylist(query: searchBox.getText()); };

```

Figure 25 - searchPlaylist function trigger

Figure 26 displays the content of the searchPlaylist function. If the user did not type anything, all rows of the playlist are deselected. Otherwise, the song name is searched via the 'whereInPlaylist' function, which returns the index corresponding to the songs array position where the song title is found. If no match is found, -1 is returned. The row containing the song searched by the user is highlighted (background of the cell is painted in coral color).



```

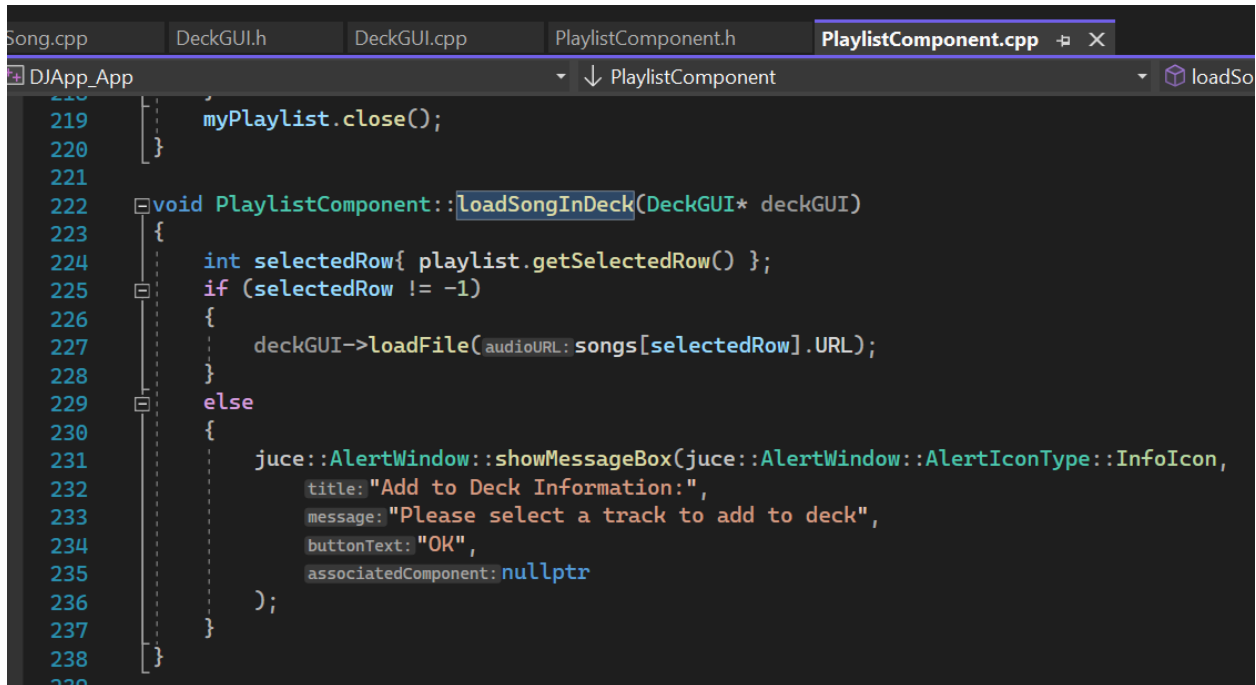
Song.cpp  DeckGUI.h  DeckGUI.cpp  PlaylistComponent.h  PlaylistComponent.cpp*
DJApp_App
300
301 }
302
303 void PlaylistComponent::searchPlaylist(juce::String query)
304 {
305     if (query != "")
306     {
307         int rowNumber = whereInPlaylist(query);
308         playlist.selectRow(rowNumber);
309     }
310     else
311     {
312         playlist.deselectAllRows();
313     }
314 }
315
316 int PlaylistComponent::whereInPlaylist(juce::String query)
317 {
318     // finds index where track title contains searchText
319     auto std::vector<Song>::iterator it = find_if(_First: songs.begin(), _Last: songs.end(),
320         _Pred: [&query](const Song& obj) {return obj.songName.contains(text: query); });
321     int i = -1;
322
323     if (it != songs.end())
324     {
325         i = std::distance(_First: songs.begin(), _Last: it);
326     }
327
328     return i;
329 }

```

Figure 26 - searchPlaylist and whereInPlaylist functions

### R3D: Component allows the user to load files from the library into a deck

As we can see in Figure 20, two instances of class `juce::TextButton` are created within `PlaylistComponent` at lines 132 and 133: `addSongToLeftDeckButton` and `addSongToRightDeckButton`. Within `PlaylistComponent.cpp`, at lines 57 and 58 a listener is attached to each button. Thanks to this listener, function `loadSongInDeck` is called every time the user clicks on any of these buttons (lines 175 and 179, Figure 21). `loadSongInDeck` is called passing as argument the deck to which we're adding the song.



