Otodecks
End-Term coursework for Object Oriented Programming – University of London

By David Della Rossa

What is OtoDecks

OtoDecks is a simple application for DJs.

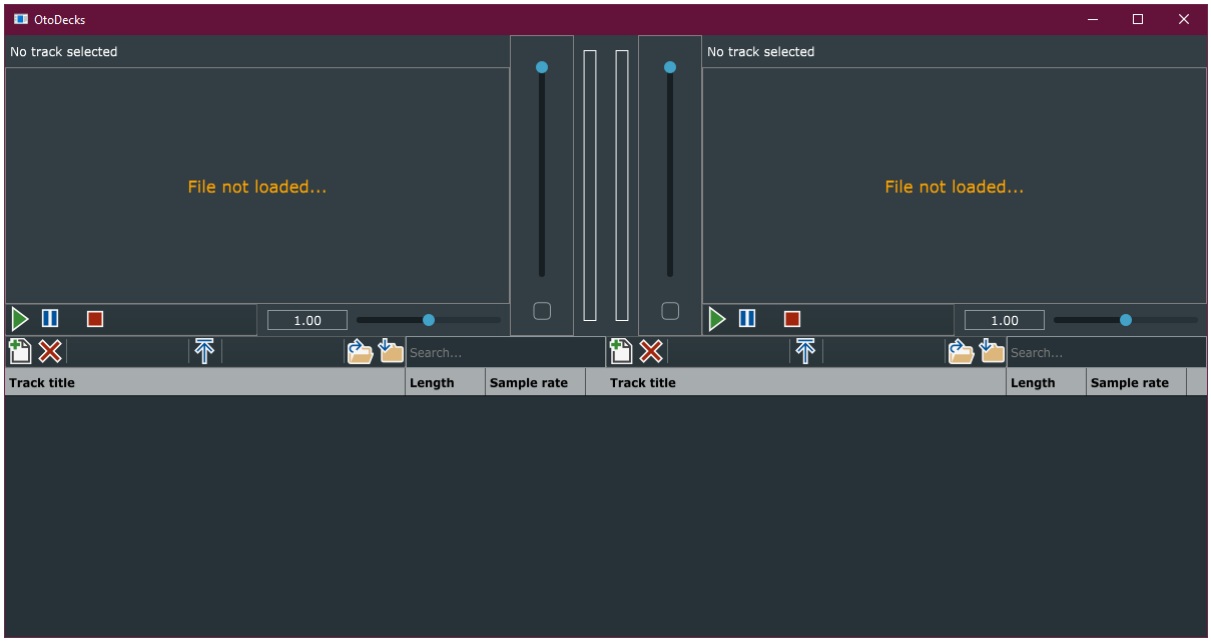

Figure 1:OtoDecks application

It presents two identical decks side by side, where the user can load audio files, play them, control volume, speed of execution and playback position.

For each deck, there is a playlist where the user can enqueue audio tracks and have them ready to play. These playlists can be saved on the hard drive and loaded at later convenience.

Playlists support drag-and-drop, so that external files can be loaded into the playlist just dragging them on. From the playlist, these tracks can be loaded, one at a time, into the player and be played.

Figure 2 shows the parts that each deck is made of. These parts will be described in more detail in the following paragraphs.

