Masters Thesis

Multi-Channel Lung Sound Recording Software

conducted at the
Institute of Signal Processing and Speech Communication
Graz University of Technology, Austria

by
Johannes Wolfgruber, BSc

Supervisor:
Univ.-Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf

Graz, March 25, 2020

Abstract

Computational lung sound analysis provides a more objective and standardized way of diagnosing lung diseases. For the development of a multi-channel lung sound recording device the software used for the audio recording plays an integral role. In this thesis, a modern audio software application specifically designed to be used in conjunction with such a recording device was developed. The requirements for the software have been analyzed to choose the most applicable technology for the development. The state-of-the-art audio development frameworks JUCE and Tracktion Engine are introduced and the software's architecture was designed for testability and future extensibility. The final implementation of the software is described in detail, including a manual for the end user. Furthermore, the hardware device consisting of an audio interface, the microphones inside stethoscope heads as lung sound transducers and the auscultation pad holding the sensors, is introduced. Additionally, different deep neural network models for multi-channel lung sound classification were re-implemented and evaluated in the machine learing framework Keras. The aim was to classify subject's breathing cycles as either healthy or pathological (patients suffering from idiopathic pulmonary fibrosis). A multilayer perceptron (MLP) model was used as the baseline model. A bidirectional gated recurrent neural network (BiGRNN) was implemented for exploiting temporal information and additionally provided with a convolutional front-end (ConvBiGRNN) to make use of spatial information. Furthermore, for comparison with these three models a dense convolutional neural network was implemented. The best performance was achieved with the BiGRNN model with a F-Score of $F_1 = 88.0\%$.

Contents

1 Introduction

This thesis has its origin in Elmar Messner's dissertation "A Holistic Approach to Multi-Channel Lung Sound Classification" [1] in which he developed a multi-channel recording device. This was used to build up a lung sound database to train neural networks for classifying subjects as either healthy or suffering from idiopathic pulmonary fibrosis. The need for such a device is also apparent in the lack of commercially available multi-channel lung auscultation devices. Traditionally lung auscultation is a very subjective assessment procedure, which relies heavily on the examiner's experience and auditory capabilities. The examination is not subject to a standardized method and thus can not be easily reproduced and compared against previous examinations. The above device tries to allow for a more objective and standardized assessment technique with multiple stethoscope heads fixed to predetermined positions and the recording of the lung sounds for later examination. With a dedicated recording software the physician is able to not only hear individual sensor positions, as with a traditional stethoscope, but also see the waveforms of all 16 sensors at once. The fact that the recordings of each examination of a patient are available, allows for an improved analysis of the development of the disease over time. In order to optimize the usability and portability and to allow the widespread deployment of such a device for the diagnosis of pulmonary diseases the auscultation pad was redesigned to house the recording hardware. As part of this thesis the accompanying recording software was developed to provide the physician with an easy to use tool for the multi-channel recording of lung sounds and the visual examination of the waveforms and spectrograms.

1.1 Computational Lung Sound Analysis

A systematic review of articles about the automatic adventitious respiratory sound analysis was performed in [2]. The authors describe the automatic detection and classification of adventitious sounds as a useful tool to help physicians in diagnosing and monitoring lung diseases. Although this has recently been the objective of a growing number of studies, there is no standardized approach and comparison for the detection and classification of adventitious sounds. The data to perform the detection and classification tasks is collected using different instrumentation, including microphones, stethoscopes and accelerometers, or obtained from online repositories and CDs. The algorithms used, span a wide range of methods from empirical rule based methods to complex machine learning techniques. The overall goal is to detect and classify abnormal respiratory sounds, as these are characteristic of serious lung diseases. Normal and abnormal respiratory sounds can be heard by performing auscultation. Auscultation is the medical term used to describe the process of listening to sounds from inside the human body using a stethoscope or other tools. Normal respiratory sounds can be categorized according to where they are heard or originate into vesicular sounds (most of the lung fields), bronchial sounds (near the second and third intercostal space), broncho-vesicular sounds (posterior chest between the scapulae/center of the anterior chest), tracheal sounds and mouth sounds [2]. Vesicular sounds are low pitched, low-pass filtered (due to the chest wall), noise-like sounds in the range of $100 - 1000Hz$ and can be heard during inspiration and early expiration. Bronchial sounds are high pitched, loud and hollow sounds in the range of $100 - 5000Hz$ and occur during both inspiration and expiration.

Broncho-vesicular sounds are pitch- and frequency-wise in between the vesicular and bronchial sounds. Tracheal sounds cover a similar range of frequencies as the bronchial sounds but with a higher pitch and a very loud and harsh sound quality. Mouth sounds cover a frequency range of $200 - 2000Hz$ with a white-noise like quality. Because of the different sound characteristics of normal respiratory sounds at different auscultation locations, the analysis can become more complex for multi-channel signals. Abnormal respiratory sounds include adventitious sounds, which are superimposed on normal respiratory sounds and can be categorized into continuous adventitious sounds (CAS) and discontinuous adventitious sounds (DAS). Abnormal respiratory sounds also include normal breath sounds occurring in abnormal areas and the lack or reduced intensity of sounds during breathing. CAS are sounds with a duration of more than $250ms$ and contain wheeze sounds (sibilant/musical, duration $> 80ms$, high pitch $> 400Hz$), rhonchi sounds (sibilant/musical, duration $> 80ms$, low pitch $< 200Hz$), stridor (sibilant/musical, duration $> 250ms$, high pitch $> 500Hz$), squawks (short musical/non-musical, duration $\pm 80ms$, low pitch $200 - 300Hz$) and gasps (whoop, duration $> 250ms$, high pitch). DAS are sounds with a very short duration of less than $25ms$ and include fine crackles (non-musical/explosive, duration $\pm 5ms$, high pitch $650Hz$), coarse crackles (non-musical/explosive, duration $\pm 15ms$, low pitch $350Hz$) and pleural rub sounds (non-musical/rhythmic, duration $> 15ms$, low pitch $< 350Hz$).

As stated in [2], there exist a number of devices for the automated lung sound analysis using different approaches. Most of these have in common, that they are large and complex and thus there is still a need for portable non-intrusive devices for monitoring lung sounds. The algorithms and methods used by the studies summarized in [2] include machine learning classifiers such as Multilayer Perceptron Models (MLP), Support Vector Machines (SVM), Gaussian Mixture Models (GMM), k-Nearest Neighbor (k-NN), Hidden Markov Models (HMM) and Logistic Regression. An important consideration in developing new and improved algorithms for detection and classification tasks is the choice of feature extraction method that is used, because this greatly influences the performance. Additionally, it is essential to obtain the data from a number of different patients, in order to get a better generalization and avoid over-fitting, under-fitting and patient specific results.

1.2 Scope of the Thesis

The goal of this thesis was to develop a modern audio software application for multi-channel lung sound recordings. The first step was to gather the softwares requirements by analyzing the pre-existing Matlab application from [1] and to see which of the features to keep or to extend. With this set of requirements the choice was made to develop a cross-platform C++ application using the frameworks JUCE and Tracktion Engine. The next step was to understand the two frameworks and how to deploy them for development. After that an overall module based architecture for the software was created including a build infrastructure which can be automated for continuous integration and supports all platforms. The main part of the thesis consisted of the implementation of the different requirements, including a real-time spectrum analyzer and level meter, calculating and plotting spectrograms of the recordings, implementing filters for the audio data as well as for the decimation process and the design of a modern user interface which is easy to use and logically arranged. Independently of the recording software, deep neural networks from [1] have been implemented in Keras and achieve similar performance. This was done with the lung sound database from [1].

1.3 Chapter Outline

The thesis is structured as follows:

- *Chapter 2:* This chapter is a collection of the requirements that had to be met by the software. The requirements itself are subdivided into functional and non-functional requirements representing the behavioral aspects and technical aspects of the application, respectively.
- *Chapter 3:* With regards to the gathered requirements the technology to use for the development of the software is chosen. The chapter also introduces the most important functionalities of the JUCE and Tracktion Engine frameworks that were used for development and describes the build tools and infrastructure of the project.
- *Chapter 4:* This chapter explains the underlying architectural concept used by the JUCE framework and the application. A high-level overview of the software architecture and the JUCE module format is presented.
- *Chapter 5:* At the beginning of this chapter important concepts and design patterns used for the software development are explained. Then the details of the implementation are described for the different modules of the application.
- *Chapter 6:* An overview of the software's functions and how to utilize them from a user's point of view is given. This includes some screenshots of the application.
- *Chapter 7:* This chapter gives a short overview of the hardware the software was developed for, including a description of the audio interface, the microphones, the stethoscope heads and the auscultation pad.
- *Chapter 8:* For this chapter, the deep neural network models from [1] were re-implemented in the high-level machine learning API Keras to see if the same results could be achieved. The implementation of the models and the classification results are shown.

2 Software Requirements

One of the first steps in developing a software application is to collect all necessary requirements. This includes all aspects from the end-users perspective, as well as technical requirements. This chapter is an outline for all requirements that have to be fulfilled by the multi-channel lung sound recording software. The requirements are partly derived from the Matlab GUI-application from [1], to maintain existing features. Additionally some new requirements were added, to extend the applications feature set.

2.1 Functional Requirements

R.1 *Multi-Channel recordings:* The main feature of the application is to be able to choose a number of up to 16 input channels for simultaneous recording. Per default all 16 channels should be recorded, but the user should be able to choose a subset of the available channels.

R.2 *Input gain control:* There has to be a control to set the input gain for each channel individually, as well as for all channels at once. This control should be hidden per default and only be accessible through an elevated mode of operation (expert/admin mode).

R.3 *Input high-pass filter:* There has to be a high-pass filter on each channel with adjustable cutoff frequency. The original recording software did not contain an adjustable high-pass filter but instead used a Bessel high-pass filter with a cutoff frequency of $f_c = 80Hz$ and a slope of $24dB/oct$ inside the recording hardware [1]. Because of the new hardware a filter before the analog-to-digital conversion was not possible and instead a digital filter with adjustable cutoff frequency and a slope of $24dB/oct$ is used. The cutoff frequency should also be adjustable for all channels individually with an option to apply the same cutoff frequency to all channels. This control is hidden by default and only accessible through the expert mode.

R.4 *Sensor visualization:* There has to be a visualization of the 16 sensors of the lung sound recording device, which resembles the arrangement of the sensors on the auscultation pad. The individual sensors can be selected and enabled/disabled. Each sensor gets a corresponding track, which contains the recorded audio clips. The 16 tracks are arranged underneath each other, such that a distinct point in time of each sensor recording is aligned vertically, with a timeline display at the top. Selecting a track or clip in this view also selects the corresponding sensor in the sensor view.

R.5 *Input level meter and frequency analyzer:* The input level has to be displayed in real-time as a bar meter for the selected sensor. This can be used to fine tune the input gain of the channel and to make sure there is no clipping occurring during recording. Different colors indicate different input levels (green for levels up to $-12dB$, yellow for levels up to $0dB$ and red for levels above $0dB$). The frequency spectrum of the input signal is displayed in real-time for the selected sensor. Additionally the frequency and phase response of the high-pass filter has to be plotted alongside the spectrum.

R.6 *Audio waveform visualization:* The recorded audio clips have to display the waveform of the recording. While recording the waveform of the input signal should be displayed in real-time. The selected area of a recording has to be displayed in a separate, more detailed view including a timeline and a playhead. The detail view allows to zoom in and out of the audio waveform.

R.7 *Spectrogram of a selected region:* Underneath the detail view the spectrogram of the selected region has to be displayed. It is possible to configure the most important settings of the spectrogram (window, overlapping, FFT size). The default values for the spectrogram are taken from the audio processing framework from [1] with a Hamming window of 512 samples length and an overlap of consecutive frames of 37.5%.

R.8 *Metadata of subject:* It is possible to store some subject related metadata. Upon creating a new project, the user has to enter an unique subject code, the subjects first and last name, the subject's gender and date of birth. These metadata elements cannot be changed later on but can be viewed in the settings pane alongside the automatically generated examination number and examination time.

R.9 *Selection behavior:* It is possible to select whole audio clips of a recording, as well as parts of it by clicking and dragging over the desired range. The selected part is highlighted on the clip and the corresponding sensor is selected in the sensor view and displayed in the detail view. The playhead jumps to the beginning of the selection and the corresponding track is solo isolated. The user should be able to activate a looping mode and the loop range should automatically span the selection range. Input monitoring is enabled automatically for the selected sensor. Only one sensor can be monitored at the same time. Additionally, the selected part can be played back and listened to.

R.10 *Saving recordings:* A recording session can be saved, including the recorded audio data and subject metadata. This session can be loaded and listened to later on.

R.11 *Modern look and feel:* The application has to follow a modern user interface and user experience (UI/UX) design. The colours and shapes are derived from some well established design guidelines. The handling of the software has to be intuitive from a user's perspective. All major controls and information has to be visible and accessible at the same time. Proportions between different elements of the user interface can be changed by the user.

2.2 Non-Functional Requirements

R.12 *"Infinite" length recording:* The length of the recordings are not limited by the application but only by the available storage on the computer running the software.

R.13 *Cross-platform compatibility:* The software should be compatible with all major operating systems (Windows, Mac OS X, Linux).

R.14 *Exporting the audio data:* When exporting the audio files the sample rate of the audio data is reduced to $f_s = 16kHz$, as is the case in [1]. Additionally the high-pass filters are applied on the audio data.

R.15 *File structure:* The structure of one recording project follows the same structure as in [1]. The main folder is named: `subjectCode_firstname_lastname`. Inside the main folder a subfolder for each examination is created containing a project file named: `examinationNumber_DD_MM_YYYY.mclsproject`. Alongside the project file there is an audio folder with subfolders for the raw audio data and the exported/downsampled audio data.

3 Technology Stack

Considering all requirements of the software, a choice has to be made on which technologies to use. General considerations when selecting a certain technology are, how well it is adopted and maintained and which license it is covered with. Additionally for an audio application with 16-channel recording capability performance is of the essence, which already rules out a lot of technologies. With these criteria and the cross-plattform capability in mind, there are two possibilities for the type of software, a cross-platform native desktop application or a cross-platform web application with a local server as the backend and a web page as the frontend. The problem with the latter is the lack of support for multi channel audio recordings. When dealing with audio in web applications we are limited to the Web Audio API [3], which is not flexible enough for working with external multi channel audio interfaces. Despite the fact that the development of a web application may be more flexible with regards to the software architecture and lose coupling of user interface and logic, the performance of a native desktop application can hardly be met. For these reasons, the choice was to develop the application as a cross-platform native desktop application in C++ with the JUCE framework [4] and the Tracktion Engine audio engine [5], which is based on JUCE. This chapter is an introduction to JUCE and the Tracktion Engine and outlines the build tools and infrastructure.

3.1 JUCE

Jules' Utility Class Extensions (JUCE) is a partially open-source cross-platform C++ application framework for desktop and mobile application development. JUCE was originally part of the code base of the Tracktion Digital Audio Workstation (DAW), before the creator Jules Storer decided to extract it as a framework. The fact that it was created for an audio application lends JUCE its main feature, the large set of audio functionality. JUCE supports audio devices on all major platforms (e.g. CoreAudio, ASIO, ALSA, JACK, WASAPI, DirectSound) and contains readers for audio file formates (e.g. WAV, AIFF, FLAC, MP3) [6]. Besides that, JUCE contains classes for user interface elements, graphics, XML and JSON parsing, multi-threading and other useful features [6]. All these features make JUCE a standard in audio application development [6]. This section is used to explain the most important classes and concepts of the JUCE framework used in the development of the Multi-Channel Lung Sound Recording (MCLSR) software. If not specified otherwise, all text in monospaced font in this chapter refers to class and method names that are part of the JUCE framework.

3.1.1 JUCE Application, Window and Components

The `JUCEApplication` class represents the entry point to a JUCE application and is used to specify the initialization and shutdown code of the application [7]. This class is derived from the `JUCEApplicationBase` class and provides default implementations of some pure virtual base class methods. An application that wants to run an event loop needs to subclass the `JUCEApplication` or `JUCEApplicationBase` classes and implement the pure virtual methods

[7]. The START_JUCE_APPLICATION macro is used to define the platform-specific boilerplate code to launch the application as well as the main function, which is the entry point of the application [7]. Typically the JUCEApplication derived class holds a pointer to the main window, which in turn holds a pointer to the main content component.

A subclass of the DocumentWindow class is used as the main window of the application. The DocumentWindow is derived from ResizableWindow, so it can be resized, minimized and maximized. The window's titlebar can show the application's name or any other text and an icon can optionally be specified with the setIcon() method [8]. As it is not advisable to add child components directly to the DocumentWindow, it typically holds a pointer to the main content component, where all child components can be added [8].

The Component class represents the base class for all user interface objects. With the addAndMakeVisible() method child components can be added to parent components, which is used to build a hierarchical graph of the user interface. The most important methods of a component are the paint() and resized() methods. The paint() method can be overridden to draw the content of the component inside the current Graphics context handed to it. It gets called when parts of the component needs redrawing, either because it is marked "dirty" by a call to the repaint() method, or because something occurred on the screen that caused a section of the window needing to be redrawn [9]. The resized() method gets called when the components size changes and is used to layout its child components. This way, when a parents size changes, it resizes itself and all its children accordingly. Because this gets called synchronously it is not advisable to perform long lasting calculations inside this method, to avoid blocking the entire message thread (the thread running the JUCE applications event-dispatch loop, essentially the main thread of a JUCE application) and thus rendering the application unresponsive [9].

3.1.2 Multithreading

JUCE provides a couple of classes for multithreading and thread synchronization like the Thread class, which encapsulates a thread. To create a thread, a subclass of the Thread class, which implements the run() method, has to be created. To start the thread (execute the run() method on its own thread) the startThread() method is called, specifying the threads priority (0 = lowest, 10 = highest) [10]. The Thread additionally provides static thread-related methods, such as sleep(), yield() and wait() [10].

According to [11], the ThreadPool class is a managed set of threads that will run a list of ThreadPoolJob objects. This class provides methods to interact with these jobs (add, remove, search for job, get number of jobs, etc.).

A ThreadPoolJob represents a task, that needs to be run on its own thread. In contrast to the direct implementation of the Thread class, the ThreadPoolJob is not responsible for the creation and lifetime of its thread (this is managed by the ThreadPool). A task can be created by subclassing ThreadPoolJob and implementing the runJob() method. This method should periodically check if the thread should exit (shouldExit() method) and return if the method returns true [12]. The runJob() method returns a JobStatus, which can be either jobHasFinished or jobNeedsRunningAgain, depending on wether the job needs to execute the runJob() method again or not.

The AsyncUpdater is used to handle a callback asynchronously. The handleAsyncUpdate() method needs to be implemented, containing the callback. A call to triggerAsyncUpdate() (save to call from any thread, but can potentially block real-time threads) posts a message to the message thread, which in turn calls the handleAsyncUpdate() method as soon as possible [13]. This class is for example useful when collecting multiple updates into a single callback.