```

219     myPlaylist.close();
220 }
221
222 void PlaylistComponent::loadSongInDeck(DeckGUI* deckGUI)
223 {
224     int selectedRow{ playlist.getSelectedRow() };
225     if (selectedRow != -1)
226     {
227         deckGUI->loadFile(audioURL: songs[selectedRow].URL);
228     }
229     else
230     {
231         juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
232             title: "Add to Deck Information:",
233             message: "Please select a track to add to deck",
234             buttonText: "OK",
235             associatedComponent: nullptr
236         );
237     }
238 }
239

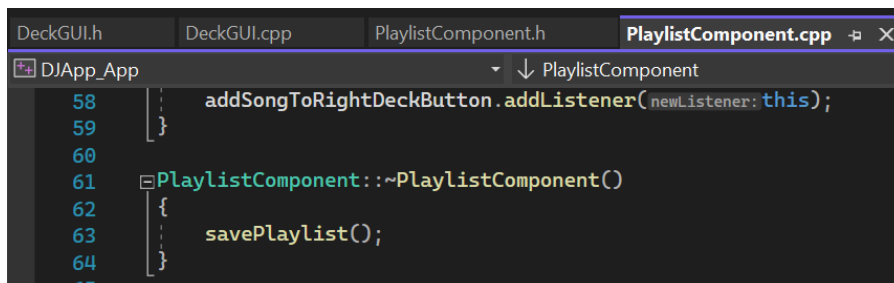
```

Figure 27 - `PlaylistComponent::loadSongInDeck`

Figure 27 shows the content of this function. If the user has selected a song from the playlist and then clicked on the 'add to left' or 'add to right' buttons, the song will be loaded to the corresponding deck. Otherwise, a warning message tells the user to select a song before clicking these buttons.

### R3E: The music library persists so that it is restored when the user exits then restarts the application

The destructor of class `Playlist` component (figure 28) is called every time the objects of this class are destroyed i.e when the app is closed.



```

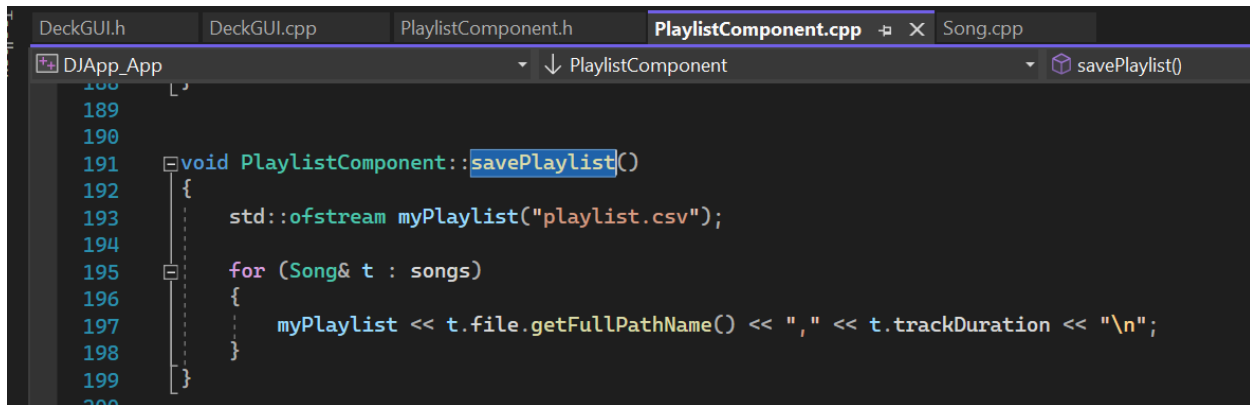
58     addSongToRightDeckButton.addListener(newListener: this);
59 }
60
61 PlaylistComponent::~PlaylistComponent()
62 {
63     savePlaylist();
64 }
65

```

Figure 28 - `PlaylistComponent::~PlaylistComponent()`



As we can see, the destructor calls function `PlaylistComponent::savePlaylist()`, which creates a file called 'playlist.csv', containing a row for each song in the playlist. Each row contains the song full path and the track duration, as we can see in figure 30.