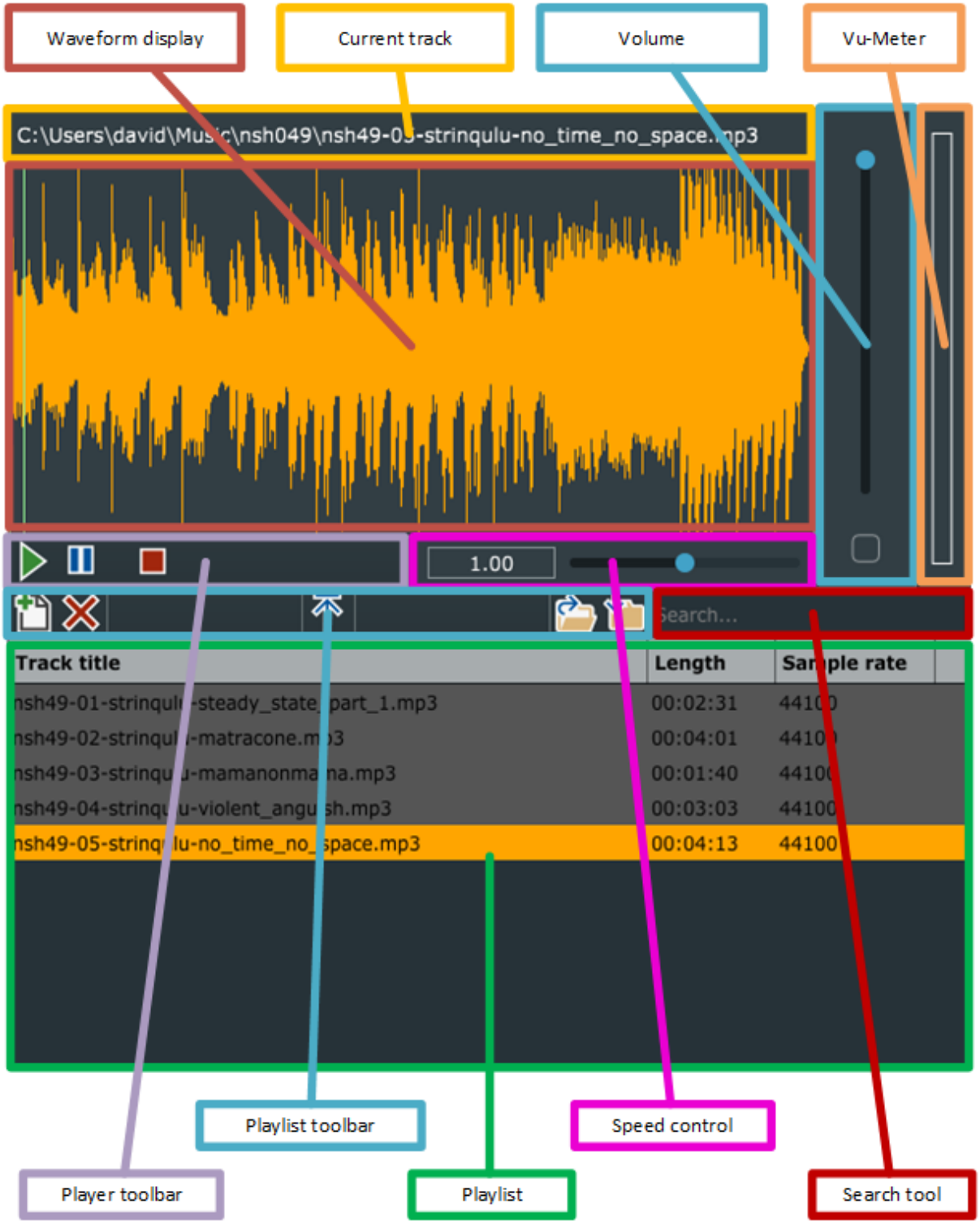

Figure 2: Deck structure

Current track

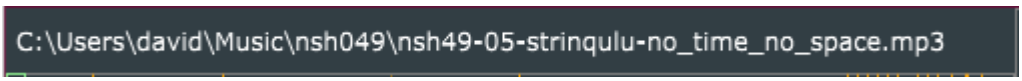

Figure 3: Current track display

This control shows the full path of the track currently loaded in the player. When the user loads a track in the player, either from the playlist or by drag-and-dropping an external file, this label is updated with the full path of the loaded track.

Waveform Display

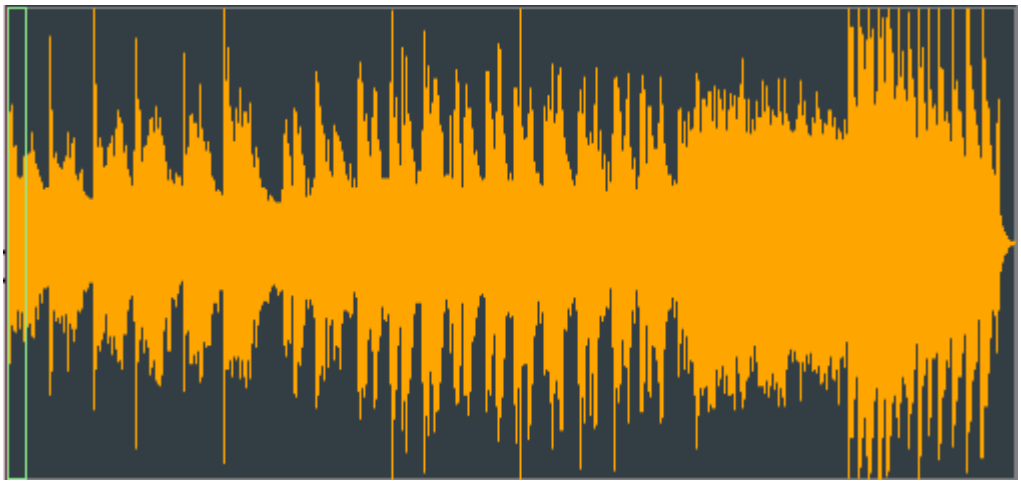

Figure 4: Waveform display

This widget displays the Waveform of the track currently loaded in the player.
It is an evolved version of the Waveform display control built during the course.

The addition I made to this control is the integration of the position slider to control the playback position. Now it is possible to drag around the vertical green rectangle that marks the current position in the track. This causes the current playing position of the track to change in real-time.

Volume

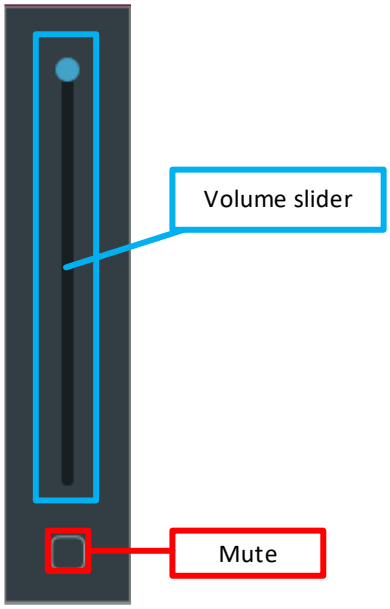

Figure 5: Volume control

This widget controls the volume of the player it is associated with.

The Mute button is a toggle button which mutes/unmutes the track instantly.

Vu-Meter

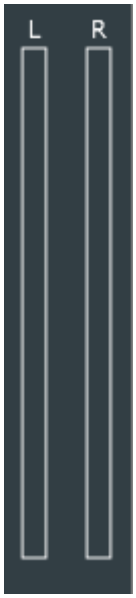

Figure 6: Vu-Meter display

First important thing to note is that this widget has no dependencies with any of the two players.

It displays the audio level of the final sound going out to the speakers.