3.1.3 Data Model

The JUCE documentation [14] describes the `ValueTree` as a powerful tree-like structure to hold data and execute undo and redo actions for changes to this data. A `ValueTree` contains a list of named properties as `var` (see next paragraph) objects and any number of sub-trees. They are a lightweight reference to a shared object, which acts as the data container. This makes `ValueTree` creatable on the stack and copyable without performance impacts, as this simply creates a new reference to the same shared object. The `ValueTree` is analogous to the XML format [15] (with the exception of text elements), where the type name maps to an XML tag and the properties map to an XML attribute. This makes XML a good serialization format for `ValueTree`. Listing 3.1 shows an abbreviated example of a serialized `ValueTree` like it is used for storing and loading of a `te::Edit` (see Section 3.2.2). If needed, `ValueTree` can also be serialized to a compact binary format very quickly. Additionally the `ValueTree` provides the option to pass an `UndoManager` to the methods that change the tree's data. This `UndoManager` is used to track any changes to the object and is an easy way of undoing and redoing changes. Listeners can be registered to a `ValueTree` to be notified when a property changes and when sub-trees are added or removed. These listeners are stored with the `ValueTree` and not with the shared object, which eliminates the need to unregister as a listener if the callbacks were registered to a copy of the `ValueTree` which goes out of scope once the notifications are no longer needed. All the above properties of the `ValueTree` make it a good fit as the underlying data structure for the data model of an application. Combined with the `ConstrainedValue` class (see Section 5.2) it is easy to follow an MVC-type architectural paradigm (see Section 4.1), where the `ValueTree` takes care of the model and controller aspects.

The `var` class is a variant type analogous to a Javascript `var` [16]. It can be used to store primitive types (`int`, `int64`, `bool`, `double`), JUCE types (`String`, `MemoryBlock`), a `Array<var>`, a `ReferenceCountedObject` or a `DynamicObject`. It can be serialized to JSON [17] and converts between types using the appropriate operators [16].

The `Identifier` class acts like an enum, representing a string for accessing properties of a `ValueTree` by name. The comparison of `Identifier` is a very fast pointer comparison, but creating an `Identifier` can be slow because it is actually a globally pooled `String` ($\mathcal{O}(\log(n))$, where n is the number of existing strings in the pool) [16]. The best way to declare them is statically, in order for them to be created at application startup [16].

```xml
<?xml version="1.0" encoding="UTF-8"?>

<EDIT appVersion="Unknown" projectID="0/883fd9" creationTime="1579867568819"
        modifiedBy="Johannes Wolfgruber" lastSignificantChange="16fd7c72c3b">
    <TRANSPORT recordPunchInOut="0" position="0.0" loopPoint1="0.0"
          loopPoint2="7.321428571428571"/>
    <MACROPARAMETERS id="1001"/>
    <MASTERVOLUME>
        <PLUGIN type="volume" volume="0.6376281380653381" id="1003" enabled="1">
            <MACROPARAMETERS id="1005"/>
            <MODIFIERASSIGNMENTS/>
        </PLUGIN>
    </MASTERVOLUME>
    <INPUTDEVICES>
        <INPUTDEVICE name="2- Scarlett 18i20 USB"/>
        <INPUTDEVICE name="Mic 1" targetTrack="1012" targetIndex="0" armed="1"/>
        ...
    </INPUTDEVICES>
    <VIEWMODEL DetailViewTimeLeft="0.0" DetailViewTimeRight="7.321428571428571"
               DetailTimeResolution="1"/>
    <PROJECTMODEL EditName="1_Test_Test"
        ProjectFolder=".\Documents\MCLS\1_Test_Test"
        ExaminationFolder=".\Documents\MCLS\1_Test_Test\001_24_01_2020"
        AudioFolder=".\Documents\MCLS\1_Test_Test\001_24_01_2020\Audio"/>
    <SPECTROGRAMSETTINGSMODEL/>
    <METADATAMODEL TimeOfExamination="20200124T130608.868+0100" SubjectCode="1"
        FirstName="Test" LastName="Test" ExaminationNumber="1"
        DateOfBirth="19900101T000000.000+0100"
        Gender="0"/>
    <TRACK id="1012" midiVProp="0.28125" midiVOffset="0.359375" name="Sensor1"
            mute="0" InputGain="12.0" Frequency="8317.0" solo="1">
        <MACROPARAMETERS id="1014"/>
        <MODIFIERS/>
        <AUDIOCLIP name="Sensor1 Recording 1" start="0.0"
              length="7.321428571428571"
            offset="0.0" id="1004"
            source="001_24_01_2020\Audio\Raw\1_Render_Test_Sensor1_Take_2.wav"
            sync="0" elastiqueMode="0" pan="0.0" colour="ffff0000">
            <LOOPINFO rootNote="-1" numBeats="14.64285714285714" oneShot="0"
                  denominator="4"
                numerator="4" bpm="0.0" inMarker="0" outMarker="-1"/>
        </AUDIOCLIP>
        <PLUGIN type="highpass" id="1230" enabled="1" frequency="8317.0">
            <MACROPARAMETERS id="1231"/>
            <MODIFIERASSIGNMENTS/>
        </PLUGIN>
        <PLUGIN type="volume" id="1017" enabled="1" remapOnTempoChange="1">
            <MACROPARAMETERS id="1018"/>
            <MODIFIERASSIGNMENTS/>
        </PLUGIN>
        <PLUGIN type="level" id="1021" enabled="1">
            <MACROPARAMETERS id="1022"/>
            <MODIFIERASSIGNMENTS/>
        </PLUGIN>
        <OUTPUTDEVICES>
            <DEVICE name="(default audio output)"/>
        </OUTPUTDEVICES>
    </TRACK>
    ...
</EDIT>
```

Listing 3.1: Example of a XML serialized `ValueTree`.

3.2 Tracktion Engine

The Tracktion Engine is partially open-source since November 2018. As JUCE developed out of Tracktion, the engine is tightly integrated with the framework and is provided as a JUCE module (see Section 4.3). The Tracktion Engine provides a high-level document object model (DOM) and application programming interface (API) for audio applications [5]. Some key features include cross-platform support, fast audio file playback via memory mapping and audio recording [5]. In this section the most important features of the Tracktion Engine that are used in the MCLSR software are explained. If not specified otherwise, all text in monospaced font in this chapter refers to class and method names that are part of the Tracktion Engine codebase.

3.2.1 The Engine

The `Engine` class is the central class of the Tracktion Engine [18]. It is implemented as a singleton, so there exists only one instance. The `Engine` class needs to be created before any `Edit` (see Section 3.2.2) and performs the initialization and shutdown of the Tracktion Engine. To customize the user interface behavior or the engine behavior, subclasses of `UIBehaviour` and `EngineBehaviour` can be passed to the `Engine`, otherwise the `Engine` will use the default behaviors. The `Engine` class is used to hold and retrieve all Tracktion Engine objects that need to be unique per session, such as the `DeviceManager`, `BackgroundJobManager`, the `AudioFileManager`, the `RenderManager` etc.

3.2.2 Data Model

The `Edit` class [19] represents a session containing objects such as tracks, input devices or the undo manager. Sub-objects of the `Edit` hold a reference to it, so they know to which `Edit` they belong. To create an `Edit`, an `Edit::Options` instance with an `Engine`, the `Edit`'s state in the form of a `juce::ValueTree` and a `ProjectItemID` have to be provided. The `Edit` class represents the entire model of the Tracktion Engine. The `Edit` state `juce::ValueTree` can be serialized and deserialized to XML and used as a project file format.

An `Edit` contains a list of `Track` objects. There are a number of different kinds of `Track` types, e.g. a `AudioTrack` for audio output [20]. The `AudioTrack` class is derived from `ClipTrack` and contains a list of all `WaveAudioClip` objects of this track. A `Track` also holds a list of `Plugin` (see Section 3.2.3) objects, which by default has a `VolumeAndPanPlugin` and a `LevelMeterPlugin` for controlling the track volume and metering it respectively. For audio input and output a `AudioTrack` contains pointers to a `TrackOutput` which is a representation of the output device for the track, and to a `WaveInputDevice`, a virtual input device representing a mono or stereo input channel.

A `Clip` object lives inside of a `Track`. As with `Track`s, there exist different kinds of `Clip` types, e.g. a `WaveAudioClip`, which holds a reference to an audio file [20]. The `WaveAudioClip` class is derived from the `AudioClipBase` class. This class is amongst other things responsible for creating proxy objects of the clips audio file. Proxy objects are essentially copies of the corresponding audio files, which are held in memory to optimize playback and manipulation of the files contents (memory access is faster than loading the file from disk every time it is needed).

The `SelectionManager` is used to hold a list of selected objects (of base type `Selectable`) [21]. It is used to select and deselect certain objects, and to notify listeners if the selection changed in order to handle the event accordingly. The `SelectionManager` allows the selection of single objects as well as a list of objects at the same time.

3.2.3 Plugin System

The `Plugin` class represents the Tracktion Engine's internal plugin format, which can be used as insert effect on `Tracks` or `Clips`. The `applyToAudioBuffer()` method can be overridden to do the audio processing on the incoming audio buffer of the track or clip on which the plugin was inserted. The `Plugin` itself does not contain any user interface, but can be accessed from the owning track via its `PluginList` to control the plugin's parameters.

3.2.4 Audio Graph

The underlying idea of the Tracktion Engine is to build up the entire signal path as a graph. The Tracktion Engine audio graph consists of `AudioNode` instances. The `AudioNode` class is the base class for all nodes. It defines, amongst others, the `renderOver()` and `renderAdding()` pure virtual methods, which both accept a reference to the `AudioRenderContext` and are called inside the audio callback. The `AudioRenderContext` contains the destination audio buffer including information about the number of channels, number of samples inside the buffer and the index of the starting sample. With this `AudioRenderContext` the subclasses of the `AudioNode` are able to do their processing inside the above mentioned methods. There exist a number of node types in the Tracktion Engine, e.g.:

- *SingleInputAudioNode:* Contains a pointer to a single input `AudioNode` and passes it on.
- *WaveAudioNode:* Plays back an audio file.
- *MixerAudioNode:* Mixes together a set of parallel input nodes.

- *PluginAudioNode:* Applies a `Plugin`s processing to the `AudioRenderContext`.

The audio graph is organized inside the `EditPlaybackContext` class, which contains methods for creating the graph. The `Edit::TreeWatcher` is used to monitor all properties of the `Edit` state that require a rebuilding of the graph and triggers the rebuild, if one of the properties changes. As this happens automatically, it is generally not needed to rebuild the audio graph manually.

The `DeviceManager` class is responsible for managing the audio devices and the audio callback. This class is essentially a wrapper around the `juce::AudioDeviceManager` class. The `juce::AudioDeviceManager` holds a set of `juce::AudioIODeviceType`s (representing the type of audio driver, such as ASIO, CoreAudio or ALSA) and one currently active `juce::AudioIODevice`. The `juce::AudioIODevice` class is subclassed to implement the different audio protocols. It is used to set the `juce::AudioIODeviceCallback`, which will be called repeatedly on a high-priority audio thread with the data from the specific implementation of the audio protocol. The `DeviceManager` acts and adds itself as a `juce::AudioDeviceIOCallback` to the `juce::AudioDeviceManager` and thus receives the audio data from the selected input device. The audio buffer is then pumped into the active `EditPlaybackContext`s, which in turn distributes it over the audio graph.

3.3 Build Infrastructure

JUCE includes a project generation tool named Projucer, that can be used to generate projects for several different IDEs (Integrated Development Environments) [22]. The Projucer is not ideal for continuous integration and automated builds. Because of that and for a seamless integration with the unit testing framework GoogleTest [23], the choice fell on CMake as the project creation tool [24]. FRUT is a tool that allows to build JUCE projects with CMake [25].

3.3.1 CMake configuration

CMake can be configured with `CMakeLists.txt` files. The MCLSR repository is structured into one top-level `CMakeLists.txt` file in the root folder, and one `CMakeLists.txt` file for each project that outputs an executable in the respective subfolder. The top-level `CMakeLists.txt` file sets the path for the JUCE framework, includes FRUT and the corresponding tools, configures googletest to pull the latest commit and build it into the specified directory and includes the subfolders for the MCLSRApp and MCLSRTests. In the `CMakeLists.txt` files for the two applications, all JUCE related options can be configured, e.g. which modules to include, which source files to include and which project types to create.

3.3.2 Build scripts

In order to automate the build process and testing for continuous integration (CI) two build scripts are provided (a Powershell script for Windows and a Shell script for Unix). The following parameters are available inside the scripts:

- `build`: Builds the project files and copies the executables to the `bin` folder.
- `clean`: Removes the build and binary directories.
- `config`: The build configuration, either "Debug" (default) or "Release".

- `create`: Creates the project files (Visual Studio 2019 solution on Windows, Xcode project on Mac OSX, makefile on Linux) inside the `build` folder. Overwrites the project files if they already exist.
- `generatedoc`: Runs doxygen to generate the HTML and LaTeX documentation from the comments in the code.
- `rebuild`: Runs `clean` and `build`, including a rebuild of FRUT.
- `recreate`: Runs `clean` and `create`, including a rebuild of FRUT.
- `test`: Runs the Unit Tests if the project was previously built.

The script can for example be called from the command line via `./build.sh --config Release --rebuild` (Unix) or `./build.ps1 -config Release -rebuild` (Windows). For continuous integration an Azure Pipeline was configured, so that with every commit to the repositories master branch, the build server pulls the commit and runs the build script and tests for all three operating systems, reporting errors and warnings.

4 Software Architecture

This chapter shows the considerations regarding the architecture of the application. The first section describes the Model-View-Controller architectural pattern and how it is used and adapted in JUCE. Additionally some key architectural concepts concerning JUCE are explained. The second section describes the project structure that underlines the architecture and parts of the software with different responsibilities. The last section shows the JUCE module format and how it was used to create the different modules.

4.1 MVC and JUCE architecture

The Model-View-Controller (MVC) architectural pattern defines three roles that are assigned to objects in the application: model, view and controller [26]. The pattern additionally defines how these roles interact with each other, as can be seen in Figure 4.1.

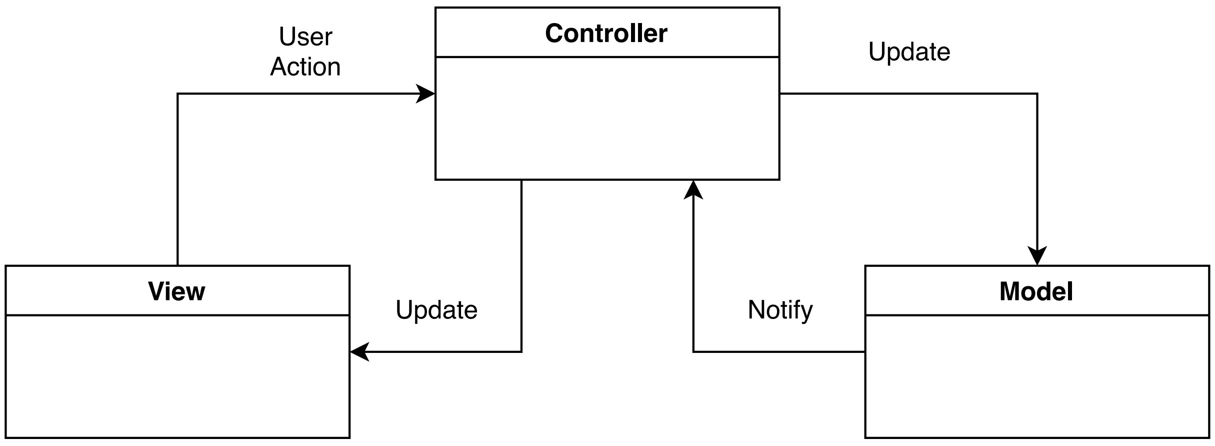


Figure 4.1: The MVC architectural pattern.

According to [26], Model objects contain the data of the application and the logic that manipulates and processes the data. Model objects can have one-to-one or one-to-many relationships with other model objects, and thus consists of one or several object graphs [26]. The persistent state of the application should be held inside model objects after loading the data into the application [26]. Model objects should have no direct connection to view objects and the communication happens across the controller object [26].

View objects are objects that can be seen by the user, contain the logic to draw themselves and respond to user actions [26]. They display the data from the model layer and allow the editing of that data.

As stated in [26], Controller objects coordinate the communication between one or more view objects and one or more model objects. They can also execute setup and coordinating tasks for the application and manage the lifetime of other objects [26]. Upon a change from the view layer via some user action that creates or changes data, the controller creates or updates the corresponding model object. When a model object changes, it notifies the controller, which in turn updates the view object.

The MVC pattern ensures a clean separation of the user interface, the application logic and the domain logic. The JUCE framework is mostly developed with the MVC pattern in mind but with a less strict separation of the view and controller roles. The `juce::Component` base class acts as both the view and the controller. All user interface elements are derived from the `juce::Component` class and contain the logic to paint and layout themselves and their child components. Additionally they contain all of the application logic for their respective use. Model objects contain underlying `juce::ValueTree` encapsulating persistent data as well as runtime state information. Components can subscribe to changes of the tree and update their view accordingly. Conversely the model object is updated upon user actions that modify the state or data of the application.

4.2 Multi-Channel Lung Sound Recording Software Architecture

To ensure a better separation between the different parts of the architecture and make the code reusable and testable, four JUCE modules (see Section 4.3) were defined and assigned different responsibilities. The core module encapsulates classes and functionality that are shared across all other modules like constants, binary data, helper and wrapper classes. The model module holds all model objects and accompanying logic. The domain module includes domain specific logic, e.g. for spectrogram calculation, sample rate conversion, input device and track management. The application module depends on all modules and contains the UI components and application logic, look and feel classes and the application state. Figure 4.2 shows an overview of the architecture and the module dependencies. As can be seen, the main application depends on all modules. The module format allows to reuse the same modules for the test application with the additional dependency to the test framework GoogleTest [23].

4.3 JUCE modules

The JUCE code base is divided into JUCE modules according to responsibilities and functionality. A JUCE module is a collection of header and source files, that can easily be added to a project if the corresponding classes or libraries are needed [27]. This way only the necessary parts of JUCE are included into the project and it does not bloat the project size and compilation time. The JUCE module format [27] states the things that need to be done in order to implement a user module. This includes naming conventions for files and a header file that acts as the module definition with a predetermined comment section at the top of the file with information about the author, module name and dependencies. This header file includes all header files of the module, the corresponding source file includes all source files.

