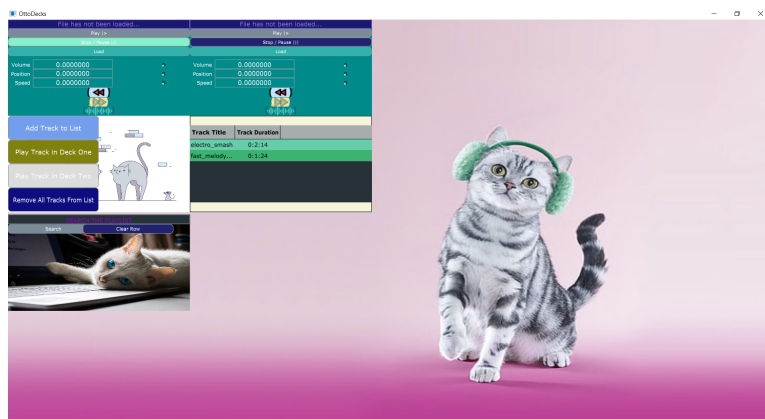


OTTO DECKS: A DJ APPLICATION WRITTEN IN JUCE

TABLE OF CONTENTS

[Purpose of the Application](#)
[Project Documentation with DoxyGen](#)
[Renaming of Files](#)
[Addition of New Files / Components](#)
[R1 Requirements](#)
[R2 Requirements](#)
[R3 Requirements](#)
[R4 Requirements](#)
[Personal Reflections](#)



The following image depicts the final version of the OttoDecks application. It went through a couple of iterations before the cat theme was chosen as the final look. Why? Because cats are *awesome*.

Purpose of the Application

Otto Decks is an audio application that was created using the JUCE framework: the underlying programming language of the framework is C++. More information on the Juce framework can be found [here](#).

The main purpose of Otto Decks is to simulate the real behaviour of a DJ. This means that audio tracks can be loaded into players, their volumes adjusted and that a custom playlist can be created and saved for later use. In the same vein, the contents of the playlist can also be deleted, although more advanced features such as deleting one track separately rather than all at once have not been implemented at the current time of writing.

Project Documentation with DoxyGen

The project documentation can be found on the following [Github page](#); the documentation provides an overview of the classes and their respective methods. Diagrams for each of the individual components have also been generated in order to visualise the inheritance relationships between specific components.

While the source code for the header files is available with links embedded to relevant documentation, no access has been given to the cpp files. In order to generate the html files, [DoxyWizard was used to create](#) Doxygen-related files from the existing source code; however, certain tweaks have been made to the CSS and HTML to give a more individual touch to the web site. DoxyGen was chosen because of its ease of use.

Throughout this report – where possible – links have been added to the specific documentation of a class and its methods; code snippets are omitted in favour of explaining the working of the implementation via the documentation or a reference to the source material that inspired it.

As an alternative to the website, the documentation is also provided in the zipped folder submitted to the Coursera platform, with the added caveat that the links on this report are only targeted towards the online instance of the documentation. Last but not least, all of the documentation should also be found in the source code itself, with all of the header files containing a description of their purpose, along with the methods being commented on in terms of their function.

Renaming of Files

Some of the classes / components have been renamed in order to better reflect their purpose. For example, the control decks controlling the two audio players are represented by the [ControlDeck](#) component. Originally these decks were referred to as *OttoDecks*. The new name was chosen as the word 'control' reflects the functionality of this component. In other words, it implies that this component handles the way the audio tracks are manipulated using the available buttons in the UI.

The control deck that loads and assigns a track from the playlist into one of the two audio players is called [PlaylistControlDeck](#). In the course lecture, this component was called *PlaylistComponent*. As with the *ControlDeck*, the new name was chosen as it offers a hint of what this component accomplishes as well as which part of the application it is targeted towards (i.e the playlist). More specifically, it implies that this component not only allows files from the playlist to be played, but also deals with the deletion and addition of said files to the playlist table.

The handling of the playlist is managed by the [PlaylistManager](#). Previously, this component was part of the *PlaylistComponent*, but the decision was made to break down the functionality into two components; this was done in order to separate differing coding logic from each other.