As the labels show, one bar is for the Left channel, and measures the total sound of both players sent to the Left channel. The other bar is for the Right channel.

This widget receives, in real time, information about the current level of the audio sample being played and builds a coloured bar whose height is proportional to the audio level received.

The coloured bar is divided into 4 different colours.

- A green bar, for audio levels from 0% to 50% of the maximum available level;
- A yellow bar, for audio levels from 50% to 66.66%;
- An orange bar, for audio levels from 66.66% to 83.33%;
- A red bar, for audio levels from 83.33% to 100%

Note that it uses the RMS level, which is a measure of the average sound.

The `VuMeter` class also uses a `gain` factor to change the proportionality factor between the height of the bar and the audio level. At the moment this extra gain is set to 2. This is to accentuate the course of the bars.

This gain variable is defined in `VuMeter.h`, line 116:

```
const float gain = 2;
```

Player toolbar

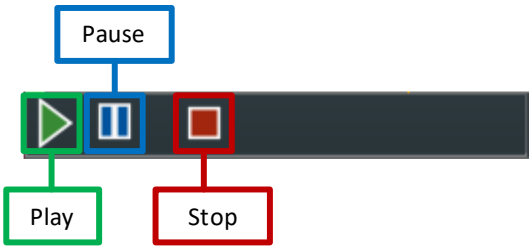

Figure 7: Player toolbar

This toolbar is meant to control the player it is associated with.

It shows three buttons:

- **PLAY**: starts the playback of the current track. If no track is loaded, or the current track is already playing, nothing happens.
- **PAUSE**: interrupts temporarily the playback. Clicking the Pause button multiple times has no effect. In order to restart the playback, click on the Play button, the playback will start from the position where it got paused.
- **STOP**: stops the playback. It resets the playback position to start, so that, clicking play, will start the reproduction all over again.

Speed control

Figure 8: Speed control

This widget controls the speed of reproduction of the track. The Label displays values from 0 to 2, centred at 1. However, consider that the effect on the speed is exponential, where *0 corresponds to 1/10th of the original speed, 1 corresponds to the original speed, 2 corresponds to 10 times the original speed*.

Playlist

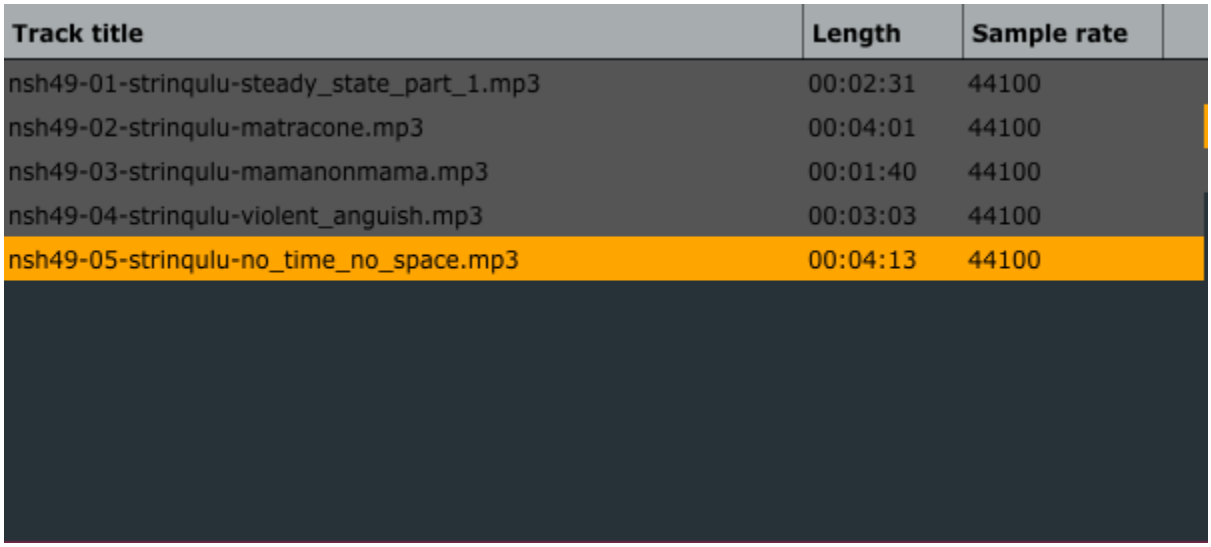

Figure 9: Playlist control

The playlist widget shows a list of pre-chosen tracks.

Tracks can be loaded into the playlist in two ways:

- *Using the* **ADD FILES** *button* in the playlist toolbar and following the procedure;
- *Dragging audio files from outside the application onto the library*. This has the advantage of using the Operative System functionalities of file search to retrieve the desired tracks and load them all at once, selecting them and dragging them onto the playlist.

The Add file functionality checks that the files about to be imported are all audio files of the types supported by the application. If not, the operation will not be allowed.

Once the files are imported into the library, the audio information is extracted from the file and a `TrackModel` object is created, containing all the relevant information, such as *Length*, *Sample Rate* and more.

Files can be removed from the playlist using the **REMOVE FILE** button from the Playlist toolbar. Files are not permanently deleted, of course, they are just removed from the playlist.

The Remove File functionality allows to remove only one file at a time. To do so, the file has to be selected first in the playlist and finally removed.

In order to load a file from the playlist into the player there are two ways:

- Select the file in the playlist and *click on the* **LOAD FILE IN PLAYER** *button* from the Playlist toolbar.
- *Double-click on the file in the playlist* that we want to load in the player.