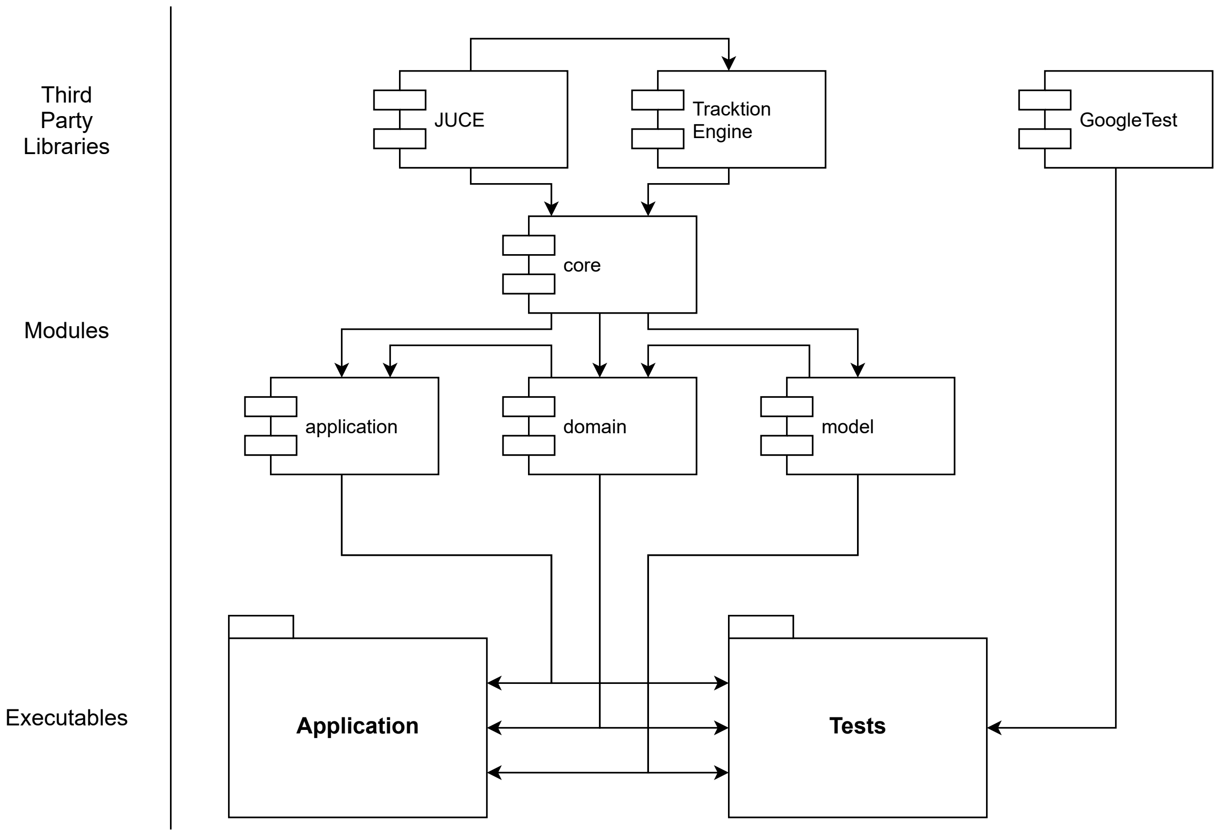


Figure 4.2: High level architectural overview of the MCLSR software.

5 Software Implementation

This chapter is a detailed description of the implementation of the Multi-Channel Lung Sound Recording software. Some important software design patterns and best practices that were used during development are explained alongside the implementation details of the core, domain, model and application modules. The last section of this chapter describes the design and implementation of the user interface. Additionally a class documentation is available in HTML and LaTeX format alongside the source code.

5.1 Design Patterns and Best Practices

A software design pattern offers generic solutions to commonly occurring problems, such that they can be applied in a multitude of different situations [28, p. 2]. This section shows some important design patterns that are used throughout the application and how they are implemented.

5.1.1 Singleton Pattern

The singleton pattern is, according to [28, p. 127], a creational pattern with the intent to enforce that only one instance of a certain class can be created and accessed from one global point. The singleton encapsulates its sole instance and can therefore control creation and access of it. To ensure a unique instance, the singleton class normally hides the operation that creates the instance behind a static or class method, that ensures that only one instance is created.

A variation of this pattern is the `juce::SharedResourcePointer`, which was used for several classes in the MCLSR software. This class template is a smart-pointer that internally creates and manages the lifetime of a shared static instance of a class [29]. The implementation differs compared to a singleton, because it counts the references to the shared object to make sure that it is created and deleted correctly [29].

5.1.2 Observer Pattern

The observer pattern is a behavioral pattern that defines a dependency between one object acting as the broadcaster and a number of different objects acting as listeners, so that when the broadcaster object changes state, all its listener objects are notified [28, p. 293].

This pattern is used extensively in JUCE with their broadcaster/listener system. The listener defines a callback function that can be invoked by the broadcaster after the listener registers with the broadcaster [30]. With `juce::ValueTree` this functionality is further abstracted and allows to subscribe to any changes in a tree. This tree is encapsulated inside the `ConstrainedValue` class (see Section 5.2.3), which subscribes itself as an observer and gets notified on any changes.

5.1.3 Abstract Factory Pattern

As stated in [28, p. 87], the abstract factory pattern provides a way of instantiating related or dependent objects without stating their concrete classes.

This pattern is used for creating different objects of the same base type at runtime and encapsulates the logic of creation of related objects. An example usage of the factory pattern can be seen in Listing 5.1. Although the `RecordingClipComponent` and `SpectrogramComponent` are not direct derivatives of the `ClipComponent`, but are contextually related to it, the `ClipComponentFactory` contains factory methods to create instances of these two classes as well as the different `ClipComponent` types.

```
ClipComponent* Create(ClipComponentType type, te::Clip::Ptr clip)
{
    switch (type)
    {
    case TypeOfClipComponent:
        return new ClipComponent(clip);
    case TypeOfAudioClipComponent:
        return new AudioClipComponent(clip);
    case TypeOfAudioDetailClipComponent:
        return new AudioClipDetailComponent(clip);
    default:
        throw;
    }
}

RecordingClipComponent* CreateRecordingClipComponent(te::Track::Ptr track)
{
    return new RecordingClipComponent(track);
}

SpectrogramComponent* CreateSpectrogramComponent(te::Clip::Ptr clip)
{
    return new SpectrogramComponent(clip);
}
```

Listing 5.1: The `ClipComponentFactory` class.

5.1.4 Best Practices in Audio Development

This section lists some best practices to consider when developing an audio software application:

- Avoid the `new` and `delete` operators for object instantiation to prevent memory leaks. Modern C++ code should instead make use of the *Resource Acquisition Is Initialization* (RAII) pattern, which ensures that the life-cycle of a resource, which must be acquired before use, is tied to the lifetime of the owning object [31]. This is done using stack allocations wherever possible and instead of raw pointers switch to smart pointers from the C++ standard library (`std::unique_ptr` and `std::shared_ptr`). Smart pointers underline the scope of the pointers and get automatically deleted once they go out of scope or the reference count is zero.

- Complex or long lasting calculations should not be done on the UI thread to keep it from blocking for a longer period of time, which would render the user interface unresponsive. Instead these calculations should be pushed to their own dedicated background threads, notifying and updating the UI asynchronously when finished. Wherever needed, a progress indicator can be used to let the user know about a background activity.

- There should be no locking on the audio thread. The audio thread itself is a high-priority thread, which gets called regularly from the audio driver with the new audio buffer. Locking can lead to deadlocks and priority inversion of threads, which in turn leads to audio dropouts. This rule also means, that there should not be any memory allocations done on the audio thread, because they can also cause a locking internally. Additionally it should be considered which data structures to use on the audio thread, because operations like resizing a `std::vector` also lock the executing thread. Another source of hidden locks are logging operations or calls to the operating system and they should be avoided from the audio thread.

- Real-time audio processing has to be as performant as possible. This entails, that all calculations happening inside one audio callback need to be finished before the next call to avoid dropouts in the audio stream. The available time is controlled by the buffer size of the audio device. A buffer size of e.g. 128 samples at a sampling rate of $f_s = 48kHz$ allows a time of $\tau = 2.67ms$ until the next call.
- To allow for easier extensibility of the application and to keep the separation of concerns as high as possible, the data model of the application should be kept separated from the user interface code. This is enforced by the MVC architectural pattern (Section 4.1).

5.2 Core Module

The core module contains binary data (fonts and vector graphics), constants, wrapper classes and helper functions that all other modules need to use. Additionally it holds classes for handling application state, application commands and managing the Tracktion Engine `te::Edit` (Section 3.2.2) class. Figure B.1 shows the file structure of the core module.

5.2.1 Constants and Identifiers

The file `Constants.h` contains global (inside the `mclsr` namespace) constants for user interface element's dimensions. This way, without any hard coded values for the dimensions, there is only one place to define and change these values, which can be accessed from everywhere in the application. Changes to the user interface can be applied quickly without the need to refactor code in several places.

The file `CommandIDs.h` contains an enumeration with identifiers for the application commands. The range of this enumeration starts with `0x300000` to avoid interference with predefined commands from JUCE and Tracktion Engine, and ends with `lastCommandIDEntry` to get the number of commands defined. Application commands are application wide commands which are assigned to keyboard shortcuts. This includes shortcuts for saving, toggling playback and for creating new examinations.

The file `Identifiers.h` contains definitions of `juce::Identifiers` that represent a whole `juce::ValueTree` or a property of one.

5.2.2 Utilities

The file `Utilities.h` contains static helper functions that can be used throughout the application. Listing 5.2 shows the function's signatures of the helper functions, divided into namespaces for general helper functions and Tracktion Engine related helper functions. The `Helper` namespace contains a function to add an array of child components to the provided parent component, which is useful for components with a lot of child components. The Tracktion Engine helper functions provide quick and easy ways to access transport and track input features, such as looping around a clip, checking if a track is armed and arming a track for recording, check if input monitoring is enabled and enabling it and checking wether a track has an input or not.

```cpp
namespace Helpers
{
    static void AddAndMakeVisible(Component& parent,
                                  const Array<Component*>& children);
}

namespace EngineHelpers
{
    template<typename ClipType>
    typename ClipType::Ptr LoopAroundClip(ClipType& clip);

    void ArmTrack(te::AudioTrack& t, bool arm, int position = 0);

    bool IsTrackArmed(te::AudioTrack& t, int position = 0);

    bool IsInputMonitoringEnabled(te::AudioTrack& t, int position = 0);

    void EnableInputMonitoring(te::AudioTrack& t, bool im, int position = 0);

    bool TrackHasInput(te::AudioTrack& t, int position = 0);
}
```

Listing 5.2: Definitions of the utility functions.

5.2.3 Wrapper Classes

The class `FlaggedAsyncUpdater` is derived from `juce::AsyncUpdater` (see Section 3.1.2) and allows handling an asynchronous update depending on wether a flag is set to true or false.

The `Delegate` class is a templated wrapper around `std::function<void(Args...)>`. The class wraps the check if the `std::function<void(Args...)>` is null, effectively only executing if it is not null. This allows for less verbose code when dealing with callback functions. Listing 5.3 shows an example for using the `Delegate` class with a lambda function. The function call in line 8 does not need a null check because it is done internally inside the `Delegate` class.

```cpp
1  Delegate<int, int, int> exampleCallback;
2
3  exampleCallback = [](int a, int b, int c)
4  {
5      // callback code
6  };
7
8  exampleCallback(1, 2, 3);
```

Listing 5.3: Example usage of the `Delegate` class.

The `ConstrainedValue` class wraps around a property of a `juce::ValueTree`. The `Delegate` class is used to register lambda functions to the callbacks of the `juce::ValueTree`. For better performance the value of the property is cached and can be filled with a default value in case the property does not preexist in the provided tree. Additionally a templated `Constrainer` class can be provided in order to constrain the underlying property to a certain range. This `Constrainer` defaults to not constraining the value.

5.2.4 Progress and Progress List

The `Progress` class represents a named progress, which can be cancelled and is used by the `ProgressList` class. It contains a `std::atomic<float>` for the current progress (between 0.0 and 1.0), and a `std::atomic<bool>` indicating wether it was cancelled or not. The `ProgressList` class manages a list of `Progress` objects. It includes a method to get the cumulated progress from all tasks and a callback called when the list changes. These classes are used for reporting the progress when exporting and downsampling the audio files.

5.2.5 Rendering Task Runner

The `RenderingTaskRunner` class is derived from the `juce::Thread` (see Section 3.1.2) class and takes a `te::Renderer::RenderTask` and a `Progress` as input arguments. It spins up a background thread, running the rendering job until finished and reporting the current progress via the `Progress` object during execution.

5.2.6 Application Commands

The `ApplicationCommandHandler` class is derived from `juce::ApplicationCommandTarget` and is used to set up all application wide commands and expose them via `Delegate` callbacks. Inside the overridden `getCommandInfo()` method the commands are associated with keypresses and additional information (command name and description). In total there are three application wide commands defined: New Examination, Save Examination and Play/Pause. These can be invoked via Ctrl+N, Ctrl+S and Space respectively. The `perform()` method is overridden to execute the corresponding `Delegate` callback.

5.2.7 Edit Manager and Application State

The `EditManager` class is one of the key parts of the application. There is only one instance of the `EditManager`, which can be accessed by declaring a `juce::SharedResourcePointer<EditManager>`. The main functionality of the `EditManager` is to manage the creation and lifetime of `te::Edit` (see Section 3.2.2) objects. Additionally the class provides methods from the current `te::Edit` object. Because the `EditManager` has basically the same lifetime as the application itself, it also creates the `juce::ValueTree` for the applications runtime state. The `EditManager` provides access to the current samplerate, blocksize, the `UIBehaviour`, the `te::DeviceManager` and can rebuild the audio graph if needed.

The `ApplicationState` class uses the `EditManager` to get the application state and provides all properties that do not need to be persisted and callbacks for when those properties change. The properties of the application state include the current state of the project creation (or loading), wether the expert mode is on or off, wether the `te::Edit` has changed since it was last saved and wether the track view should follow the playhead during playback.

The `EditCreationState` enum represents the current state of the `te::Edit` creation or loading and is necessary to trigger the appropriate callbacks in a defined and comprehensible way. At first the state is in the `InitialState`. When `CreateNewEdit` or `LoadEdit` is called, the state switches to `BeginCreation`, which can be used to clean up anything possibly open from a previously loaded `te::Edit`. After the `te::Edit` object is created, the `BindValues` state signals that the `juce::ValueTree` of the `te::Edit` exists and can be used to bind all properties. The `UpdateModel` state stores all new user input into the model, the `FinishCreation` state is used to clean up the creation process. The final state is the `EditLoaded` state which signals that everything is set up and the application is ready.

5.3 Domain Module

The domain module contains the logic that solves the softwares problem domain. In software engineering this is often called the business logic as compared to the application logic. The latter is concerned with the softwares behavior without touching the problem domains as such. The business logic on the other hand contains everything to solve the problem domains. In the case of the MCLSR application this includes the algorithms and signal processing that are needed to record and save the audio, calculate spectrograms, convert the samplerate, the high-pass filter and frequency analyzer. Figure B.2 shows the file structure of the domain module. This section describes the signal processing used in each of the problem domains as well as their implementation.

5.3.1 Tracks and Inputs Manager and Transport Manager

Recording and playback of audio files are a large part of the Tracktion Engine. The `TracksAndInputsManager` and `TransportManager` classes extract and wrap the important parts for handling tracks and audio inputs and the transport respectively. The class diagrams are shown in Figure C.1. The `TracksAndInputsManager` provides methods to add and delete tracks, retrieve tracks and input devices and to assign inputs. The high-pass filters are added to all tracks and can be retrieved individually. The cutoff frequency and gain can be set for an individual track or all tracks at once. The `TransportManager` can be used to interact with the `te::PlayHead` (get and set the current position, set a looping range) or call methods on the `te::TransportControl` (toggle recording and playback, check if the engine is playing or recording).

5.3.2 Highpass Filter Plugin

The `HighpassFilterPlugin` (requirement **R.3**) is derived from the `te::Plugin` class. The class diagram can be seen in Figure C.4. In the `initialise()` method, which is called on all `te::Plugin` objects by the Tracktion Engine, the sample rate is set, the filter state is reset and the `FrequencyAnalyzer` (see Section 5.3.3) is set up. The `UpdateFilter()` method is used to calculate and set the filter coefficients if the filters cutoff frequency has changed. The filter is implemented as a cascade of Biquad filters using the `juce::dsp::IIR::Filter` class and the `juce::dsp::FilterDesign::designIIRHighpassHighOrderButterworthMethod()` method.

The JUCE IIR filter is implemented with the transposed Direct Form II, which can be seen in Figure 5.1. The designing method calculates the Q-factor for each stage i of the cascade as follows:

$$Q_i = \frac{1}{2cos((2i+1) \cdot \frac{\pi}{2N})}, \quad \text{for } i = 0, 1, \tag{5.1}$$

where $N = 4$ is the filter order to get the desired slope of $24dB/oct$ (two cascaded second order

Biquad filters). The coefficients are then calculated by:

$$n = tan\left(\frac{\pi f_c}{f_s}\right), \tag{5.2}$$

$$b_0 = \frac{1}{1 + {}^1\!/_{Q_i} \cdot n + n^2}, \tag{5.3}$$

$$b_1 = -2 \cdot b_0, \tag{5.4}$$

$$b_2 = b_0, \tag{5.5}$$

$$a_0 = 1, \tag{5.6}$$

$$a_1 = 2 \cdot b_0 \cdot (n^2 - 1), \tag{5.7}$$

$$a_2 = b_0 \cdot (1 - {}^1\!/_{Q_i} \cdot n + n^2). \tag{5.8}$$

The plugin's `applyToBuffer()` method is called regularly from the audio callback with the next buffer that needs processing. The `te::AudioRenderContext` passed to the callback contains the buffer, the index of the start sample and the number of samples that should be processed. Figure 5.4 shows the audio callback of the `HighpassFilterPlugin`. After the filter coefficients are updated (line 5), the buffer's channels are cleared (line 7), except for the first one, which holds the relevant audio data. The `for`-loop iterates over all samples and processes each one with the filter cascade (lines 16-21). Because the input is a mono signal it is just copied to the right channel. After the filter is applied, the buffer values are clipped if they are not in the range $-3 \leq x \leq 3$ (line 24). Then the buffer is added to the `FrequencyAnalyzer` to get the spectrum after applying the high-pass filter (line 26).

```
 1  if (renderContext.destBuffer != nullptr)
 2  {
 3      SCOPED_REALTIME_CHECK;
 4
 5      UpdateFilter();
 6
 7      te::clearChannels(*renderContext.destBuffer, 1, -1,
           renderContext.bufferStartSample, renderContext.bufferNumSamples);
 8
 9      for (int sample = renderContext.bufferStartSample; sample <
           renderContext.bufferNumSamples; ++sample)
10      {
11          auto input = renderContext.destBuffer->getReadPointer(0, sample);
12
13          auto outputL = renderContext.destBuffer->getWritePointer(0, sample);
14          auto outputR = renderContext.destBuffer->getWritePointer(1, sample);
15
16          *outputL = static_cast<float>(_filter[0]->processSample(*input));
17          for (int i = 1; i < _filter.size(); ++i)
18          {
19              *outputL = static_cast<float>(_filter[i]->processSample(*outputL));
20          }
21          *outputR = *outputL;
22      }
23
24      te::sanitiseValues(*renderContext.destBuffer,
           renderContext.bufferStartSample, renderContext.bufferNumSamples,
           3.0f);
25
26      Analyzer.AddAudioData(*renderContext.destBuffer);
27  }
```

Listing 5.4: The `applyToBuffer` method of the `HighpassFilterPlugin`.