```

188
189
190
191 void PlaylistComponent::savePlaylist()
192 {
193     std::ofstream myPlaylist("playlist.csv");
194
195     for (Song& t : songs)
196     {
197         myPlaylist << t.file.getFullPathName() << "," << t.trackDuration << "\n";
198     }
199 }
200

```

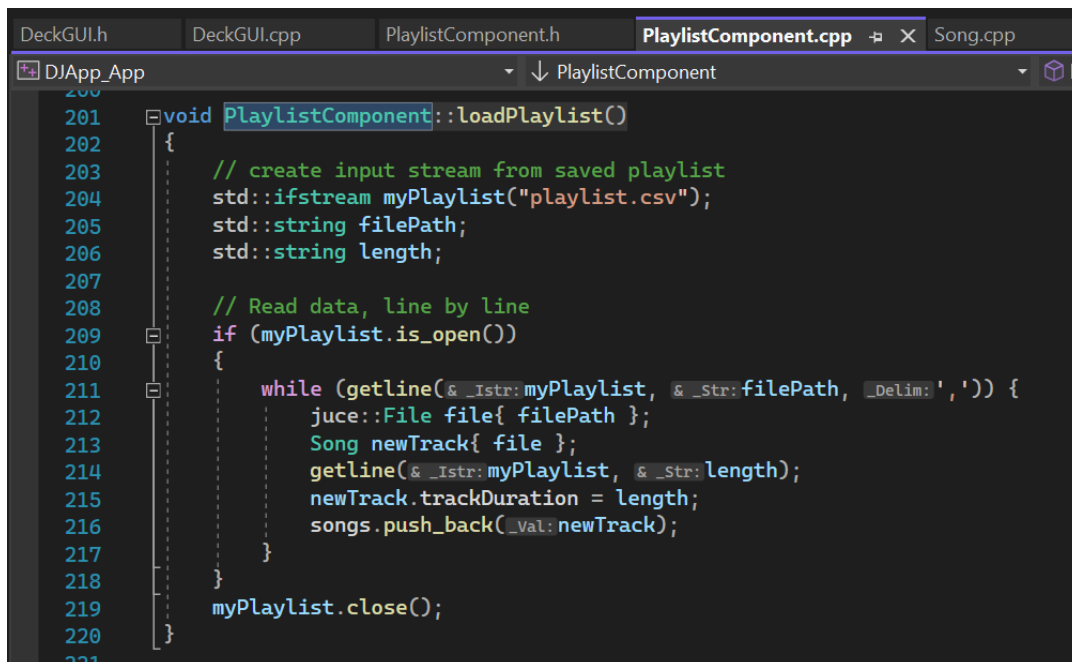
Figure 29 - `PlaylistComponent::savePlaylist()`

C:\Users\ventafri\Desktop\Uni\ProdJUCer\tracks\bad_frog.mp3	8:01
C:\Users\ventafri\Desktop\Uni\ProdJUCer\tracks\stomper_reggae_bit.mp3	1:28
C:\Users\ventafri\Desktop\Uni\ProdJUCer\tracks\hard.mp3	0:13

Figure 30 - `playlist.csv` sample content

Hence, when the user closes the app, the playlist is saved.

When the user open the app, from within the `PlaylistComponent` constructor, function `PlaylistComponent::loadPlaylist()` is called, which reads the content of this .csv file and loads each song back into the playlist (figure 31).