In addition, a single file can be loaded directly into the player, without passing from the library, dragging and dropping from outside the application, right onto the player.

The playlist can be saved into a file, in order to be reused later.

Click the **SAVE PLAYLIST** on the Playlist toolbar to start the process.

The playlist can be loaded from an external file. To start the process, click on **THE LOAD PLAYLIST** button on the playlist toolbar. The application notifies the user that the current playlist will be overwritten and, if the user agrees, the chosen playlist file is loaded.

Note that when saving the playlist, only the physical path of the files will be saved, not the associated metadata. Those will be recalculated while reloading the file, when the playlist will be reopened.

The playlist file will be saved with an extension *.oto*. This is a text file containing a list of paths to audio files, one per line. If, during the load process some of the files result not to be available or of the wrong format, the file is skipped and additional information is written in the console.

The playlist files will be saved into the User's ApplicationData directory configured for the system.

Playlist toolbar

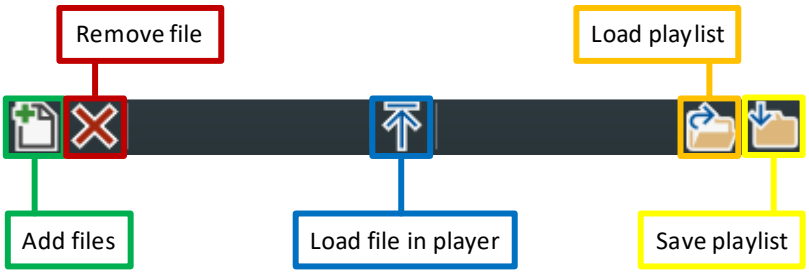

Figure 10: Playlist toolbar

The playlist toolbar shows three sets of buttons.

The group on the left is for *adding and removing files* to/from the playlist.

The group on the centre actually contains only one button, to invoke the process that *loads the currently selected file into the player*.

The group on the right is for *loading and saving the current playlist* from/to the drive.

Search tool

Figure 11: Search tool

This is a text edit widget where the user can type some text. While typing, all the files currently in the playlist will be scanned and, if found containing the search text into their name, will be highlighted in red.

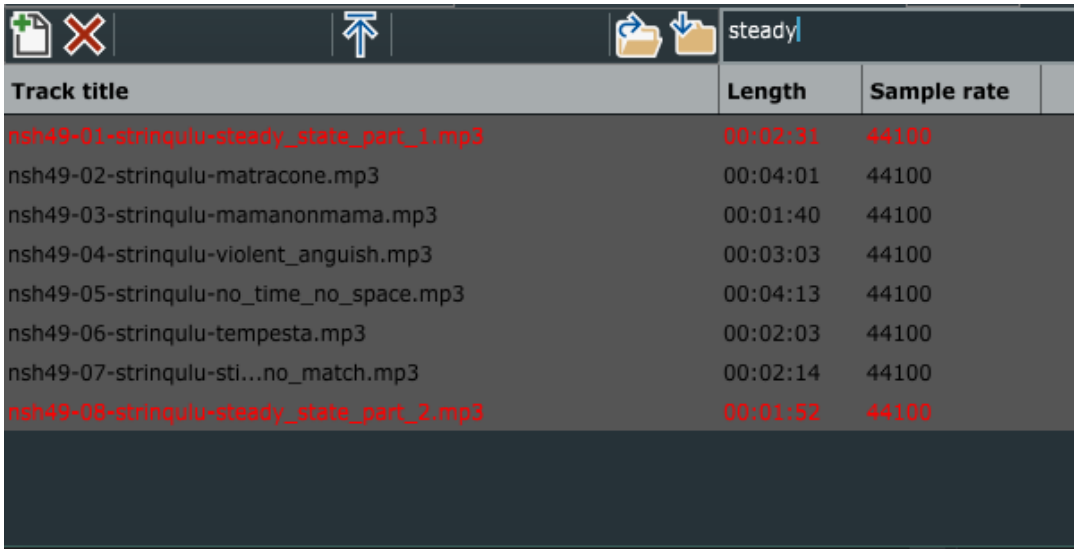

Figure 12: Searching files in the playlist

This is made possible intervening during the `CellPaint` process of the grid and checking whether the string, going to be displayed in the current cell, does contain the search text. If it does, the colour of the `Graphics` object is changed. This will be explained later in more details.

The code executing this logic can be found in `PlaylistGrid.cpp at line 158`, method `paintCell`:

```
if(searchText.length()> 0 && track.fileName.contains(searchText))
{
    graphics.setColour(Colours::red);
}
```

Structure of the application

The application is made of the following parts, each of which is delegated to a single action:

- **PLAYLIST**: manages the user's playlist
- **PLAYER**: manages the playback of the current track
- **MIXER**: provides functionalities like volume control and audio level display
- **OTHER CLASSES**: like custom **LISTENERS**, to implement the **OBSERVER** pattern and improve the architecture, decoupling the different sub-systems.