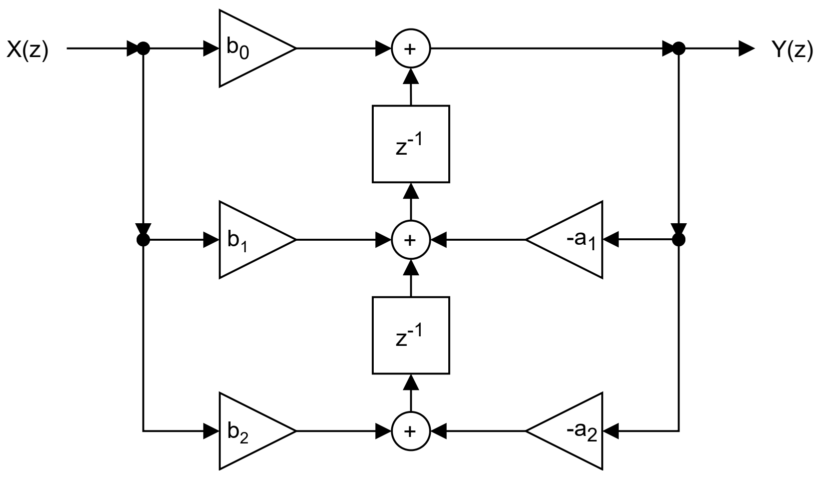


Figure 5.1: Transposed Direct From II.

5.3.3 Frequency Analyzer

The `FrequencyAnalyzer` class (requirement **R.5**) is used to calculate the spectrum of the input buffer and fill a `juce::Path` to be drawn inside a component. It is derived from `juce::Thread` in order to execute the calculations on a background thread. The FIFO buffer of the analyzer is filled with a call to the `AddAudioData()` method. The background thread checks periodically if enough samples of the FIFO are ready to compute the FFT, if not the case the thread waits. Otherwise, the data is copied into the FFT buffer, multiplied with a Hann window and transformed into the frequency domain with `performFrequencyOnlyForwardTransform()`. This method computes the magnitude frequency spectrum [32] and stores it inside the FFT buffer. The length of the FFT is fixed to $N = 4096$ to get a reasonably good frequency resolution. In order to smooth the spectrum, it is averaged over 4 frames. This happens with the help of a second buffer, with 5 channels, where the first channel contains the spectrum to plot, and the rest contains the spectra of 4 consecutive frames. The smoothed spectrum is calculated as follows:

$$|\boldsymbol{X}[k]| = \frac{1}{4}\sum_{i=1}^{4}\frac{1}{N}|\boldsymbol{X}[k+i]|, \tag{5.9}$$

where $|\boldsymbol{X}[k]|$ is the magnitude spectrum of the k-th frame and N is the number of frequency bins of the FFT. The `CreatePath()` method takes the smoothed spectrum and adds a line between each of the data points.

5.3.4 Spectrogram

A spectrogram is used as a tool for the spectral analysis of the audio signals. It is defined as an intensity plot (usually in dB) of the Short-Time Fourier Transform (STFT) magnitude. The STFT is a sequence of FFTs of windowed data frames, which are allowed to overlap. The mathematical definition of the STFT uses the Discrete Time Fourier Transform (DTFT) and looks according to [33] as follows:

$$
\begin{aligned}
X_m(\omega) &= \sum_{n=-\infty}^{\infty} x[n]w[n-mR]e^{-j\omega n} \\
&= DTFT_\omega \left\{x \cdot SHIFT_{mR}(w)\right\}, \qquad (5.10)
\end{aligned}
$$

where $x[n]$ is the input signal at time n, $w[n]$ denotes the window function of length M, $X_m(\omega)$ is the DTFT of the windowed data centered at time mR, and R is the hopsize (in samples) between successive DTFTs.

In practice, the theoretical definition is implemented using a succession of FFTs of windowed data frames in the `Spectrogram` class (requirement **R.7**). A `for`-loop iterates over the frames of the audio signal, where the total number of frames is the total number of samples divided by the hopsize R. A `juce::AudioFormatReader` is used to extract the correct frame from the audio file (where the length of the frame is the FFT length and the starting sample for the reader is mR). The frame gets multiplied by the window and a FFT is performed with `performFrequencyOnlyForwardTransform()`. For plotting the spectrogram, the values of the FFT are mapped onto a grey scale and set as pixels inside `juce::Image` with dimensions $(w \times h)$ = (number of frames $\times$ $\frac{N}{4}$), where N is the size of the FFT.

5.3.5 Sample Rate Conversion

The sample rate conversion to a sample rate of $f_s = 16kHz$ (requirement **R.14**) is done in a two step process. The first step is the filtering of the input signal with a steep FIR filter to avoid aliasing after the decimation. For calculating the filter coefficients JUCE provides the static method `designFIRLowpassKaiserMethod()` which accepts the cutoff frequency, sample rate, the normalized transition width and the stop band attenuation in dB. Internally the filter is designed using the windowing method with a Kaiser window. The calculations of the β-parameter and the size of the window follow the design formulas from [34]:

$$
\beta = \begin{cases} 0.1102 \cdot (\alpha - 8.7), & \text{if } \alpha > -50, \\ 0.5842 \cdot (\alpha - 21)^{0.4} + 0.07886 \cdot (\alpha - 21), & \text{if } 21 \leq \alpha \leq 50, \\ 0, & \text{otherwise,} \end{cases} \qquad (5.11)
$$

where $-\alpha$ is the relative sidelobe attenuation (in negative dB). The filter order can then be calculated with:

$$
N = \frac{\alpha - 7.95}{2.285 \cdot 2\pi\Delta f}, \qquad (5.12)
$$

where Δf is the transition width between pass band and stop band. To avoid aliasing for a target sampling frequency of $f_s = 16kHz$ the stop band has to start at the Nyquist frequency $f_N = \frac{f_s}{2} = 8kHz$. A transition width of $\Delta f = 400Hz$ and a sidelobe attenuation of $\alpha = 60dB$ is chosen, which results in a cutoff frequency of $f_c = 7.6kHz$ (same as in [1]), a filter order of $N = 436$ and $\beta = 5.6533$. The frequency response of the anti-aliasing low-pass filter can be seen in Figure 5.2.

The second step is a linear interpolation between fractional buffer positions. The conversion ratio is computed from the difference in size of the input and output buffer. For a decimation from $f_{S_{in}} = 48kHz$ to $f_{S_{out}} = 16kHz$ and an input buffer size of $N_{in} = 512$, a output buffer of size $N_{out} = N_{in} \cdot \frac{f_{s_{out}}}{f_{s_{in}}} = 512 \cdot \frac{16}{48} = 170.67 \approx 171$ is used, which results in a conversion ratio of $r = \frac{512}{171} = 2.9942$. The interpolated value y at the fractional buffer position ξ of the input x is then computed as:

$$y = x[u] \cdot (\xi - l) + x[l] \cdot (1.0 - (\xi - l)), \tag{5.13}$$

where $l = \lfloor x \rfloor$ and $u = l + 1$. The next fractional buffer position to compute is $\xi_{n+1} = \xi_n + r$. The samples in between are skipped, which results in the decimated signal.

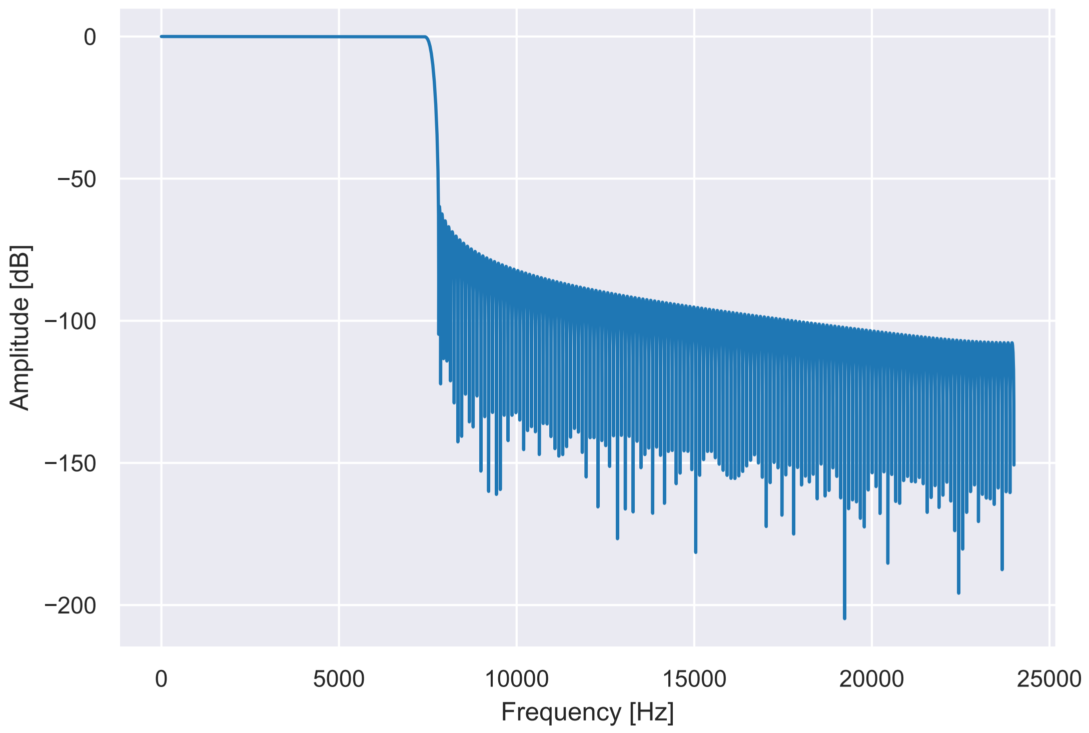


Figure 5.2: Frequency response of the anti-aliasing filter.

5.4 Model Module

The model module contains all model classes related to persistent storage as well as the logic for validating user input and converting between objects and `juce::ValueTree` properties. Figure B.3 shows an outline of the file structure of the model module.

5.4.1 The Model Classes

The basic idea of a model class is, to hold a `juce::ValueTree`, which is a child of the `te::Edit`'s `juce::ValueTree`. This `juce::ValueTree` contains all properties specific to the responsibility of the model class. All model classes are derived from the `ModelBase` class, which exposes some methods that can be overridden and holds the `juce::ValueTree` (protected `_state` member), the application state (protected `_appState` member) and the `EditManager` instance (protected `_editManager` member). The `ModelBase` class registers a callback on the `OnEditCreationState Delegate<EditCreationState>` from the application state, which calls the `OnEditCreation()` method. This way, by overriding the `OnEditCreation()` method, the model class can decide what happens during the different stages of the `te::Edit` creation. The default implementation of the base class calls the `BindValues()` method. The `BindValues()` method has to be overridden to refer the model's properties to the corresponding `juce::ValueTree` and add callbacks. It is important to call this method once the `te::Edits` `juce::ValueTree` is created (in the `BindValues` state of the `EditCreationState` enum). Figure C.6 shows the class diagram of the different model classes. The `MetaDataModel` holds the subject's metadata, the `ProjectModel` provides all project specific data like filepaths. The `SpectrogramModel` stores the parameters of the spectrogram and the `ViewModel` encapsulates all view specific data like track width or time resolution.

5.4.2 Converters and Validation Services

The `Converters.h` file contains the `DateConverter` and `FftSettingsConverter` classes. The `Date Converter` provides methods to convert `juce::Time` objects to integers which are used as indices for the `juce::ComboBoxes` of the metadata. The `FftSettingsConverter` holds methods to convert `juce::ComboBox` indices to a double (for the overlap) and a `juce::dsp::WindowingFunction<float>::WindowingMethod` (for the window), which are needed for the spectrogram calculation.

5.5 Application Module

The application module contains all components that make up the user interface and interact with the model and domain. Additionally the module provides some classes summed up under the term infrastructure, with the application logic that is not inherent to the components themselves. This includes classes for handling the runtime state of tracks, track headers and user interface object selection, as well as a `juce::Viewport` derivative, icon names and helper functions and a subclass of `te::UIBehaviour`. Figures B.4 to B.7 show the folder structure of the application module.

5.5.1 Components

Figure 5.3 shows a dependency graph of all components of the application. Every component needs to have a parent component and can contain multiple child components. The top level component is the `MainWindow`, which owns and is completely filled by the `MainComponent`. All components are derived from the `ComponentBase` class, which is in turn derived from the `juce::Component` class. The `ComponentBase` class provides access to the `ApplicationState`, `ViewModel`, `SelectionState` and `EditManager` and makes it necessary for the subclass to implement the `paint()` and `resized()` methods. Additionally some `ColourIds` are defined that affect all components of the application. The components are grouped into subfolders by their purpose as follows:

- *Clips:* This subfolder contains all components representing `te::Clip` objects as well as the `ClipComponentFactory` class for creating clip component objects. The `ClipComponent` class is a base class, defining some colors for the clips and a `te::Clip::Ptr` member, that holds a reference to the represented clip. The `AudioClipComponentBase` class is derived from the `ClipComponent` and adds functionality to draw audio waveforms (requirement **R.6**). This class is subclassed to implement the `AudioClipComponent`, which defines the selection logic for the audio clips, and the `AudioClipDetailComponent`, which contains a `TimelineComponent` and a `PlayheadComponent` as well as the logic to zoom in and out of the audio waveform. The `RecordingClipComponent` represents a clip during recording.

The `Spectrogram- Component` displays the spectrogram of the selection underneath the corresponding `Audio- ClipDetailComponent`.

- *MetaData:* The `MetaDataComponent` and `CoreMetaDataComponent` are both subclasses of the `MetaDataComponentBase` and represent the available metadata of the patient (requirement **R.8**). The base class contains the logic to setup the comboboxes and update the model. The `CoreMetaDataComponent` is used before project creation to enter all the necessary data of the subject, the `MetaDataComponent` is used to represent the data inside the settings pane.

- *Recorder:* This folder contains the `MainComponent` and its child components related with the audio recording process. The `HeaderComponent` represents the header bar at the top of the window and contains the `TransportComponent` including the widgets to control the application's transport, the `ProjectControlsComponent` for controlling project related functions, the settings button to open and close the settings pane, and the `TransportClock` to display the current time of the playhead. The `RecorderComponent` contains the `MicrophoneComponent`, the `TrackToolsContainerComponent` and the `TrackViewComponent`. The `MicrophoneComponent` is a visual representation of the auscultation pad with its sensors and lets the user enable or disable sensors depending on their individual use case (requirement **R.4**). The `TrackToolsContainerComponent` contains a level meter, a spectrum analyzer and the high-pass filter frequency and phase response, as well as controls to adjust the input gain and filter frequency of the selected track (requirements **R.2**, **R.3**, **R.5**). The `TrackViewComponent` contains a time-based representation of the sensors. Each sensor is represented by a track containing the recorded audio clips (requirement **R.4**). A `TimelineComponent` at the top shows the time axis, the `PlayheadComponent` shows the current position of the playhead. When selecting a clip or parts of it, the selected area is shown as a clip with corresponding spectrogram underneath the `RecorderComponent`. The detail view can be enlarged or reduced by dragging the upper edge up or down respectively.

- *Settings:* The settings contain the metadata, the spectrogram settings (FFT size, hopsize and window) and controls for selecting and configuring the audio device and driver. The `SettingsComponent` holds all settings and can be shown or hidden via the settings button on the right edge of the `HeaderComponent`.

- *Tracks:* This folder contains the components representing `te::Track` objects. The `TrackComponent` represents the time-based view of one sensor and holds an audio clip (requirement **R.4**). All `TrackComponents` are stacked vertically inside the `TrackContainerComponent`. This component is in turn condensed together and made scrollable with the corresponding `TrackLabelComponents`. The `TrackHeaderComponent` contains the logic to enable or disable the recording of the corresponding track, the `TrackToolsComponent` contains, as stated above, several tools concerning the track.

- *Widgets:* The `IconPushButton` and `IconToggleButton` are derived from the `juce::TextButton` and can be used with an icon instead of text. The `TransportClock` displays the current time of the playhead. The `ProgressBarComponent` is a progress bar overlay covering the whole window used for displaying the progress during exporting and conversion of audio files. The `LevelMeterComponent` and the `FrequencyResponseComponent` are used inside the `TrackToolsComponent` and display the current input level as a bar graph and the frequency/phase response as well as the spectrum of the input signal.

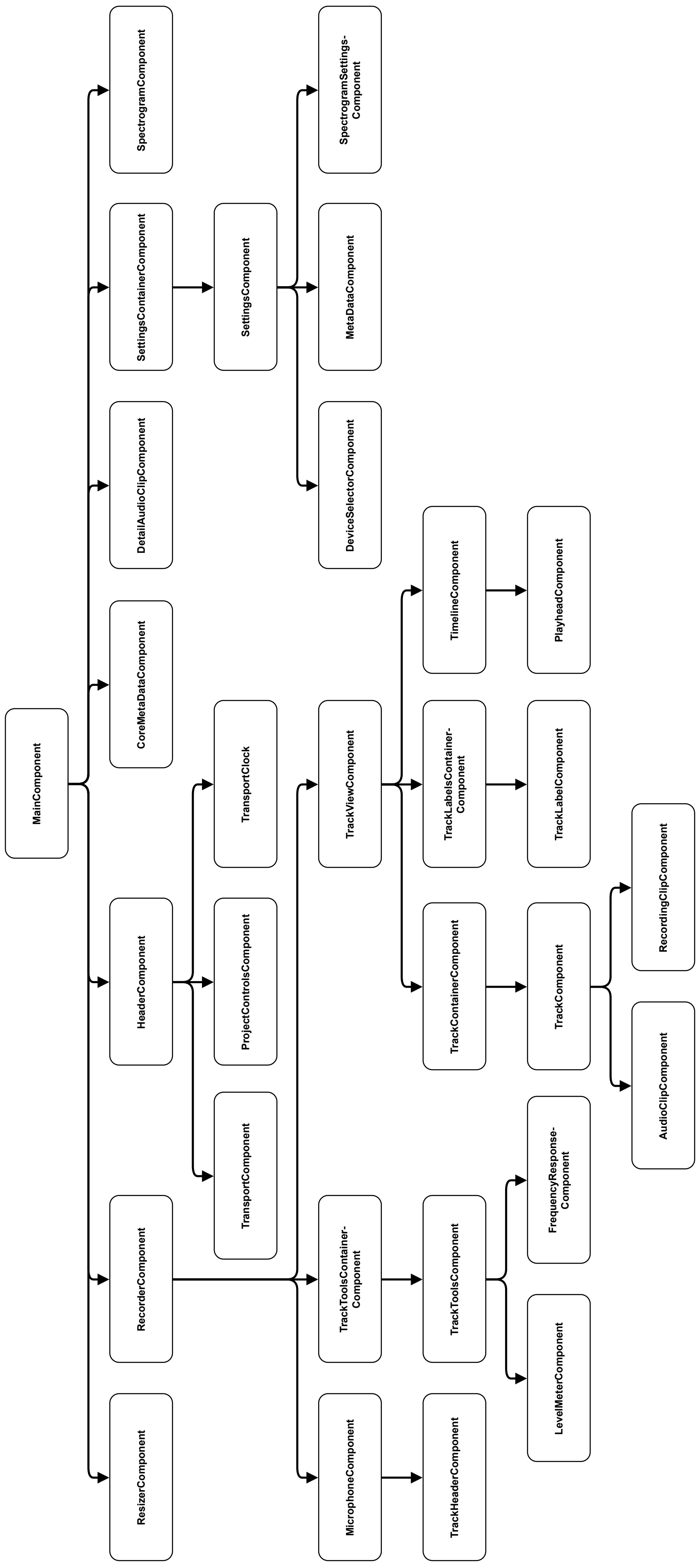


Figure 5.3: Graph of the components.