Addition of New Files / Components

In addition to the renaming of files, three new files were added to OttoDecks in order to enhance the functionality of the application. These files are:

- ★ The [AudioMetaData](#) class which contains metadata related information about a track.
- ★ The [SearchField](#) component which contains all logic related to searching for items in the playlist.
- ★ The [TrackFile](#) class which retrieves information such as the length name of a file.

R1 Requirements

This section of the report provides an overview of the basic functionality that was worked on during the lecture; it also includes additional features created for the final project.

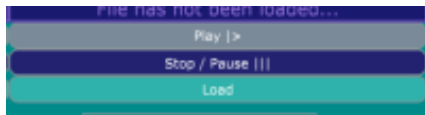
Requirements Table

The following table provides references as to how each of the R1 prerequisites have been met. The methods and classes mentioned should serve as a guideline as to where to find a given requirement. Links have been provided to the classes' documentation.

Requirements	Relevant Method(s)	Relevant Class(es)
R1A: can load audio files into audio players.	loadURL(URL& audioURL) buttonClicked(Button* button)	DJAudioPlayer ControlDeck
R1B: can play two or more tracks.	playSong() buttonClicked(Button* button)	DJAudioPlayer ControlDeck
R1C: can mix the tracks by varying each of their volumes.	setGain(double gain) buttonClicked(Button* button)	DJAudioPlayer ControlDeck
R1D: can speed up and slow down the tracks.	setSpeed(double ratio) buttonClicked(Button* button)	DJAudioPlayer ControlDeck

Description of Features: Buttons and Sliders

The basic functionality of the Otto Decks application can be broken down into three buttons and sliders. Each of these buttons and sliders have specific tasks associated with them as discussed in the paragraphs below.



The table below displays the functionality of the buttons, along with some relevant implementation details.

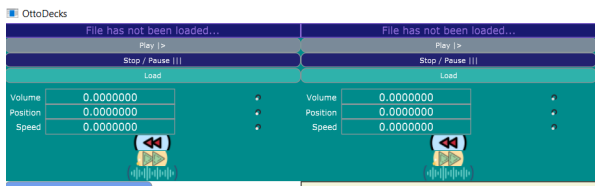
Functionality	Implementation Details
Playing or resuming the audio of a track.	An instance of <code>AudioTransportSource</code> is used to call the start method within that class .
Stopping or pausing the track of an audio file.	An instance of <code>AudioTransportSource</code> is used to call the stop method within that class .
Uploading an audio file into one of the two available players.	By clicking the <code>loadButton</code> in the ControlDeck , the <code>load</code> function in the DJAudioPlayer component is called.



The table below provides an overview of how the sliders' functionality was implemented.

Functionality	Implementation Details
The volume of the audio.	An instance of <code>AudioTransportSource</code> is used to call the gain method within that class .
The position of the audio.	An instance of <code>AudioTransportSource</code> is used to call the position method within that class .
The speed of the audio.	An instance of <code>ResamplingAudioSource</code> is used to call the setResamplingRatio method .

Description of Features: The ControlDeck Component



This component is in charge of mapping each button to the appropriate function call that deals with the relevant audio effect. These functions are all located in the [DJAudioPlayer](#) component.

Functionality	Implementation Details
If the play button is clicked, it triggers the <code>playSong</code> function.	The <code>buttonClicked</code> method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.
If the load button is clicked, it calls the <code>loadURL</code> function.	The <code>buttonClicked</code> method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.

If the stop button is clicked, it kicks off the stopSong function.	The buttonClicked method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.
Each time a slider's value is changed, it communicates the desired audio effect to the relevant function in DJAudioPlayer.	The ControlDeck contains a method called sliderValueChanged which takes in a pointer to a specific slider as an argument and then checks whether it has been altered. If the value of a slider has been changed, it assigns that action with a specific action in DJAudioPlayer .

Description of Features: The DJAudioPlayer Component

This component deals with the audio and the various ways its playback can be controlled / manipulated. The table below highlights its functionality as well as any necessary implementation details.

Functionality	Implementation Details
Loads a track from a given url which is often a directory.	This is handled by the AudioFormatManager class.
Plays and stops a track.	These methods are dealt with in AudioTransportSource .
Sets the volume, speed and gain of a track.	These methods are dealt with in AudioTransportSource , with the exception of speed which is handled by ResamplingAudioSource .
Deals with all the required audio processing.	This is all done by AudioSource , which is the base class for the ones mentioned above.