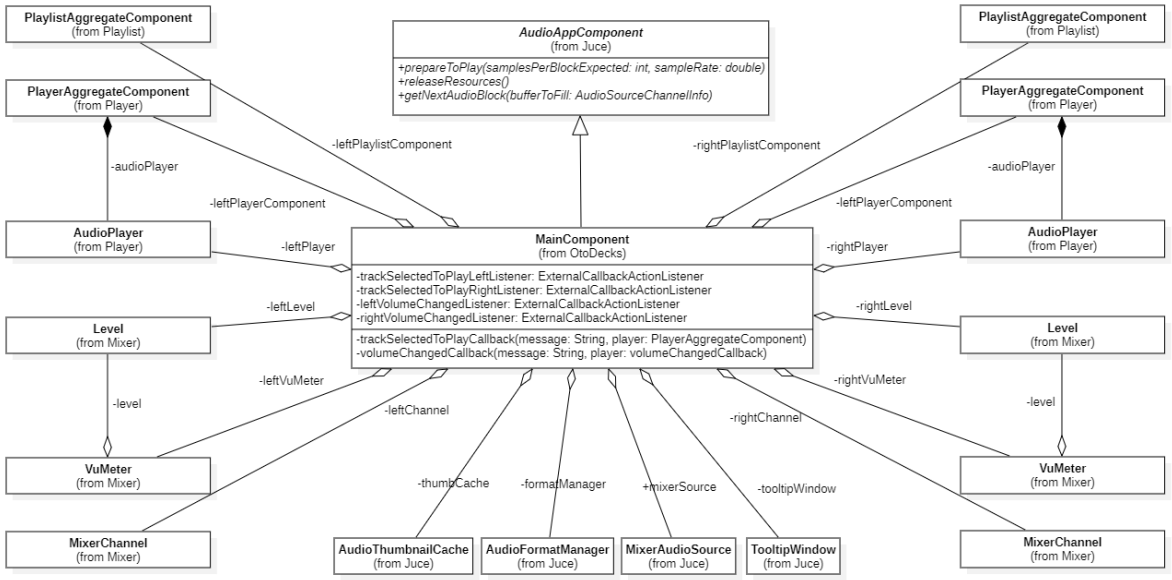

Figure 13: Class Diagram of the MainComponent and related classes

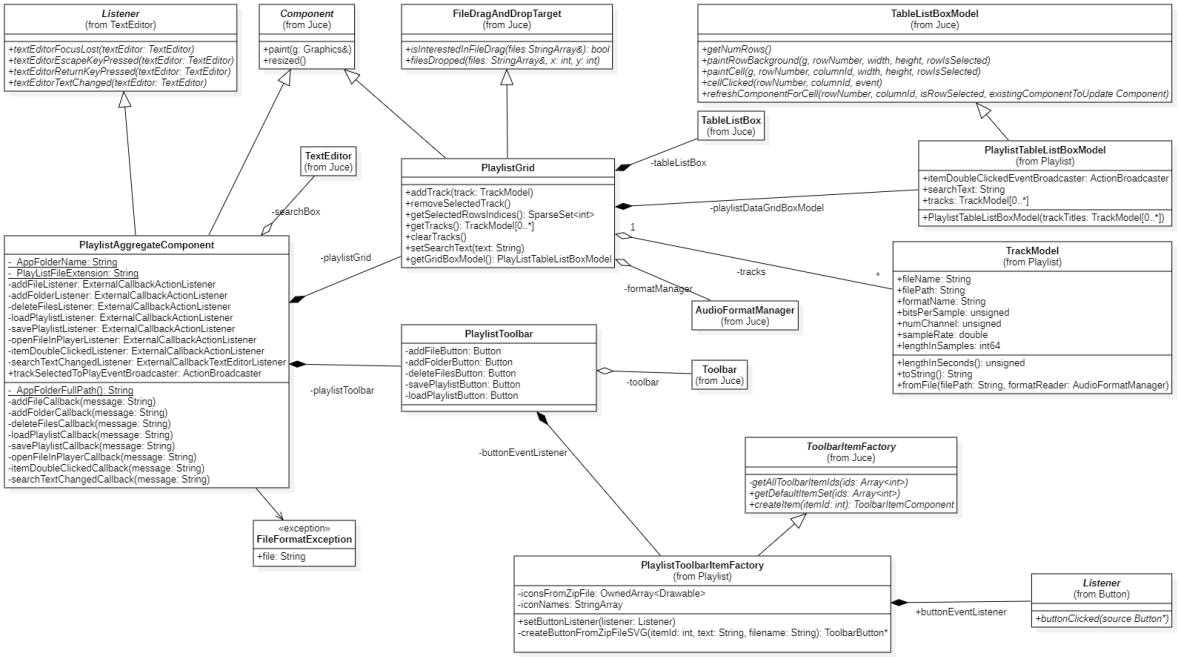

Figure 14: Class Diagram of the Playlist sub-system

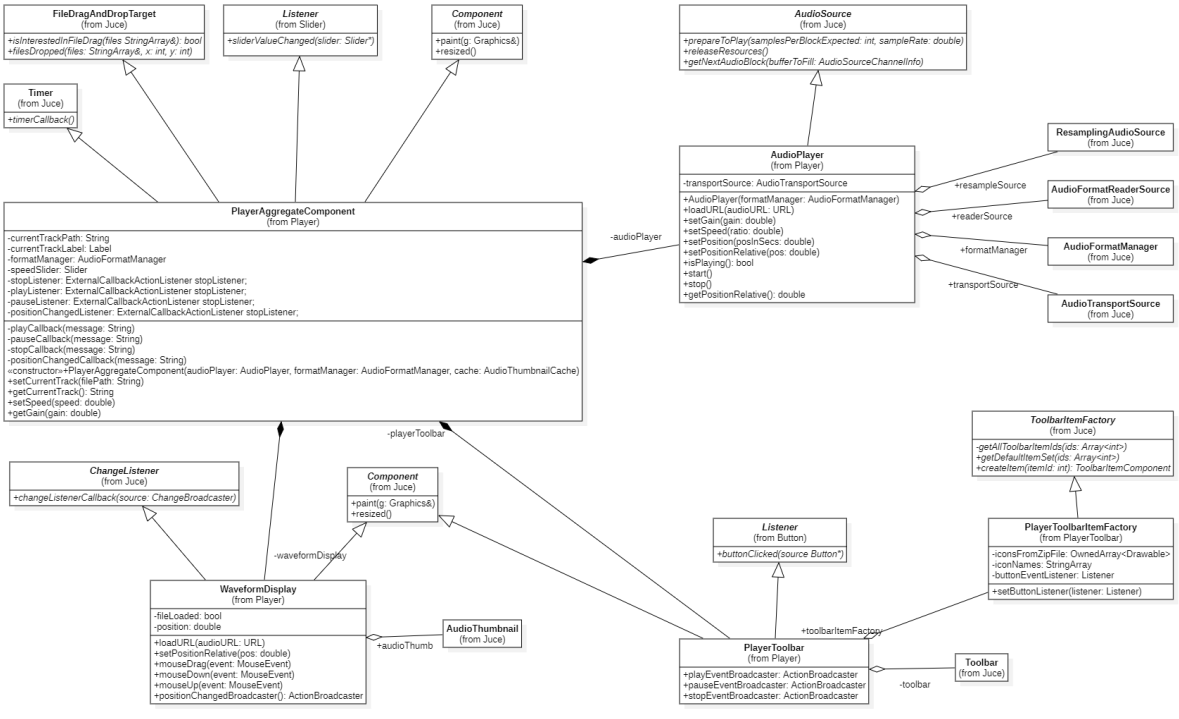

Figure 15: Class Diagram of the Player sub-system

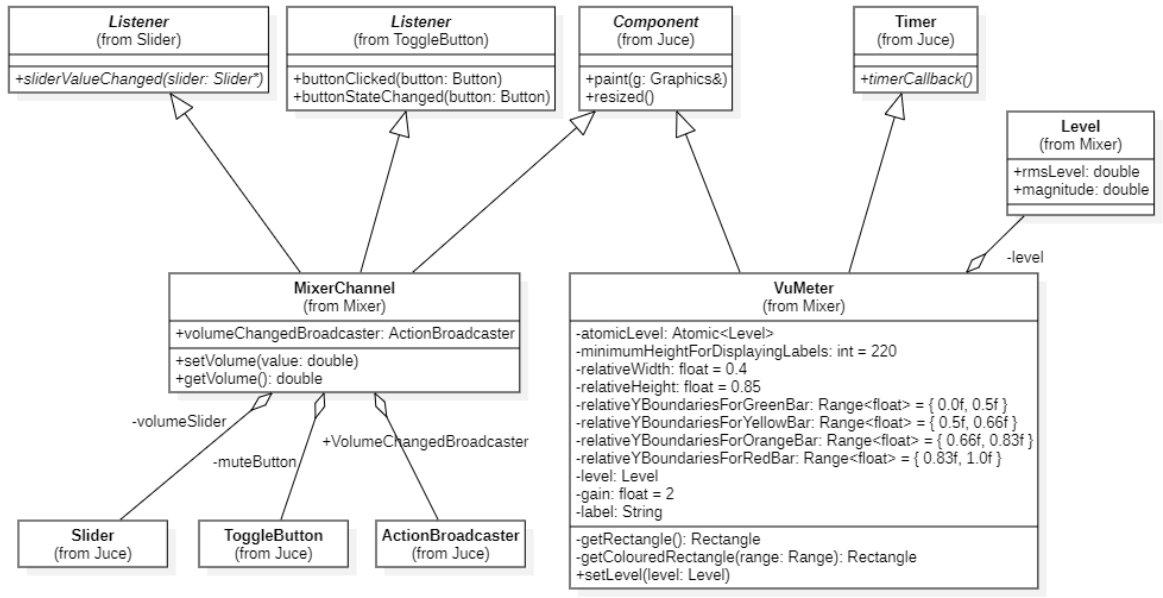

Figure 16: Class Diagram of the Mixer sub-system