5.5.2 Infrastructure

The Infrastructure folder contains classes related to the applications runtime state and behavior, which are used by the components. The following is a list with a description for each of the classes:

- *NotifyingViewport:* The `NotifyingViewport` class is derived from `juce::Viewport` and is used as a container class for components that are bigger than the container and need scrolling. It provides callbacks for when the horizontal or vertical scroll position changes providing the new scroll position. This class is mainly used to automatically synchronize the scrolling of the timeline and track labels with the track view.
- *SelectionState:* The `SelectionState` class (requirement **R.9**) is a wrapper for the `te::SelectionManager` class. It contains a singleton `te::SelectionManager` instance and provides methods to interact with it, e.g. `GetSelectedObject()`, `IsSelected()` and `NotifyOfSelectionChange()`. The `SelectionState` subscribes to changes of the `te::SelectionManger` and notifies listeners via the `OnObjectSelected` and `OnObjectDeselected` `Delegate`s. It also contains the logic to enable input monitoring for the selected track and to solo it.
- *TrackState:* The `TrackState` class represents the runtime state of the `TrackComponent`. It provides callbacks for when clips are added, removed or change and when the `te::TransportControl` changes state (for recording).
- *TrackToolsState:* The `TrackToolsState` class represents the runtime state of the `TrackToolsComponent` and provides methods to set the input gain and filter frequency of the corresponding track.
- *UIBehaviour:* The `UIBehaviour` class is a subclass of the `te::UIBehaviour` class which describes the Tracktion Engine behavior related to UI. It uses the default implementations for all methods except the `runTaskWithProgressBar()` methods, which is used to overlay a progress bar during rendering and downsampling of the audio files.

5.6 User Interface

The main requirements for the user interface are a modern look and feel and an intuitive control experience. To achieve these requirements the Google Material Design Guidelines and Color Tool were used for designing the user interface of the application. The Google Material Design Guidelines are a set of guidelines, components and tools for user interface design, made for Android and web applications and utilized in a lot of modern applications [35, 36].

5.6.1 Look and Feel

The `juce::LookAndFeel` class defines the appearance of all JUCE user interface elements and can be subclassed to create a custom look and feel for the entire application or parts of it [37]. The JUCE user elements define an abstract base class with their look and feel methods, which are then implemented by the `juce::LookAndFeel` base class. Over time the default look and feel of JUCE widgets changed and with every major change a new `juce::LookAndFeel_Vx` class inheriting the previous one was created. For the MCLSR software the `MCLSRLookAndFeel` class (requirement **R.11**) subclasses `juce::LookAndFeel_V4` and defines the applications colour scheme and font. Figure C.7 shows the class diagram of the look and feel class and all virtual methods that have been overridden to create the applications design.

5.6.2 Grid and FlexBox Layout

The layouting of a component is done in its `resized()` method. There are a number of ways to create layouts but for this application mostly the `juce::Grid` and `juce::FlexBox` classes (implementing the CSS Grid and FlexBox layout specification [38,39]) were used. These classes provide containers for layouts using declarative rules [40], which are easy to use and allow responsive layouts for the application. Grids are used for fixed column and row layouts, whereas FlexBoxes are used for more flexible layouts. Listing 5.5 shows an example usage of the `juce::Grid` class in the `resized()` method of the `HeaderComponent`. The `juce::Grid` object is instantiated inside local scope and the layout of rows and columns is defined. This can be done like in this case with fractional integers (`1_fr` etc.), which is just a short form of defining a fraction of the whole width or height of the grid. Afterwards the grid items are defined in the desired order and with additional informations such as the alignment or justification of the item inside the grid. With the call to `performLayout()` the grid is layed out within the given rectangle.

```
auto r = getLocalBounds().reduced(Margin, 0);

Grid grid;
using Track = Grid::TrackInfo;

grid.templateRows = { Track(1_fr) };
grid.templateColumns = { Track(6_fr),
                         Track(10_fr),
                         Track(12_fr),
                         Track(10_fr),
                         Track(6_fr) };

grid.items =
{
    GridItem(_mclsLogoLabel),
    GridItem(_transportComponent),
    GridItem(_clock),
    GridItem(_projectControlsComponent),
    GridItem(_settingsButton).withJustifySelf(GridItem::JustifySelf::end)
};

grid.performLayout(r);
```

Listing 5.5: Example usage of the `juce::Grid` class.

5.6.3 Icons and Icon Helper

The icons used in the application are also part of Googles Material Design. They are embedded into an open-source icon font [41]. To allow easy access to the various icons, the `Icons.h` file includes an `Icons` namespace, in which the icon names are defined with their respective UTF-8 codes. The `IconHelper` class provides some useful methods to get, transform and draw icons. The icon can then be drawn within a graphics context by setting the contexts font to the icon font and then using the `drawFittedText()` method of the `juce::Graphics` class with the specified icon name as the text parameter.

6 User Manual

This chapter aims to give a high-level overview of the functions of the software and how to use them from a users point of view. The description follows the use case of a patient with no prior recordings getting her/his first examination.

Upon application startup the user is greeted with the screen shown in Figure 6.1. The header bar at the top of the window will be explained in detail later, because at this stage most of the functionality is disabled without a loaded examination. The middle of the screen shows input fields for creating a new examination or loading an existing one. The text input fields are underlined in red as long as their content is empty or incorrect (e.g. names containing numbers or the subject code containing special characters) and the button to create the examination is disabled until all fields have been filled. It is important to enter all data correctly as this cannot be changed after the examination is created. Figure 6.2 shows the input fields with wrong and correct values respectively.

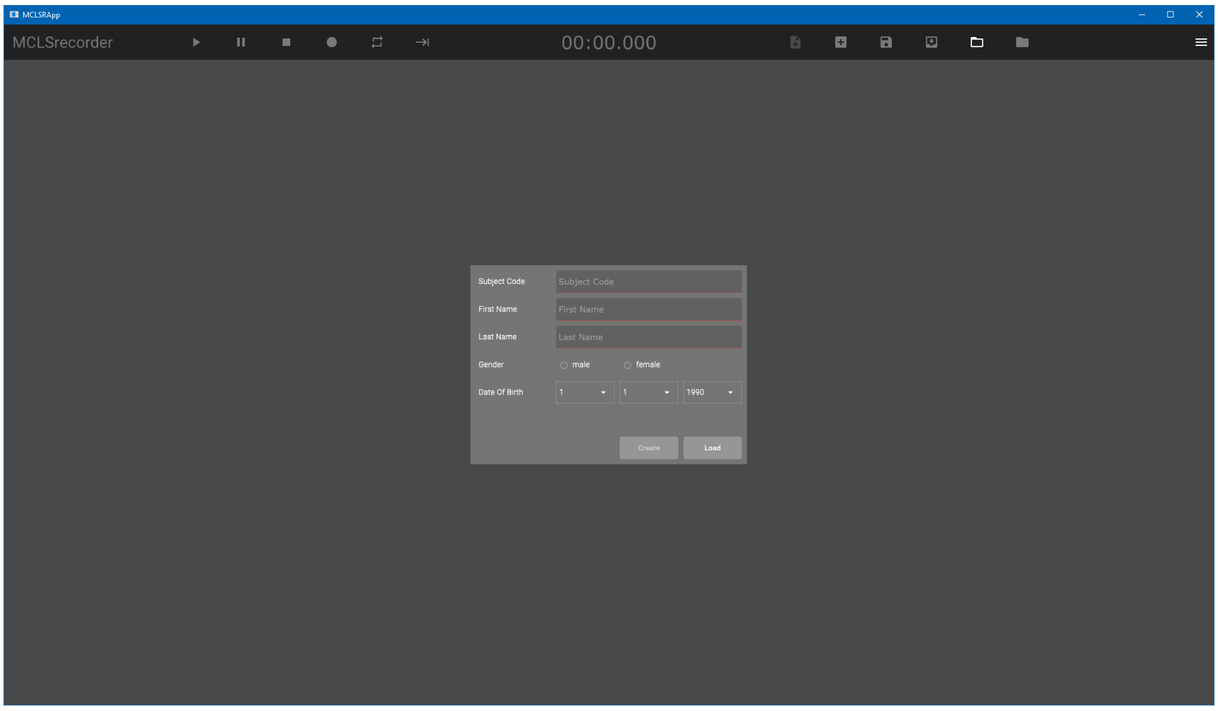
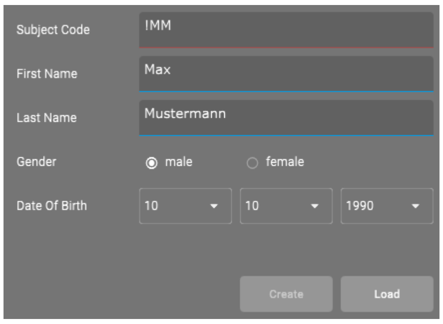


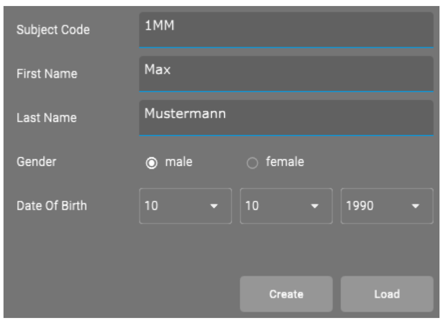
Figure 6.1: The application's startscreen.

Clicking on the load button opens a file dialog window, where the user can choose previously saved sessions (`*.mclsproject`-files) to load. After clicking on the button to create a new examination, the user is asked to select a containing folder for the examination. In this step, the software checks, if there already exists a subject with the same code and name inside this folder. If all values match, a new examination with a consecutive examination number is created inside the subjects subfolder. If the subject code already exists, the user is warned and has to change the code in order for it to be unique. If the values do not exist, a new subject folder with a subfolder for the first examination is created inside the selected folder.



Subject Code: IMM
First Name: Max
Last Name: Mustermann
Gender: (● male) ○ female
Date Of Birth: 10 / 10 / 1990

(a) Validation error in the subject code input field.



Subject Code: 1MM
First Name: Max
Last Name: Mustermann
Gender: (● male) ○ female
Date Of Birth: 10 / 10 / 1990

(b) Correct inputs.

Figure 6.2: Subject meta data input.

After creating the examination, the user is presented with the screen shown in Figure 6.3. The largest part of the screen is occupied by the time-based track view of the 16 channels. On the top left the visualization of the sensors with the same layout as the auscultation pad can be seen. The input meter and frequency analyzer are located underneath the sensor view. The header bar at the top remains, but is now active. The timestamp of the current playhead position is displayed in the middle of the bar. The buttons for controlling playback and recording are on the left side of the bar, with their functions from left to right:

- *Play:* Start the playback from the current position or stop it at the current position if the software is already playing. Playback can also be started and stopped using the *Space* key on the keyboard.
- *Pause:* Pause the playback at the current position.
- *Stop:* Stop the playback at the current position if audio is playing, reset the current position to the beginning if playback is inactive.
- *Record:* Start recording from the current position or stop recording if the software was recording. This button is disabled if there already exists a recording, because only one recording per examination is allowed.
- *Loop:* Loops around the current selection if active.
- *Follow:* If active, the track view will follow the playhead so the current position is always in the middle of the screen.

The buttons for project related functions are on the right side of the bar, from left to right:

- *New subject:* This button reveals the input fields for creating a new subject similar to the start screen in Figure 6.1. A second click switches back to the currently loaded examination.
- *New examination:* This button is used to create a new examination for the currently loaded subject. A subfolder with a consecutively numbered examination is created. The keyboard shortcut for this feature is *Ctrl+N*.
- *Save examination:* Save the current examination. This button is only enabled if there have been changes to the examination. The keyboard shortcut for saving is *Ctrl+S*.
- *Export audio:* This button is used to export all audio files with the currently selected filter settings applied and a downsampling to $f_s = 16kHz$ afterwards. This is best done once a suitable filter setting is found. The recordings only need to be exported again, if the filter

settings changed after the last export. Depending on the length of the recording this can take a while and the software can not be used during the operation. A progress bar with the current progress is displayed over the window.

- *Load examination:* Opens a dialog to choose an examination to load. Examination files have the fileending `*.mclsproject` and are located inside the examination subfolders.
- *Open containing folder:* Reveals the containing folder of the active examination in the operating systems file manager for quick access.

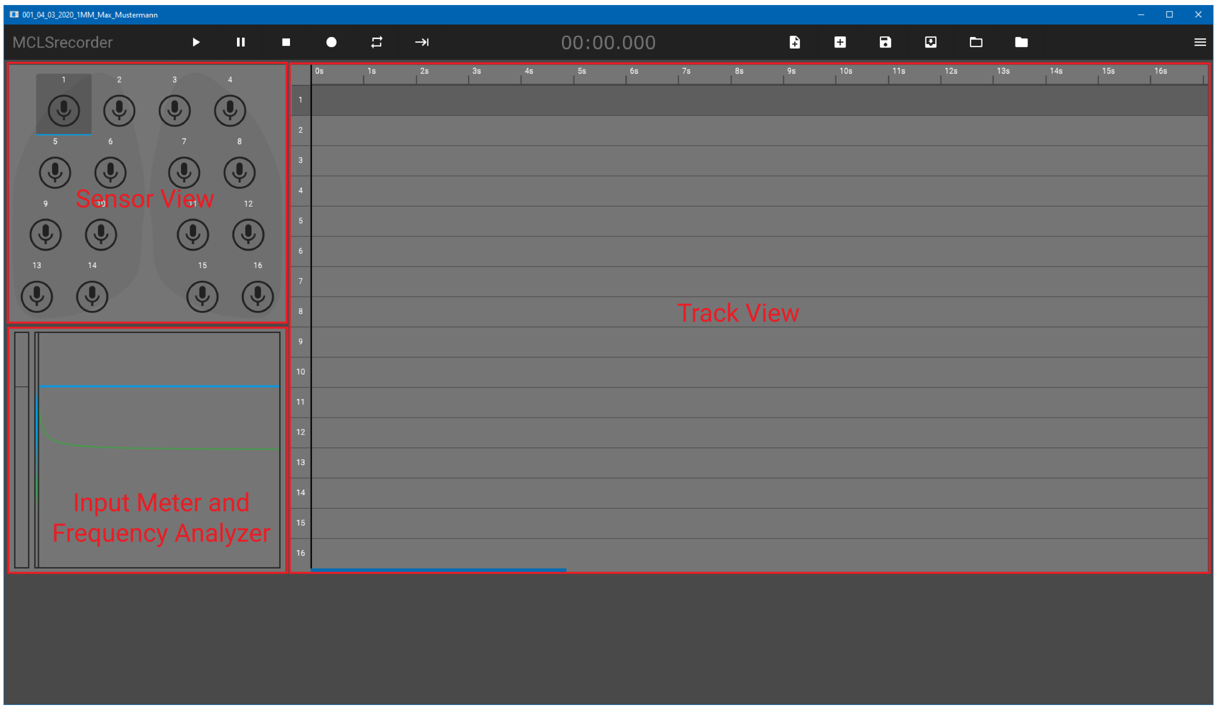


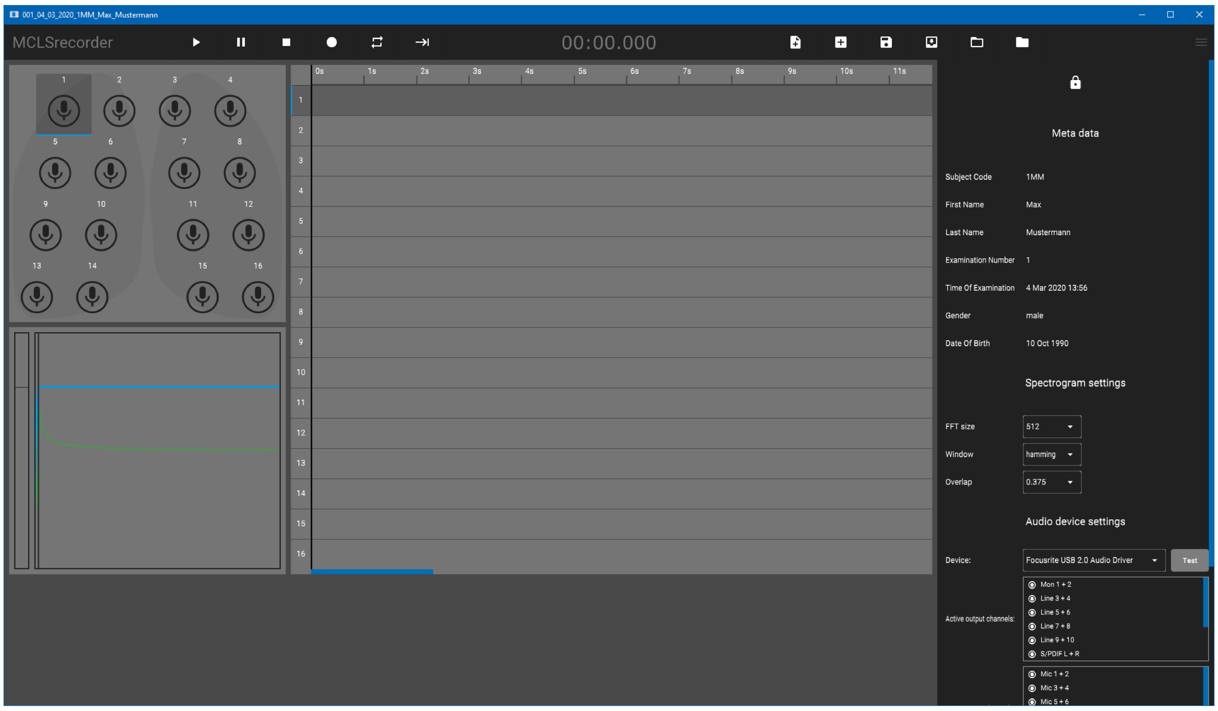
Figure 6.3: The initial screen of a new project.