R2 Requirements

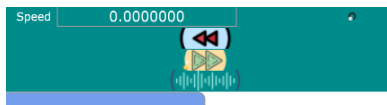
This section deals with the custom control deck that was created for the final project.

Requirements Table

The following table provides references as to how each of the R2 prerequisites have been met. The methods and classes should serve as a guideline as to where to find a given requirement. Links have been provided to the classes' documentation.

Requirements	Relevant Method(s)	Relevant Class(es)
R2A: Component has custom graphics implemented in a paint function.	paint(juce::Graphics& graphics) repaintButtons(); repaintSliders();	All of these methods belong to the ControlDeck component.
R2B: Component enables the user to control the playback of a deck somehow.	void rewindSong(); void fastForwardSong(); void startLoop(bool buttonsOn) buttonClicked(Button* button)	The first three methods in the previous row belong to DJAudioPlayer , while buttonClicked is part of the ControlDeck .

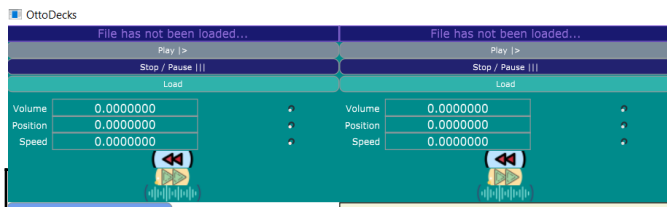
Description of the Features: New Buttons



In addition to the basic controls, three new buttons have been added; rather than using text buttons, image buttons are now employed. Building upon the design of the original buttons, they contribute to the Otto Decks application as listed in the table below.

Functionality	Implementation Dets
The rewinding of a track.	The buttonClicked method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.
The fastforwarding of a track.	The buttonClicked method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.
The looping of a track.	The buttonClicked method in ControlDeck calls the relevant audio effect in DJAudioPlayer , via a pointer. This is done once the relevant button has been clicked.

Description of the Features: The ControlDeck Component



The table below lists the functionality of this component as well as some relevant implementation details.

Functionality	Implementation Details
Assigning each button to its correct DJAudioPlayer function.	The buttonClicked method - which is an override of Button::EventListener - in ControlDeck uses conditional statements to differentiate each button reference from one another in order to call upon the right method of the DJAudioPlayer pointer.
The ability to change the colours of the sliders and buttons depending on their state of activity.	The ControlDeck contains a method called repaintButtons / repaintSliders, which – depending on certain mouse conditions – changes the colour of a respective button, using if / else statements to differentiate between states.
Custom designs such as making the slides rotary and adding labels next to them.	Rather than customising the LookandFeel class, the setSliderStyle method in the Slider class was used to accomplish this.

Description of the Features: The DJAudioPlayer Component

Beyond the basic functionality, this component also deals with the following audio-related tasks as displayed in the table below. The table also provides some relevant implementation details.

Functionality	Implementation Details
Rewinding or fast forwarding a song via the help of the track's current position and then adding / subtracting from it.	Both methods take the current position of the AudioTransportSource into account and then – depending on whether the track is to be moved forward or backwards – subtract or add a specific value from it.
Looping a track in the case that the relevant button has been clicked.	When the loop button has been triggered, the AudioTransportSource's isLooping method will be set to true .
Getting the file length in seconds, which is important for displaying the file length of a track in the playlist.	The implementation for formatting the audio's length in a certain manner was inspired by the following piece of code .

R3 Requirements

This section is concerned with the playlist and the manner in which it has been implemented. The implementation of the playlist is

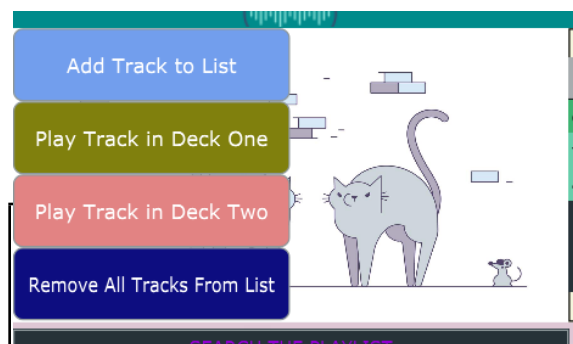
one of the most complex in the entire Otto Decks application; it contains three components and two classes. The classes serve as placeholders for specific audio-related properties, such as the track file's length or the audio's metadata information that can be retrieved from the player. Meanwhile, the components are representations of various GUI parts.