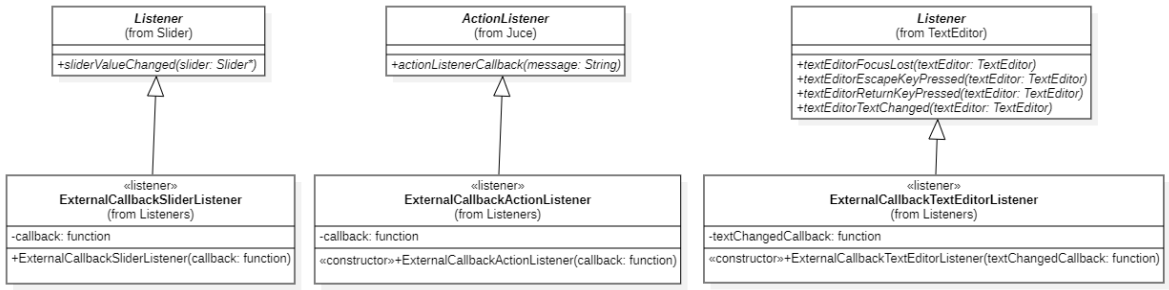

Figure 17: Class Diagram of the Listeners used to exchange messages among the components

R1: 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.

R1A: Component has custom graphics implemented in a paint function

An additional new component that makes use of the `paint` function to display its content is the **VUMETER**.