The rightmost button is used to expand or collapse the settings panel as shown in Figure 6.4. The button at the top of the settings panel is used to enable/disable the expert mode, which reveals additional functionality as can be seen in Figure 6.5. This includes controls to set the input gain and cutoff frequency of the high-pass filter and a button to apply the current settings to all tracks. Additionally there are buttons for soloing and muting of tracks and a track settings button to change the input ordering. These functions were mainly used during development for debugging purposes and should be left as it is.

The next segment of the settings panel displays the subject meta data of the currently loaded examination. This includes the subject code, the subjects name, the examination number, the time of the examination and the subjects gender and date of birth.

Underneath the meta data the settings for the spectrogram can be found. This includes settings for the size of the FFT, the window to use and the overlapping factor. Changing one of these settings will cause a recalculation of the spectrogram.

The last segment of the settings can be used to select audio input and output devices (depending on the audio driver and platform there are separate options for input and output (Mac OS X with CoreAudio and Linux with ALSA) or one combined option (Windows with ASIO)) and configure the active channels as well as the sampling rate and blocksize of the audio callback. Depending on the operating system and how many audio devices are detected on the computer, this could be set to the wrong device by default. To make sure the correct device is used, the audio device settings can be checked and changed to the correct device before creating or loading an examination.



Meta data

Subject Code	1MM
First Name	Max
Last Name	Mustermann
Examination Number	1
Time Of Examination	4 Mar 2020 13:56
Gender	male
Date Of Birth	10 Oct 1990

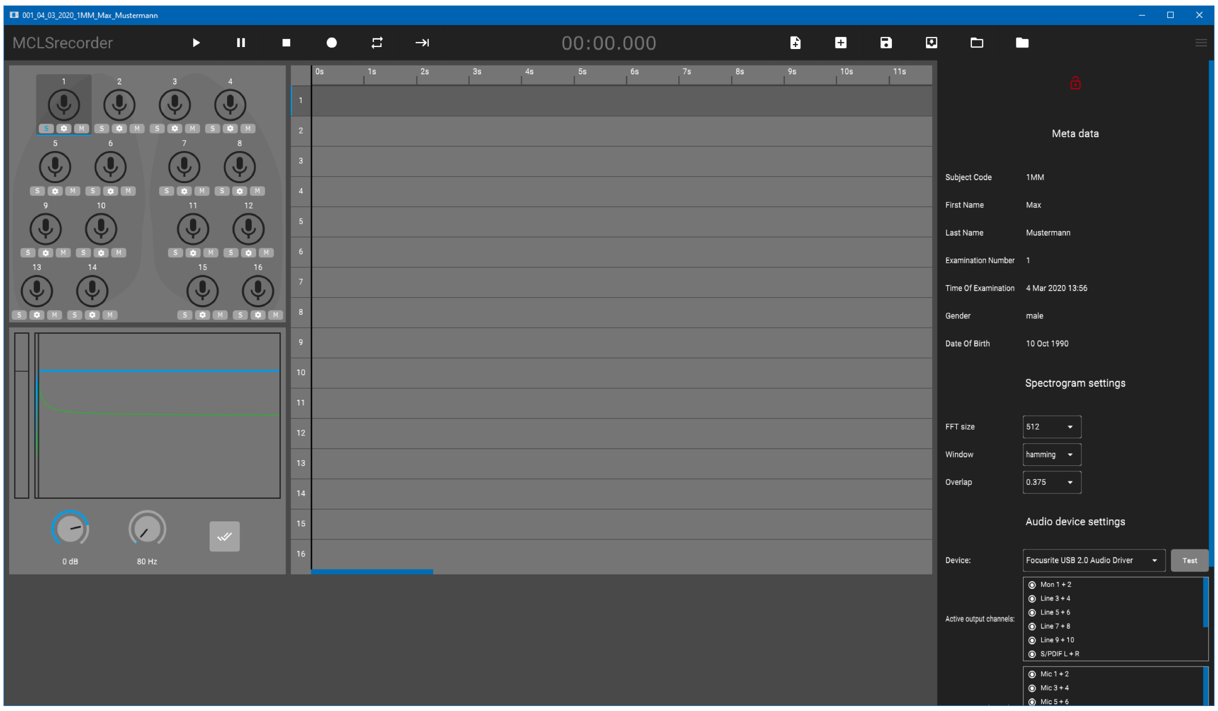
Spectrogram settings

FFT size	512
Window	hamming
Overlap	0.375

Audio device settings

Device:	Focusrite USB 2.0 Audio Driver
Active output channels:	Mon 1 + 2, Line 3 + 4, Line 5 + 6, Line 7 + 8, Line 9 + 10, S/PDIF L + R
	Mic 1 + 2, Mic 3 + 4, Mic 5 + 6

Figure 6.4: The initial screen of a new project with expanded settings panel.



Meta data

Subject Code	1MM
First Name	Max
Last Name	Mustermann
Examination Number	1
Time Of Examination	4 Mar 2020 13:56
Gender	male
Date Of Birth	10 Oct 1990

Spectrogram settings

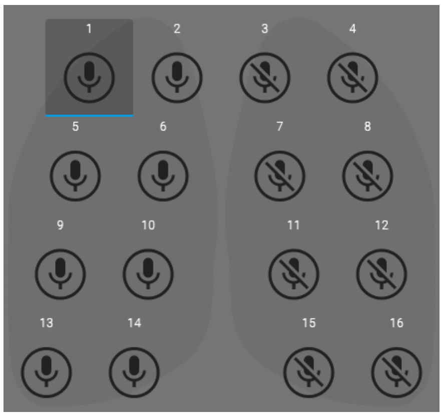
FFT size	512
Window	hamming
Overlap	0.375

Audio device settings

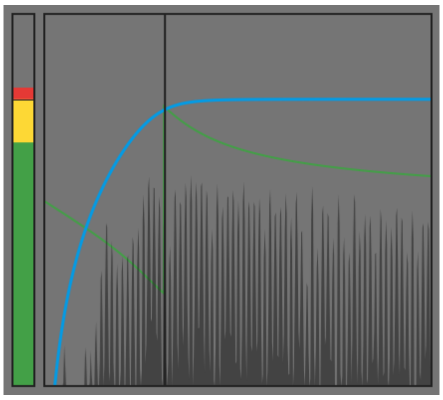
Device:	Focusrite USB 2.0 Audio Driver
Active output channels:	Mon 1 + 2, Line 3 + 4, Line 5 + 6, Line 7 + 8, Line 9 + 10, S/PDIF L + R
	Mic 1 + 2, Mic 3 + 4, Mic 5 + 6

Figure 6.5: Additional functionality with expert mode enabled.

On the left side of the window the user has access to the sensor visualization, shown in Figure 6.6a, as well as some input meters displaying the level and spectrum of the current input signal or played back audio as can be seen in Figure 6.6b. The sensor visualization mimics the sensor placement on the hardware. Individual sensors can be enabled or disabled for recording by clicking on the microphone symbols. Clicking on a sensor number selects the sensor and the corresponding track on the right side. The bar on the left side of the `TrackToolsComponent` displays the current input level. The black horizontal line represents $0dB$, the bar is displayed in green up to $-12dB$, in yellow from $-12dB < x < 0dB$ and in red above $0dB$. The spectrum of the signal is displayed in real-time on the right side, along with the high-pass filters frequency (blue) and phase (green) response. The vertical black line indicates the filters cutoff frequency. When hovering the mouse over the level meter or the spectrum a tooltip with additional information pops up. On the level meter the tooltip displays the current input level in dB in real-time, on the spectrum the mouse positions corresponding frequency, level and phase angle are displayed.



(a) The `MicrophoneComponent`.



(b) The `TrackToolsComponent`.

Figure 6.6: The sensor and input signal visualizations.

Clicking the record button starts a recording from the current position (the beginning by default). The recordings are visualized by red clips (spanning from the start of the recording up to the current recording position) on each track with enabled microphone, as shown in Figure 6.7. The current position is indicated by the black vertical line, the playhead. After clicking the record button again or the stop button, the recording is stopped. The clip's colour changes to blue as shown in Figure 6.8 and the examination, including the raw audio data, is automatically saved. Every track corresponds to a sensor in the sensor view on the left side. Clicking on the sensor number selects the track and the corresponding sensor. Above the tracks a timeline is used to display a time grid in seconds. Clicking anywhere inside the timeline causes the playhead to jump to that position. Additionally the playhead can be dragged to a desired position.

By clicking on one of the recorded clips or by clicking and dragging the mouse over a part of the clip the detail view will be opened up on the bottom of the window, shown in Figure 6.9. This view consists of a larger display of the selected area including a timeline and a playhead with the same functionality as in the track view, as well as the corresponding spectrogram of the selected part. The waveform can be zoomed in and out with the mouse wheel or by clicking left and right respectively. The whole detail view can be resized by clicking and dragging up or down between detail view and track view, as shown in Figure 6.10.

Once the recording was finished successfully and the cutoff frequency of the high-pass filter was set as needed, the recordings can be exported via the export button. This applies the filter settings and downsamples the audio data. While rendering the audio files, a progress bar is displayed as an overlay, as can be seen in Figure 6.11. A new examination for the same subject can now be created by simply clicking the corresponding button in the header bar. This will create the new examination and increase the examination number of this patient.

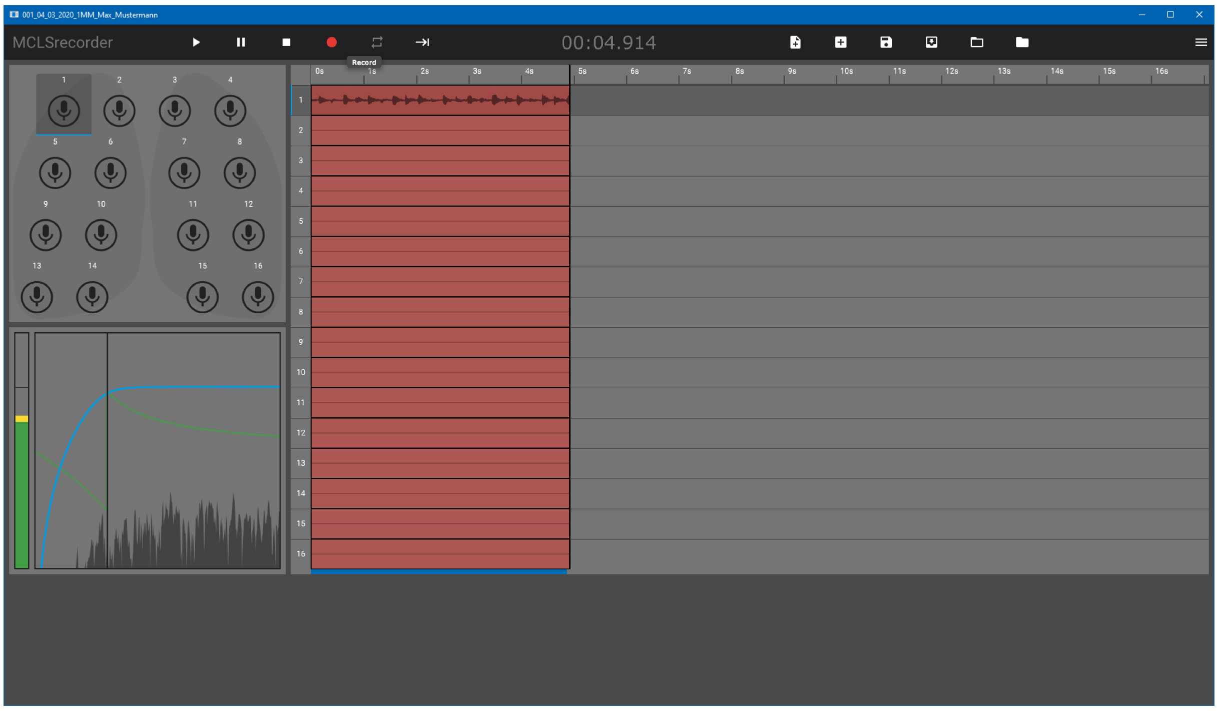


Figure 6.7: Recording the subjects lung sounds.

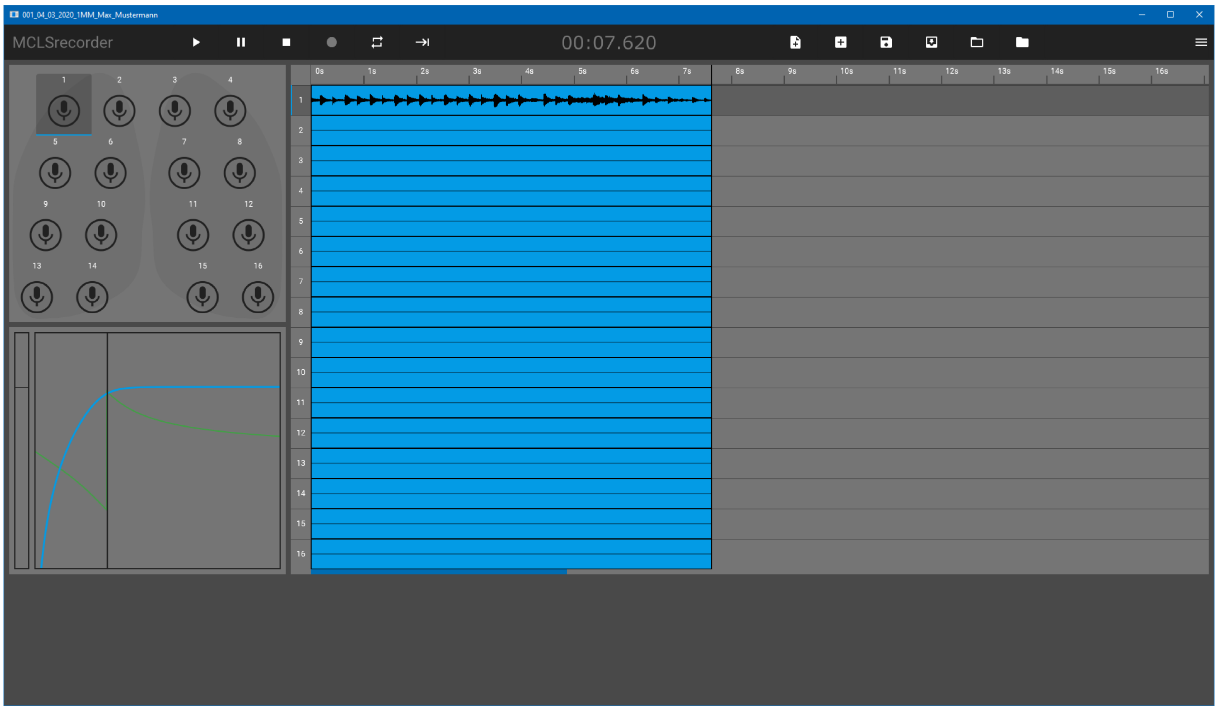


Figure 6.8: Finished recording session.

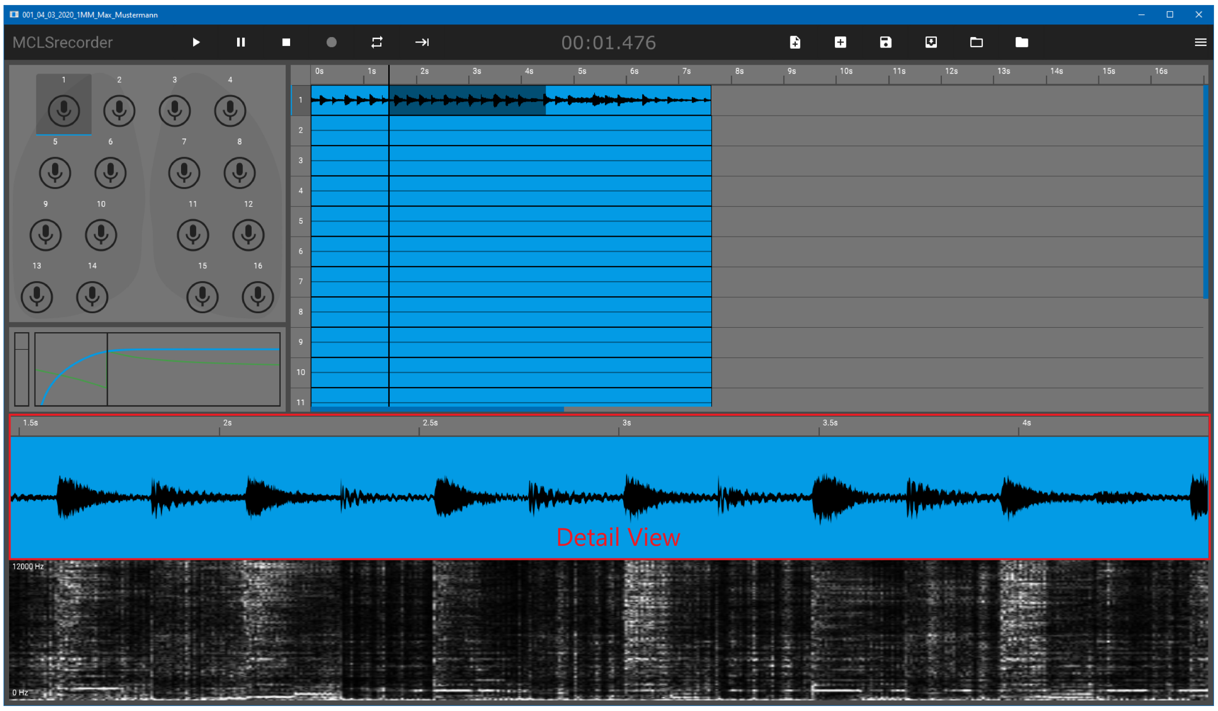


Figure 6.9: Detail view and spectrogram.