Requirements Table

The following table provides references as to how each of the R3 prerequisites have been met. The methods and classes should serve as a guideline as to where to find a given requirement. Links have been provided to the classes' documentation.

Requirements	Relevant Method(s)	Relevant Class(es)
R3A: Component allows the user to add files to their library.	juce::Array<juce::File> PlayListControlDeck::loadInTracks()	PlayListControlDeck
R3B: Component parses and displays meta data such as filename and song length.	addTrackToTracksVector(TrackFile& trackFile)	PlayListManager
R3C: Component allows the user to search for files.	searchThePlaylist(juce::String& inputtedText)	PlayListManager
R3D: Component allows the user to load files from the library into a deck.	addTrackToPlayerOne() addTrackToPlayerTwo()	PlayListManager
R3E: The music library persists so that it is restored when the user exits then restarts the application.	saveTracksFile()	PlayListManager

Description of the Features: The PlayListControlDeck Component

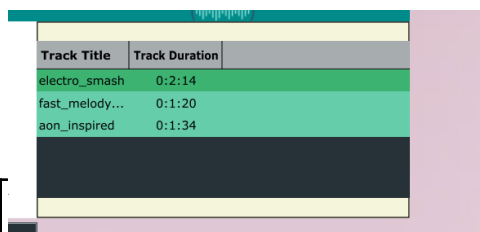


The main objective of this component is the creation of the tracks.txt file; it is also in charge of assigning each button to a specific role. The table below lists the functionality and some relevant implementation details.

Implementation Details	
Functionality	
Adding a track to the playlist table via loading it from a directory.	The PlayListControlDeck contains a method called loadInTracks that filters the search for files to mp3 ones and then, using the FileChooser class, applies the browseForMultipleFilesToOpen function to load in a particular track. This method is triggered when the loadTrack button has been clicked – the check as to whether it has been clicked is performed by the buttonClicked function.
Playing a selected track in the playlist in either of the available audio players.	The PlayListControlDeck contains two buttons called loadToDeckOne and loadToDeckTwo. The buttonClicked function checks whether these buttons have been clicked or not – if one of them has, a pointer to PlayListManager calls either the addTrackToPlayerOne or addTrackToPlayerTwo method.
Removing all tracks from the playlist	The method removeAllContentsFromFile in PlayListControlDeck is triggered when the button removeTrack is clicked. Whether it has been clicked or not, is checked by the buttonClicked method. If removeTrack has been activated, then the tracks.txt file is opened and its content is truncated. A method in the PlayListManager is also called in order to deselect all the rows in the playlist table.

Determines how buttons are displayed depending on their state of activity.	The repaintButtons method in PlayListControlDeck changes the look of the buttons depending on their state.
Populates the tracks by writing them into a txt file.	The populateTracksFile function in PlayListControlDeck opens a text file and – depending on whether the track is already in the file – writes the aforesaid track into it.
Paints the buttons and backgrounds onto the screen	The paint and resize methods in the PlayListControlDeck component are responsible for drawing the image of the cat background on the screen as well as determining how the buttons are displayed.

Description of the Features: The PlayListManager Component

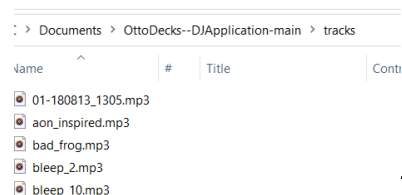


Track Title	Track Duration	
electro_smash	0:2:14	
fast_melody...	0:1:20	
aon_inspired	0:1:34	

The following component is tasked with handling the contents of the playlist table. It also has four buttons that deliver the following duties as listed in the table below.