This control receives the audio levels of the current playing sample in real-time, and based on that, it builds a bar with a height proportional to the audio level of the current sample, with four portions of different height, coloured with different colours, based on the level value.

The entire construction of this bar is made in the `paint` function of the `VuMeter` class.

One first major problem I encountered writing this component is due to the fact that the user interface lives in a different thread than the audio thread. It is therefore impossible to just read the level value of the current audio sample, in `MainComponent::getNextAudioBlock`, which lives in the audio thread, and invoke the `VuMeter::paint` right after, which lives in the UI thread.

I made a first attempt to solve the issue using the `MessageManagerLock` class provided by **Juce**. This did actually allow to invoke the `paint` method from within the `getNextAudioBlock`, and have the component updated in real time, but this also introduced glitches in the audio, due to the audio thread being blocked for the time needed to update the user interface.

I tried then a different approach, based on the `Atomic` class and a `Timer`. The `Atomic` class managed the thread-safe access to the variable I needed to share from the two concurrent threads, while the timer, which ran in the UI thread, triggered the call of the paint method.
This solved the problem in a satisfactory way, having the UI updated in real time, without introducing issues in the audio.

This is the declaration of the `Atomic` variables, containing the audio levels per channel, in `MainComponent.h`:

```
//Thread-safe wrapper for Level, used to send the current audio levels
to the left VuMeter
    Atomic<Level> leftLevel;
//Thread-safe wrapper for Level, used to send the current audio levels
to the right VuMeter
    Atomic<Level> rightLevel;
```

In `getNextAudioBlock`, in `MainComponent.cpp`, the two variables are set:

```
leftLevel.set({ leftRmsLevel, leftMagnitude });
rightLevel.set({ rightRmsLevel, rightMagnitude });
```

In `VuMeter.cpp`, the `timerCallback` function, managing the timer's event, calls the function that sets the levels:

```
void VuMeter::timerCallback()
{
    setLevel(atomicLevel.get());
}
void VuMeter::setLevel(Level level)
{
    this->level = level;
    this->repaint();
}
```

R1B: Component enables the user to control the playback of a deck somehow

WAVEFORMDISPLAY component is now capable of controlling the playback position on the current track.

The user can drag the vertical bar showing the current position around and control the playback position. This gives a more immediate feedback on where the new playback location is going to be.

The `WaveformDisplay` class now controls the mouse events performed onto the component and uses that information to move the position bar in the control and set the new position in the currently playing track accordingly.

The interaction of the user with the component is managed via the mouse event handlers provided by the `Component` class, in particular `mouseDrag`, `mouseDown`, `mouseUp`.

`mouseDown` and `mouseDrag` are very similar: they grab the mouse position and calculate the `relativePosition` respect to the total width of the component, then they send a message containing the new coordinates, through the `positionChangedBroadcaster`.

```
//Calculate the new relative position
double relativePosition = event.getPosition().getX() /
static_cast<double>(getWidth());
//Control overflows and underflows
if (relativePosition < 0)
    relativePosition = 0;
if (relativePosition > 1)
    relativePosition = 1;
positionChangedBroadcaster.sendActionMessage(static_cast<String>(relativ
ePosition));
```

The only difference between the two methods is that `mouseDown` changes the cursor type to `DraggingHandCursor`.

```
//Set the mouse cursor to draggingHand
setMouseCursor(MouseCursor::DraggingHandCursor);
```

`mouseUp`'s only duty is to reset the mouse cursor to default when the repositioning is terminated.

```
//Reset the cursor to Normal
setMouseCursor(MouseCursor::NormalCursor);
```

The message sent by the **BROADCASTER** it then caught by the `PlayerAggregateComponent`'s listener `positionChangedListener`, which invokes the `positionChangedCallback` function, which in turn resets the position on the player:

```
void PlayerAggregateComponent::positionChangedCallback(const String&
message)
{
    this->audioPlayer.setPositionRelative(message.getDoubleValue());
```

```
}
```

R2: Implementation of a music library component which allows the user to manage their music library

The Playlist sub-system of my application is an evolution of the **PLAYLIST** Component developed during the course.

Main `PlaylistAggregateComponent` is a container component which coordinates the two parts forming the user widget, `PlaylistGrid` and `PlaylistToolbar`.

It acts as a **MEDIATOR** *(GoF – Design Patterns)*, coordinating the exchange of information among the *collegues*, which in this case are `PlaylistGrid` and `PlaylistToolbar`.

The **PLAYLISTTOOLBAR** is a component that shows a toolbar with buttons that, when clicked, generate events caught by the `PlaylistAggregateComponent`. This then re-routes the events to the **PLAYLISTGRID** component where these events are dealt with.

The `PlaylistToolbar` receives the click events on the toolbar buttons and sends the proper messages, `addFile`, `loadPlaylist`, etc, via **BROADCASTERS**.

For the declarations of the **BROADCASTER**, please see `PlaylistToolbar.h, lines 42 to 65.`

For the use of **BROADCASTERS** to send messages, see `PlaylistToolbar.cpp`, method `PlaylistToolbar::buttonClicked`.

For how the messages are caught by the `PlaylistAggregateComponent`, please see `PlaylistAggregateComponent`'s constructor, in `PlaylistAggregateComponent.cpp`.

R2A: Component allows the user to add files to their library

The **ADD FILE** button on the toolbar opens a dialog box where the uses can select one or more audio files to add to the library. In addition, files can also be added to the library dragging and dropping them from outside the application.