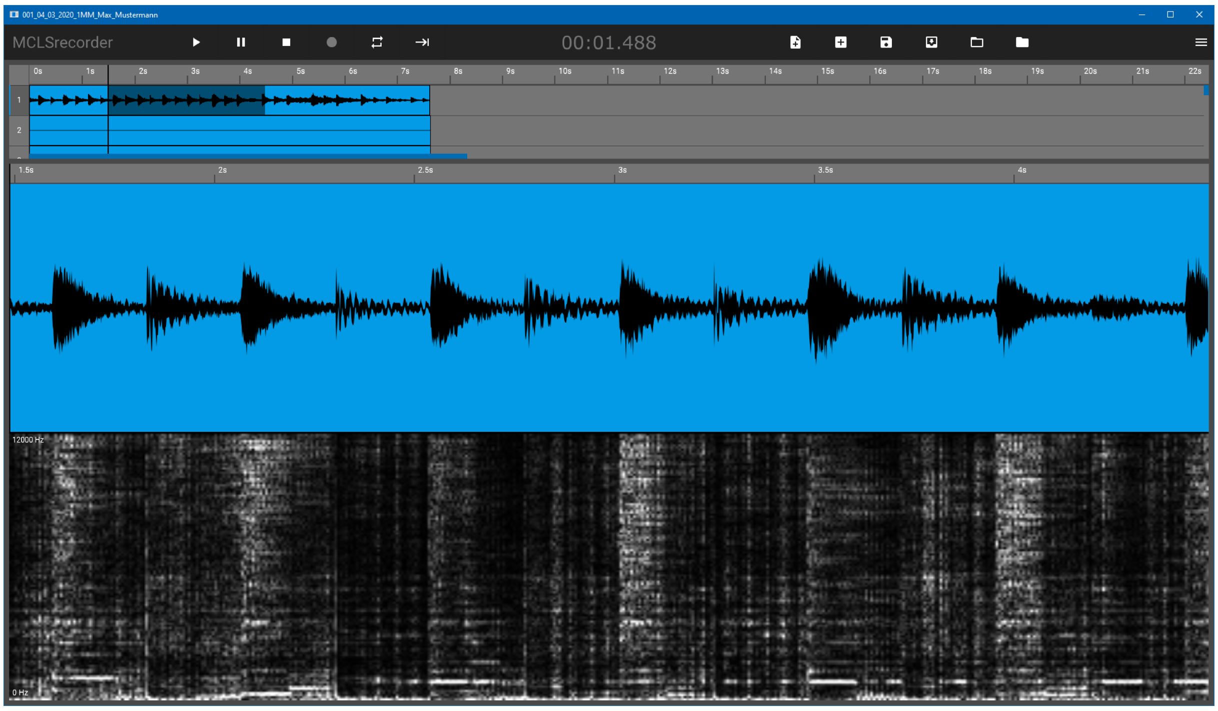


Figure 6.10: Resized detail view and spectrogram.

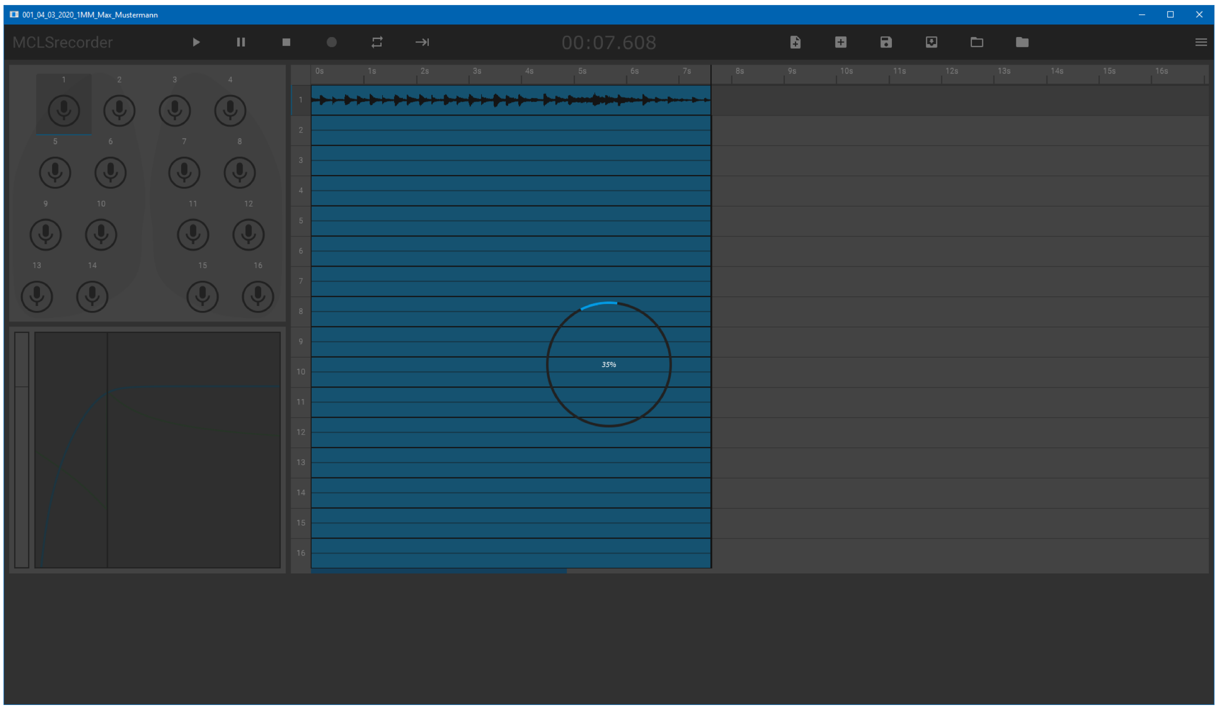


Figure 6.11: Progress bar during audio file export.

7

Hardware

This chapter describes the hardware for which the Multi-Channel Lung Sound Recording Software was developed. The hardware was developed alongside the software at the SPSC Laboratory but not as part of this thesis. Although the software can be used with any audio interface with the correct drivers, it was particularly developed for this hardware. The hardware is based on the Multi-Channel Lung Sound Recording Device developed by Elmar Messner for his PhD-Thesis [1] shown in Figure 7.1. The main motivation for further development of the existing hardware was to increase usability and portability. Ideally all of the needed hardware is housed inside the foam pad and only one USB cable is used to connect and power the audio interface inside. To fulfill these requirements a 16-Channel audio interface with a small form factor and the capability to draw its power solely from the USB connection is needed. This also entails that the microphones that are used need to have a small form factor and a low power consumption. The choice fell onto the miniDSP MCHStreamer Multi-Channel Multi-Protocol USB audio interface [42] in conjunction with Infineon IM69D120 MEMS microphones [43]. The following sections are an outline of the parts and their specifications.

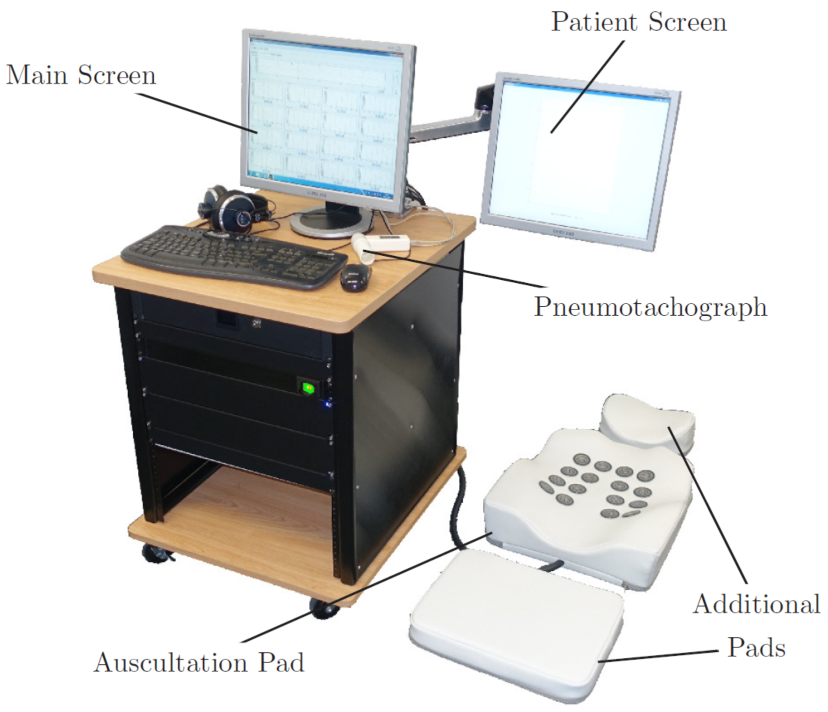


Figure 7.1: Multi-Channel Lung Sound Recording Device. Source: [1].

7.1 Audio Interface

Originally the idea was to use the 16-channel USB 2.0 soundcard with MEMS microphones developed by Andreas Wöhrer during his masters thesis [44]. Due to some technical problems with this board the miniDSP MCHStreamer (Figure 7.2), which also uses the XMOS XCore 200 chipset, was chosen. With dimensions of $13 \times 40 \times 62mm$ ($H \times W \times D$) it is small enough to fit inside the auscultation pad.

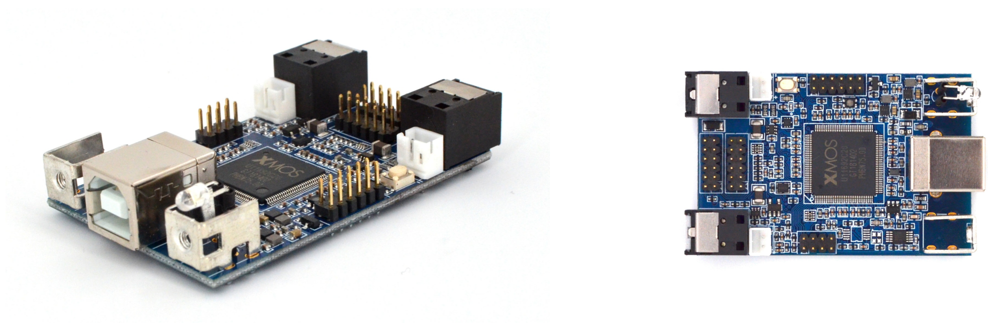


Figure 7.2: The miniDSP MCHStreamer Multi-Channel USB audio interface. Source: [45].

As described in the MCHStreamer manual [42] it includes a number of features like a pair of optical ports for TOSLINK or ADAT input and output, a pair of headers for SPDIF input as well as headers for logic-level input and output data formats such as I2S, TDM, PDM and DSD. The different modes can be used by loading the corresponding firmware provided by the manufacturer. The audio interface can be connected and powered via a single USB Type A to USB Type B cable. In order to use the board with 16 input channels, the PDM firmware was chosen. This firmware supports sampling rates of $f_s = 8kHz$, $11.025kHz$, $12kHz$, $16kHz$, $32kHz$, $44.1kHz$ and $48kHz$. The PDM input data is received on J3 supplying two channels per physical data line. The channels are mirrored in I2S and sent to the computer over USB. There is a single PDM clock which is duplicated onto two output lines. The data for the left channel of each PDM line is taken on the falling edge and the data for the right channel is taken on the rising edge of the PDM clock as shown in Figure 7.3. The computer receives input data always in the form of a 24-bit word. The audio interface acts as a USB Audio class 2.0 device, which is supported by Mac OS X (CoreAudio) and Linux (ALSA) natively without the need for a dedicated driver. For Windows an ASIO driver is provided, including a control panel to configure sampling rate, clock, buffer size and volumes.

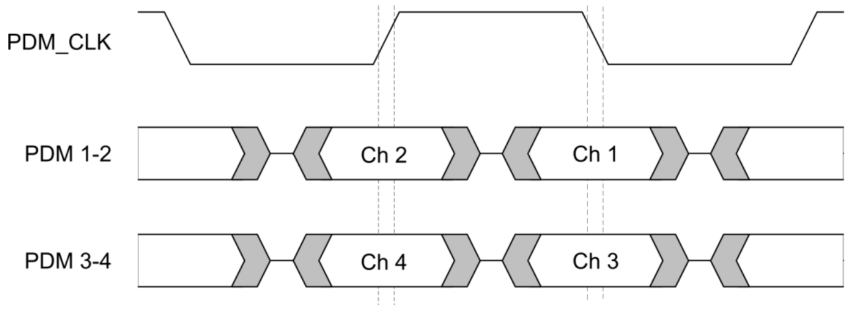


Figure 7.3: PDM timing. Source: [42].

7.2 MEMS Microphones and Stethoscope Heads

Because of their small form factor, the integrated ADC and the fact that they provide a PDM signal the digital MEMS microphones IM69D120 by Infineon were chosen as microphones for the recording hardware. According to the datasheet [43] the IM69D120 was designed for applications that require a high SNR, a wide dynamic range and low distortions. It features a SNR of $69dB(A)$, a dynamic range of $95dB$ and a sensitivity of $-26dBFS$. The microphone's Application Specific Integrated Circuit (ASIC) contains a low-noise preamplifier and a high-performance sigma-delta ADC (latency at $1kHz$: $\tau = 6\mu s$) providing the PDM output [43]. The microphone package dimensions are $4 \times 3 \times 1.2mm$ and it is soldered onto a small PCB providing the connections to the audio interface. This PCB is mounted at the rear of the newly designed stethoscope head (see Figure 7.5) such that the microphone lines up with the hole drilled through the head. For pressure equalization with the surrounding a venting hole ($d = 0.4mm$) is drilled perpendicular to the rotation axis of the stethoscope head. The fully assembled stethoscope head can be seen in Figure 7.4, including a protection cap with a tension relief for the microphone cable.

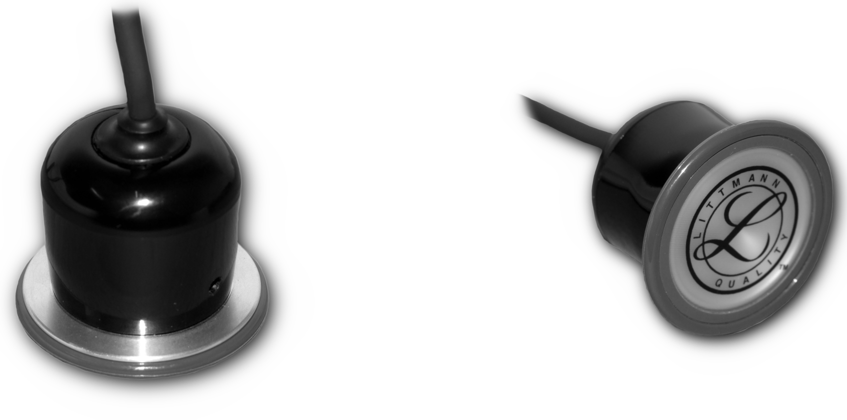


Figure 7.4: Fully assembled stethoscope head.

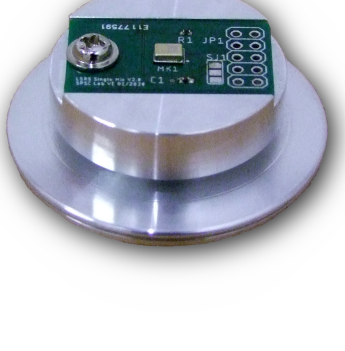


Figure 7.5: Stethoscope head with MEMS PCB mounted at the back.

7.3 Auscultation Pad

The auscultation pad (see Figure 7.6) follows the same sensor layout as in [1]. The bottom of the pad consists of a rigid foam pad which houses the audio interface inside an aluminium enclosure and the cables connecting the stethoscope heads with the interface. On top of the rigid foam pad a softer foam is used that adapts to the subject's physique. The foams are covered by artificial leather. Instead of the heavy multi-core microphone cable only one USB cable comes out of the pad to connect to the computer.

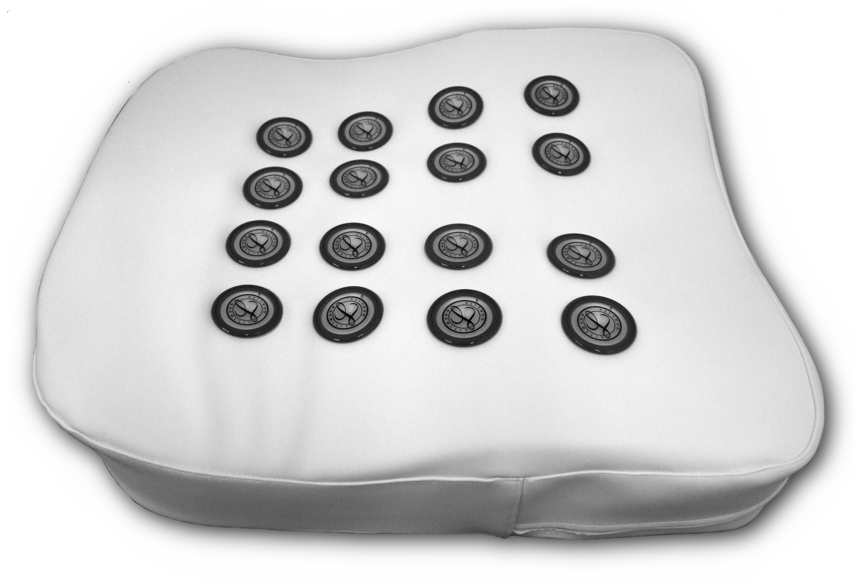


Figure 7.6: Auscultation pad.

8 Multi-Channel Lung Sound Classification with Keras

Keras [46] is one of the most popular and fastest growing deep learning frameworks [47]. It is a high-level deep learning API, which can run with TensorFlow, CNTK or Theano as backends. The idea behind Keras is to allow for easy and fast prototyping, to support both convolutional and recurrent neural networks and to run seamlessly on CPU and GPU. The first public version of Keras was published on March 28, 2015 by François Chollet [48].

In this chapter Keras was used to implement and evaluate the proposed models from [1] and examine if the performance can be met or even increased. Additionally the dataset is used to evaluate the performance of a Keras implementation of a DenseNet [49], which is used as a front-end to the bidirectional gated recurrent neural network [1].

8.1 Multi-Channel Lung Sound Classification

In [1], Messner et al. present an approach to multi-channel lung sound classification with convolutional recurrent neural networks. The aim is to classify breathing cycles as either normal or abnormal from patients with idiopathic pulmonary fibrosis (IPF). Together with their recently developed 16-channel lung sound recording device they conducted a clinical trial with lung healthy subjects and patients with IPF. Their proposed convolutional recurrent neural network achieves a F-Score of $F_1 \approx 92\%$. The processing framework consists of a 16-channel recording of a full breathing cycle with a sampling frequency of $f_s = 16kHz$, which is zero-padded according to the longest recording in each set. Each recording is then processed with a STFT to extract spectral information of the lung sounds. This results in 257-bin log magnitude spectrograms and leads to a $[4 \times 4 \times 257]$-dimensional[1] or if stacked 4112-dimensional feature vector for each frame. The features are then processed by the neural network and a frame-wise classification is performed. Details of the models, such as layers, loss, etc. can be found in [1]. The outputs from all frames are summed up for each of the three classes (*healthy*, *pathological* and *no signal*) and the maximum value of the sums determines the final class, ignoring the *no signal* class.

8.1.1 Multilayer Perceptron Model

The Multilayer Perceptron model (MLP) is used as a baseline model. MLPs are also called feedforward neural networks (FNN) and are the simplest type of artificial neural networks. In an MLP the information flows forward through the fully connected layers without feedback connection. It consists of an input layer, followed by hidden layers and an output layer, as can be seen in Figure 8.1. For a frame-wise processing with frame index $f \in \{1, \ldots, F\}$ and the

[1] $4 \times 4 \hat{=} 16$-channels, corresponds to the grid of microphones from the multi-channel lung sound recording device, 257 corresponds to the bins of the log magnitude spectrogram.

index of the hidden layers $l \in \{1, \ldots, L-1\}$ the MLP can be described mathematically as [1]:

$$\mathbf{h}_f^l = g\left(\mathbf{W}_x^l \mathbf{x}_f^l + \mathbf{b}_h^l\right) \tag{8.1}$$

$$\mathbf{y}_f = m\left(\mathbf{W}_y \mathbf{h}_f^{L-1} + \mathbf{b}_y\right). \tag{8.2}$$

The hidden states, which are used as inputs for the next hidden layer, are computed by applying a non-linear activation function $g(\cdot)$ to the sum of the bias term $\mathbf{b}_h^l$ and the dot product of the input weight matrix $\mathbf{W}_x^l$ and the input vector $\mathbf{x}_f^l$ [1]. The output $\mathbf{y}_f$ is computed by applying a non-linear function $m(\cdot)$ to the sum of the output bias and the dot product of the output weights and the hidden states of the last hidden layer [1].