Functionality	Implementation Details
Saves the tracks to a vector which is used as a container to store them in one place.	Using the addTrackToTracksVector method in the PlayListManager , a reference of TrackFile is passed into it; information from this reference – such as the url of the file – is extracted and then used to populate the tracks vector.
Loads the tracks from a txt file into the application.	The PlayListManager contains a method called loadTracksFile that opens a text file, gets the necessary information from it and also sets the file length. This information is then pushed into a tracks vector that is then loaded into the playlist table.
Saves the tracks into a txt file if the application has been closed.	The saveTracksFile method in the PlayListManager iterates through the available tracks in the tracks vector and saves each track in a text file.
Determines which audio player a track is assigned to.	The methods addTrackToPlayerOne and addTrackToPlayerTwo, by using pointers to the ControlDeck, each load a track to an assigned deck by calling the loadDroppedTrack in ControlDeck . This method itself uses a pointer to the DJAUDIOPlayer to load a file. A pointer to the WaveFormDisplay is also used in order to display a waveform while the selected track is being played.
Searches the track vector for a given input string.	The PlayListManager contains a method called searchThePlaylist. This particular function loops through the tracks vector and makes note of any item that matches the inputted text.

Description of the Features: The TrackFile class



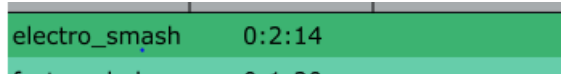
Name	#	Title	Cont
01-180813_1305.mp3			
aon_inspired.mp3			
bad_frog.mp3			
bleep_2.mp3			
bleep_10.mp3			

This class deals with tasks such as the retrieval of track file specific properties like the file name. Moreover, the [TrackFile](#) also contains the following information about a track as displayed in the table below.

Functionality	Implementation Details
The file length, url and name in	The TrackFile contains getter methods for the file name and url, which are determined when an instance of it is created. In other words, they are created when the constructor of TrackFile

order to use them for the playlist.	is called. The file length contains a setter method as the length can only be calculated from the audio track's metadata.
Specific file properties such as the entire file path.	The method <code>getTrackFileProperties</code> returns an instance of a <code>TrackFile</code> where the constructor takes in an instance of juce File class . This class contains methods that return the entire file name, just to name an example.

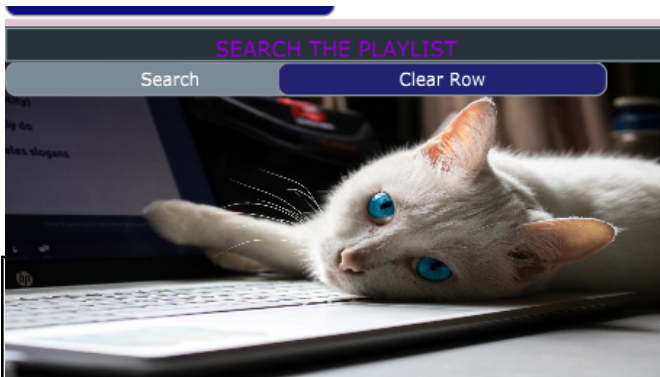
Description of the Features: The `AudioMetaData` class



The [AudioMetaData](#) class supervises the retrieval of audio-related metadata such as the length of a track file. The table below lists the functionality of this component as well as some relevant implementation details.

Functionality	Implementation Details
Get the audio track length by using a pointer to the <code>DJAudioPlayer</code> component in order to call the relevant function.	The AudioMetaData class contains a method called <code>getAudioTrackLength</code> that references the <code>getFileLengthSeconds</code> function in DJAudioPlayer . This method uses an instance of <code>AudioTransportSource</code> in order to use its <code>getLengthInSeconds</code> function.
Format the audio track into the desired string representation.	The AudioMetaData contains a method called <code>getFormattedAudioString</code> that converts an argument representing a double in seconds into a string – broken down into small strings to depict hours, minutes and seconds. These string subsets are then formatted into an appropriate form.

Description of the Features: The `SearchField` component



The [SearchField](#) component displays the search field and deals with how the search result is depicted on the playlist. The table below lists the functionality and some relevant implementation details.

Functionality	Implementation Details
Retrieves the result of a search and then displays it on the table by selecting the relevant row.	The SearchField component's <code>searchThroughPlaylist</code> method contains a pointer to the PlayListManager and its function, <code>searchThePlaylist</code> . This particular function loops through the tracks vector and makes note of any item that matches the inputted text.
Sets up general search field related properties such as the font size.	The <code>setUpSearchFieldProperties</code> function in SearchField sets up properties such as the input restrictions, the font that is to be applied to the text that appears before input is entered into the search field and the way the text is displayed in the UI.