```

200
201 void PlaylistComponent::loadPlaylist()
202 {
203     // create input stream from saved playlist
204     std::ifstream myPlaylist("playlist.csv");
205     std::string filePath;
206     std::string length;
207
208     // Read data, line by line
209     if (myPlaylist.is_open())
210     {
211         while (getline(&_Istr:myPlaylist, &_Str:filePath, _Delim:',')) {
212             juce::File file{ filePath };
213             Song newTrack{ file };
214             getline(&_Istr:myPlaylist, &_Str:length);
215             newTrack.trackDuration = length;
216             songs.push_back(_Val:newTrack);
217         }
218     }
219     myPlaylist.close();
220 }
221

```

Figure 31 - `PlaylistComponent::loadPlaylist()`

## R4: Implementation of a complete custom GUI

My OtoDecks GUI is significantly different from the one seen in class (Figure 32). I have changed the layout, added custom knobs by leveraging [this Git repository](#), changed the look and feel of the application by changing the color scheme (coral [color](#)), and added functionality.

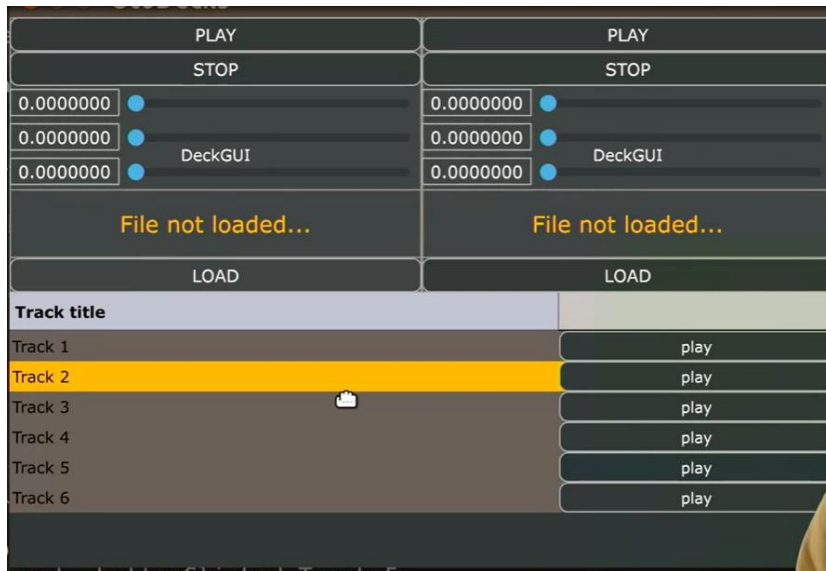


Figure 32 - Application developed in class



Figure 33 - The new application

## R4A, B, C: Layout is different, it includes custom components, and music library

As we can see by comparing Figure 32 and 33, the application appearance has been significantly changed. These are the main differences:

- the playlist is now in the middle, in-between the two decks
- the color scheme is different as I have customized the look and feel of the basic components to be using a coral color scheme.
- custom knobs are used. These are rotary sliders with custom appearance.

In addition to the above-mentioned layout changes, I have added the following functionality:

- Slider to control the track position, right below the waveform display
- Rewind and fast forward buttons for each deck
- Two additional knobs/ sliders on top of volume and speed: these modify two reverb properties. Wet level and freeze.
- The buttons are changed from being instances of the TextButton class to being instances of the ImageButton class. This way, it was possible to add the typical images for track related controls (play, rewind, fast forward).