Please look at `PlaylistAggregateComponent.cpp`, method `addFileCallback` to see the code details.

The **DELETE FILE** button lets the user remove the currently selected file from the library.

R2B: Component parses and displays meta data such as filename and song length

When the user chooses the files to select, or drags files from outside onto the library, these files are parsed, relevant information is extracted, such as length and quality of the file, and part of this information is displayed in the playlist grid.

The code where the audio metadata is extracted from the file can be found in `PlaylistGrid.cpp`, method `PlaylistGrid::TrackModel::fromFile`.

```
PlaylistGrid::TrackModel PlaylistGrid::TrackModel::fromFile(const
String& filePath, AudioFormatManager& formatManager)
```

```
{
    File selectedFile(filePath);
    std::unique_ptr<AudioFormatReader>
formatReader(formatManager.createReaderFor(selectedFile));

    if (formatReader == nullptr)
        throw FileFormatException{ selectedFile.getFullPathName() };

    PlaylistGrid::TrackModel track(
        selectedFile.getFileName(),
        selectedFile.getFullPathName(),
        formatReader->getFormatName(),
        formatReader->bitsPerSample,
        formatReader->numChannels,
        formatReader->sampleRate,
        formatReader->lengthInSamples
    );
    return track;
}
```

The `TrackModel` class offers also a method for calculating the length in seconds of the track:

```
unsigned PlaylistGrid::TrackModel::lengthInSeconds() const
{
    if (sampleRate != 0)
        return lengthInSamples / sampleRate;
    else
        return 0;
}
```

You can find this method in `PlaylistGrid.cpp`.

Please note that the `TrackModel` class, along with `PlaylistTableListBoxModel`, are created as *nested classes* of `PlaylistGrid`, because these classes are strictly related to the parent class `PlaylistGrid`, and have no sense without it. This is exactly how **Juce** library does in many cases. See for example `Button` and `Button::Listener`.

R2C: Component allows the user to search for files

The toolbar also has a search text field which the user can use to search within the library.

Typing characters into this search field, a search is performed in the currently loaded playlist and all those files containing the search string into their name will be highlighted in red.
For a more general search for files into the file system, the user can rely on the powerful search tools of the OS to find the files she is after and drag them into the library using the drag and drop functionality.

When the user types into the **SEARCH** `TextEditor`, the event is caught by the `searchTextChangedListener` which calls the method `setSearchText` on the `PlaylistGrid`, passing the message containing the current search text. This method of `PlaylistGrid` sets a variable `searchText` on the `playlistDataGridBoxModel` and finally invokes the repaint method which, in turn, causes the grid to be redrawn.

When the grid is repainted, the method `PlaylistGrid::PlaylistTableListBoxModel::paintCell` is called for every cell in the grid. Here, if the filename of the track contains the search text, the corresponding item is drawn in red.

```
if(searchText.length()> 0 && track.fileName.contains(searchText))
{
    graphics.setColour(Colours::red);
}
```

Please look at `PlaylistAggregateComponent`'s constructor, in `PlaylistAggregateComponent.cpp`, to see how the `searchTextChangedListener` is configured.

Then, at `searchTextChangedCallback` method to see where the message is passed through to `playlistGrid`.

Finally, look at `paintCell` method in `PlaylistTableListBoxModel.cpp` to see how the check on the filename is done.

R2D: Component allows the user to load files from the library into a deck

The toolbar provides a **OPEN FILE IN PLAYER** button, which loads the selected file into the correspondent player. As the player can only play one file at a time, only one file at a time can be selected from the **PLAYLIST**.

Clicking on the **OPEN FILE IN PLAYER** button on the **PLAYLIST TOOLBAR**, a message is broadcasted and received by the `openFileInPlayerListener`, which retrieves the file full path from the selected row in the grid and broadcasts another message containing the file `fullpath`, using the `trackSelectedToPlayEventBroadcaster`.

Look in `PlaylistAggregateComponent`'s constructor in `PlaylistAggregateComponent.cpp` to see how the `openFileInPlayerEventBroadcaster` and the `openFileInPlayerListener` are configured.

See also `openFileInPlayerCallback` in `PlaylistAggregateComponent.cpp` to see how the `trackSelectedToPlayEventBroadcaster` is used to send the message.

This message is then received by the `trackSelectedToPlayLeftListener` in `MainComponent` (see constructor for the configuration of the **LISTENER**), which hands the message over to the player's method `setCurrentTrack`.

See method `trackSelectedToPlayCallback` in `MainComponent`.

This same process can also be trigger double-clicking on the selected track in the grid.

As additional feature, an external file can be added to the player without passing through the playlist, simply selecting one single file from, i.e. a Windows Explorer, and dragging it directly onto the player.

This implementation can be found in the methods `isInterestedInFileDrag` and `filesDropped` of `PlayerAggregateComponent.h`

R2E: The music library persists so that it is restored when the user exits then restarts the application

The toolbar also provides two buttons to save the current library into a file and load a previously saved playlist file into the playlist.

When the user clicks on the **SAVE PLAYLIST** button, an event is raised by the **PLAYLIST TOOLBAR** and caught by the `PlaylistAggregateComponent`. The event is then processed by the method `savePlaylistCallback` in `PlaylistAggregateComponent.cpp`

Playlists have file extension *.oto* and are saved into a custom folder called *OtoDecks* within the *ApplicationData* folder of the user.

Loading a playlist from the mass storage follows the same workflow, but uses the method `loadPlaylistCallback` in `PlaylistAggregateComponent.cpp`.