Paints the custom background image onto the screen.	In order to customise the OttoDecks application further, the paint method in the SearchField component draws an image of a cat next to a laptop.
---	--

R4 Requirements

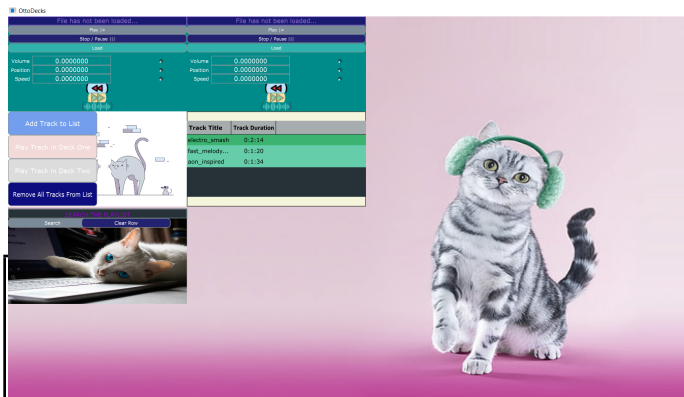
This section deals with the custom GUI that was created for the final project. The layout went through numerous changes, taking feedback from my partner and friends into account. While the layout is not the most user-friendly, the decision was made to make it as original as possible, with the theme of cats being a determining factor in its final appearance.

Requirements Table

The following table provides references as to how each of the R4 prerequisites have been met. The methods and classes should serve as a guideline as to where to find a given requirement. Links have been provided to the classes' documentation.

Requirements	Relevant Method(s)	Relevant Class(es)
R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls.	All of these requirements are handled by the paint and resize methods in the relevant components.	PlayListControlDeck PlayListManager MainDJAudioManager
R4B: GUI layout includes the custom Component from R2.	All of these requirements are handled by the paint and resize methods in the relevant components.	ControlDeck
R4C: GUI layout includes the music library component from R3.	All of these requirements are handled by the paint and resize methods in the relevant components.	PlayListControlDeck PlayListManager MainDJAudioManager

Description of the Features: Custom GUI Layout



The Otto Deck layout consists of a few building blocks, which each contribute to the whole look of the application. These components are listed in the table below, along with some implementation details that were relevant to the creation of the layout. .

Functionality	Implementation Details
The two control decks, positioned next to each other.	The two control decks are brought together / linked by the MainDJAudioManager's resize method, which is responsible for the final display of all the GUI components.
The PlayListControlDeck and the playlist table, which are underneath the two decks.	The PlayListControlDeck is responsible for delegating buttons to certain events, while the PlayListManager determines how tracks are displayed.
A search field that is displayed underneath the PlayListControlDeck.	All of the code that is relevant for this lives in the SearchField component.
A search and clear button that are	All of the code that is relevant for this lives in the SearchField component.

underneath the search text field.	
A custom background that displays a cat with headphones.	This was achieved by using the getFromMemory method provided by the ImageCache class in Juce.
A custom background incorporated within the PlayListControlDeck that shows an angry cat.	This was achieved by using the getFromMemory method provided by the ImageCache class in Juce.
A custom background displayed within the search field that shows a white cat and a laptop.	This was achieved by using the getFromMemory method provided by the ImageCache class in Juce.

Personal Reflections

Upon the end of a project, I seldom reflect on whether it was a successful venture or not – the overall relief and satisfaction upon having finished it tend to be the more prevalent emotions. However, during the process of writing this document, I did notice a few things that I think went well, while also being able to dwell upon some less positive experiences.

In terms of negative experiences, I feel that the creation of this report was daunting – coming from a humanities background, I am new to technical writing and have had to curb my enthusiasm – so to speak – when it comes to using metaphors or hyperboles. While I tried avoiding sounding too dry, I do hope that this report has a professional tang to it. Moreover, I have also had a hard time to stop myself from constantly proofreading and altering this report.

In addition to the writing of the report, the video was also a bit of a fiasco. I have had to take multiple reshoots as talking in front of an audience makes me nervous, even if I am merely recording myself. I also hate the sound of my own voice. Other negative experiences were the discovery of bugs while recording said video, which was annoying as I had to go back and fix them, rendering the entire video recording process moot.

Positive experiences for me were the creation of the project itself – I loved dabbling with the code, especially in terms of refactoring. There is something beautiful in seeing an application come to life in small iterations, and I am very happy to have been given the opportunity to work on a DJ app. While I will never be a professional DJ, I feel that OttoDecks has taught me a lot in terms of programming.