The best performing MLP in this case consists of an input layer, which receives the stacked 4112-dimensional feature vector of each timestep, followed by one hidden layer with 300 neurons and an output layer with softmax activation and three outputs.

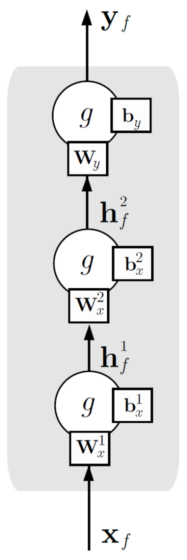


Figure 8.1: Flow graph of an MLP with two hidden layers. Source: [1].

8.1.2 Bidirectional Gated Recurrent Neural Network

More suitable for processing sequential input data are recurrent neural networks (RNN) [50]. There is a plethora of different architectures of RNNs, including vanilla RNNs, gated recurrent neural networks (GRNN) and long short term memory networks (LSTM), of which the last two are more suited to model longer temporal dependencies [1]. A GRNN is chosen here, because it is computationally more efficient than a LSTM model but achieves comparable performance. In a bidirectional model the sequence is processed in the forward layer from the first to the last frame and additionally in the backward layer from the last to the first frame, as shown in Figure 8.2. Both hidden state sequences are stacked and fed to the next hidden layer and from the last hidden layer to the output layer with softmax activation and three outputs. The advantage of bidirectional RNNs over a vanilla RNNs is that they not only use past information (forward layer) but also future information (backward layer) [1].

The best model consists of an input layer which receives sequences of 4112-dimensional feature vectors, followed by 4 bidirectional hidden layers and the output layer.

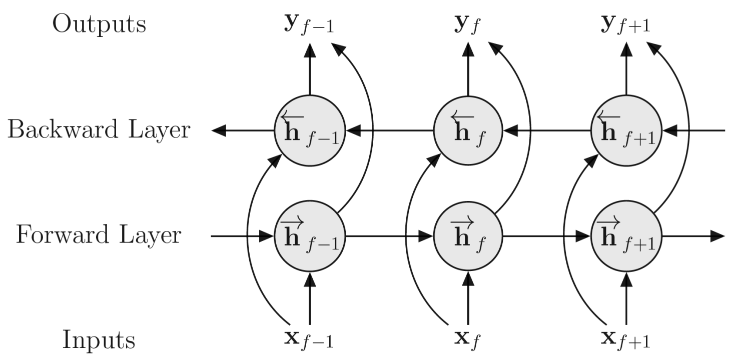

Outputs
Backward Layer
Forward Layer
Inputs
$\mathbf{y}_{f-1}$
$\mathbf{y}_{f}$
$\mathbf{y}_{f+1}$
$\overleftarrow{\mathbf{h}}_{f-1}$
$\overleftarrow{\mathbf{h}}_{f}$
$\overleftarrow{\mathbf{h}}_{f+1}$
$\overrightarrow{\mathbf{h}}_{f-1}$
$\overrightarrow{\mathbf{h}}_{f}$
$\overrightarrow{\mathbf{h}}_{f+1}$
$\mathbf{x}_{f-1}$
$\mathbf{x}_{f}$
$\mathbf{x}_{f+1}$

Figure 8.2: Bidirectional recurrent neural network. Source: [1].

8.1.3 Convolutional Bidirectional Gated Recurrent Neural Network

Convolutional neural networks (CNNs) are feedforward neural networks used primarily for grid-like data such as images (2-D grid of pixels) or time-series (1-D grid of samples at regular time intervals) [50]. CNNs are comprised of convolutional layers, subsampling (pooling) layers and fully connected layers. The concept of a convolutional layer and a subsampling layer is shown in Figure 8.3. Each convolutional layer consists of multiple feature maps K and performs an image convolution of the input layer with a $m \times m$ kernel (filter), which results in a feature map of size $(N-m+1) \times (N-m+1)$, if a stride of one is used [1]. After the convolutional layer, a subsampling layer (size $n \times n$) can be applied, which uses a summary statistic (maximum, average, L^2-norm) to replace the outputs in the vicinity of a certain location and reduce the size to ${}^{(N-m+1)}/_{n} \times {}^{(N-m+1)}/_{n}$ [1]. Mathematically the operation to obtain the feature map $k \in \{1,\ldots,K\}$ in layer l is described as [1]:

$$h_{ij}^{kl} = g\left(\mathbf{W}^{kl} * \mathbf{X}_{ij}^{l} + b_k^l\right). \tag{8.3}$$

The output h_{ij}^{kl} is computed by applying a non-linear activation function to the sum of the bias term b_k^l and the convolution of a size $m \times m$ section of the input image $\mathbf{X}^l$ at position $i, j \in \{1,\ldots,N-m+1\}$ and the filter $\mathbf{W}^{kl}$.

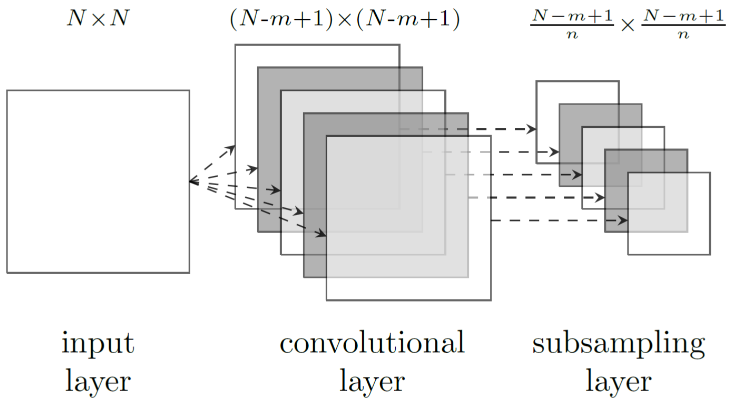

$N \times N$
$(N\text{-}m{+}1) \times (N\text{-}m{+}1)$
$\frac{N-m+1}{n} \times \frac{N-m+1}{n}$
input layer
convolutional layer
subsampling layer

Figure 8.3: Illustration of a convolutional layer (with $K = 5$ feature maps) and a pooling layer. Source: [1].

In [1], Messner et al. choose a CNN as a front-end to the BiGRNN model. One image is the 4×4-microphone grid with a depth of 257 channels. The first convolutional layer uses a kernel size of 1×1 to reduce the channels to 30 feature maps. The second convolutional layer uses a kernel size of 3×3 and a stride and padding of 1 and generates 30 feature maps as well. Every convolutional layer needs to be applied to every timestep individually. The output of the

CNN is then flattened and fed to the BiGRNN model as sequences of features. The complete processing framework, including the convolutional front-end and the fully connected recurrent layers, is shown in Figure 8.4.

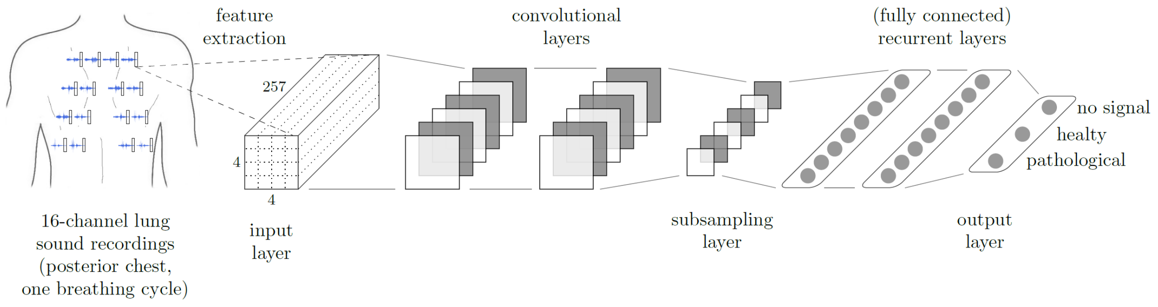


Figure 8.4: Frame-wise multi-channel lung sound processing framework with a recurrent neural network and a convolutional neural network front-end. Source: [1].

8.1.4 The Dataset

The dataset consists of recordings of 387 breathing cycles (252 healthy and 135 IPF) conducted from 23 subjects (16 healthy and 7 IPF). The sounds were recorded over the posterior chest at different airflow rates. For each subject two 16-channel recordings with shallow and deep breathing over several breathing cycles were recorded. Each breathing cycle is labeled frame-wise as either *healthy*, *pathological* and *no signal*. Further details are provided in [1].

8.1.5 Expected Results

The implementation in [1] uses the precision (P_+), Sensitivity (Se) and the F-Score (F_1) as evaluation metrics. The precision score indicates how many of the samples labeled as pathological are actually true. The sensitivity score provides information on how many of the pathological samples are actually labeled as pathological. The F-Score is the weighted average of these two scores and is used as a metric for the overall performance due to the uneven class distribution. Messner uses a 7-fold cross-validation with the recordings of each IPF patient appearing once in the test set. Because of the small size of the dataset and the class imbalance the scores are micro-averaged. The obtained scores in [1] are shown in Table 8.1.

Model	P_+(%)	Se(%)	F_1(%)
MLP	75.0	37.8	50.2
BiGRNN	93.1	80.0	86.1
ConvBiGRNN	100.0	85.9	92.4

Table 8.1: Evaluation of the three models.

8.2 Introduction to Keras

8.2.1 Sequential and Functional API

The main data structure of Keras is a *model*. There are two ways to develop a model with Keras. The sequential and the functional model API. The sequential model is a linear stack of layers.

The model is defined first, then the layers are added to it sequentially. Only models with a linear architecture can be built with the sequential API [51]. For more complex and non-linear models Keras offers the functional API. With the functional API every layer is defined on its own and an input tensor is assigned to it as argument. After all the layers and connections are defined, the model is instantiated and only needs references to the input and output layers. The connections in between are already defined and can be arbitrary. This allows for example for multi-input and output models or shared layers [52].

8.2.2 Training

When the model is defined, it needs to be compiled in both APIs in order to configure the learning process. In the compilation step the loss function, optimizer and the metrics to display during training are defined. Training a model is done via a call of the `fit` function. The function takes the training inputs and labels as numpy arrays, as well as the validation inputs and labels. It also takes the number of epochs to train, the batch size and a list of callbacks (see section 8.2.5). It returns a dictionary containing the history of training, that is the loss, accuracy, validation loss and validation accuracy for every epoch [51].

8.2.3 Prediction

After finishing the training process, the `predict` function is used to generate predictions on the test data [53]. The obtained activations can then be used to evaluate the performance of the model.

8.2.4 Layers

The main building blocks for neural networks are layers. Keras offers a range of predefined layers and wrappers to use to design the neural network. Additionally own layers can be written, which inherit from Keras' `Layer` class. In this project, the focus was to test the performance of models that consist entirely of Keras layers. All layers have methods in common, that allow to get and set the weights of a layer and to obtain a dictionary of the configuration of the layer, which can be used to reinstantiate the layer [54].

8.2.5 Callbacks

Callbacks in Keras are functions that are called during training at the end of every epoch [55]. This is for example used to save the model after every epoch, to apply early stopping, to log the losses and accuracies to TensorBoard for visualization and to schedule the learning rate across epochs.

8.3 Implementation of Deep Neural Networks in Keras

8.3.1 Project Structure

The project structure follows an object-oriented programming paradigm and is based on [56]. It is built upon base classes for data access, models, training, evaluating, visualizing and object creation. The `DataProvider` class provides functions to obtain the data from file and reshape it. The `Model` class represents a neural network model and provides functionality to save and load models to and from file and to build a model for training. Every model has its own class derived from the base class and implements the function to build the model. The `Trainer` class takes in a model instance and provides the functionality to train it. The `Evaluator` class is used to evaluate the performance of the previously trained model. The `Visualizer` class provides functions for the visualization of data and training history. To keep object creation manageable and in one place the `Factory` class provides functions to instantiate objects of each of the previously mentioned classes.

8.3.2 Configuration

In order to configure the different models and training scenarios, the most important parameters are stored in human-readable configuration files (either as .yaml-file or .json-file).

```yaml
model:
        name: "convbigrnn"
        input_shape: !!python/tuple [!!null , 4, 4, 257]
        merge_mode: "concat"
        feature_maps: 30
        optimizer: "adam"
        learning_rate: 0.00001
        epochs_drop: 10.0
        momentum: 0.8
        dropout_rate: 0.5
        hidden_units: 200
        output_units: 3

data:
        number_of_folds: 7
        data_path: "data"

trainer:
        history_path: "model_history"
        epochs: 400
        batch_size: 32

callbacks:
        checkpoint_directory: "model_checkpoints"
        tensorboard_directory: "logs"
        patience: 50

logging:
        logging_filepath: "convbigrnn.log"
```

Listing 8.1: Configuration file for the ConvBiGRNN model.

Listing 8.1 shows the configuration of the ConvBiGRNN model. This provides a quick way of changing parameters and keeping track of the used values.

8.3.3 The `MLPModel` Class

The `MLPModel` class is inherited from the `ModelBase` class. As this is the most basic of the three models, the Sequential API is used for the implementation. Listing 8.2 shows the code to build the baseline MLP model, discussed in Section 8.1.1. The model consists of one `Dense` layer, followed by a `Dropout` layer and the dense output layer. All layers are wrapped in `TimeDistributed` layers, to duplicate the model across all frames of the sequence. The first layer needs to define the input shape of the stacked 4112-dimensional feature vectors. The weights are initialized with orthogonal matrices as suggested in [1].

```python
def build_model(self):
    self.model = Sequential()
    init = Orthogonal(gain=1.0, seed=42)

    self.model.add(TimeDistributed(Dense(units=self.hidden_units,
                                         activation='relu',
                                         kernel_initializer=init),
                                   input_shape=self.input_shape,
                                   name='MLP_Dense_1'))
    self.model.add(TimeDistributed(Dropout(self.dropout_rate),
                                   name='MLP_Dropout_1'))
    self.model.add(TimeDistributed(Dense(units=self.output_units,
                                         activation='softmax',
                                         kernel_initializer=init),
                                   name='MLP_Output_Layer'))

    opt = self.get_optimizer()

    self.model.compile(loss='categorical_crossentropy',
                       optimizer=opt,
                       metrics=['accuracy'])
```

Listing 8.2: Function to build the MLP model with the sequential API.

8.3.4 The `BiGRNNModel` Class

The `BiGRNNModel` class inherits from the `ModelBase` class. This model is more complex and uses the functional API. Listing 8.3 shows code for the BiGRNN described in Section 8.1.2. For the gated recurrent units `CuDNNGRU` layers are used. These layers are optimized for use with Nvidia-GPUs and speed up the training time significantly. The GRUs are wrapped in `Bidirectional` layers, which are used for recurrent neural networks and create a forward- and a backward-pass on the timeaxis of the recurrent layer supplied to it. Between the recurrent layers are again `Dropout` layers, which are wrapped in `TimeDistributed` layers. The layers use the same orthogonal initializer as the `MLP` model.

```
def get_bigrnn_layers(self, inputs, initializer):
    bigrnn = Bidirectional(CuDNNGRU(units=self.hidden_units,
                                    return_sequences=True,
                                    kernel_initializer=initializer),
                           merge_mode=self.merge_mode,
                           name='BiGRNN_1')(inputs)
    bigrnn = TimeDistributed(Dropout(self.dropout_rate),
                             name='BiGRNN_Dropout_1')(bigrnn)
    bigrnn = Bidirectional(CuDNNGRU(units=self.hidden_units,
                                    return_sequences=True,
                                    kernel_initializer=initializer),
                           merge_mode=self.merge_mode,
                           name='BiGRNN_2')(bigrnn)
    bigrnn = TimeDistributed(Dropout(self.dropout_rate),
                             name='BiGRNN_Dropout_2')(bigrnn)
    bigrnn = Bidirectional(CuDNNGRU(units=self.hidden_units,
                                    return_sequences=True,
                                    kernel_initializer=initializer),
                           merge_mode=self.merge_mode,
                           name='BiGRNN_3')(bigrnn)
    bigrnn = TimeDistributed(Dropout(self.dropout_rate),
                             name='BiGRNN_Dropout_3')(bigrnn)
    bigrnn = Bidirectional(CuDNNGRU(units=self.hidden_units,
                                    return_sequences=True,
                                    kernel_initializer=initializer),
                           merge_mode=self.merge_mode,
                           name='BiGRNN_4')(bigrnn)
    bigrnn = TimeDistributed(Dropout(self.dropout_rate),
                             name='BiGRNN_Dropout_4')(bigrnn)
    return bigrnn
```

Listing 8.3: Function to get the BiGRNN layers with the functional API.

8.3.5 The `ConvBiGRNNModel` Class

The `ConvBiGRNNModel` class inherits from the `BiGRNNModel` class in order to reuse the bidirectional recurrent layers from Listing 8.3. The convolutional frontend of this model consists of two `Conv2D` layers with kernel sizes of 1×1 and 3×3 respectively, both with stride and padding of 1. The `Conv2D` layer is a 2D convolutional layer, e.g. for spatial convolution over images. In this case an "image" corresponds to the 4×4 microphone grid with a depth of 257 channels. The convolutional layers need to be wrapped in `TimeDistributed` layers in order to process the sequences. To connect the convolutional frontend to the bidirectional recurrent network, the output of the second convolutional layer needs to be flattened by a `Flatten` layer. This stacks the $[4 \times 4 \times 30]$-dimensional features to a [480]-dimensional feature vector. All layers are again initialized with the orthogonal initializer. The `ConvBiGRNNModel` is shown in Listing 8.4.

8.3.6 Evaluation

The `Evaluator` class is used for the post processing of predictions and evaluation of the performance of the models. The models output layer has three outputs for every timestep (no signal, healthy, pathological). In order to get one class per sample, the predicted values are summed over all timesteps and the 'no signal'-class is ignored. The processed predictions and test data labels from all folds are concatenated in lists and the micro averaged scores are calculated.

```python
def build_model(self):
    self.inputs = Input(shape=self.input_shape, name='ConvBiGRNN_Input_Layer')
    init = Orthogonal(gain=1.0, seed=42)

    conv_bigrnn = TimeDistributed(Conv2D(filters=self.feature_maps,
                                         kernel_size=(1, 1),
                                         kernel_initializer=init,
                                         strides=1,
                                         padding='same',
                                         activation='relu'),
                                  name='ConvBiGRNN_Conv2D_1')(self.inputs)
    conv_bigrnn = TimeDistributed(Conv2D(filters=self.feature_maps,
                                         kernel_size=(3, 3),
                                         kernel_initializer=init,
                                         strides=1,
                                         padding='same',
                                         activation='relu'),
                                  name='ConvBiGRNN_Conv2D_2')(conv_bigrnn)
    conv_bigrnn = TimeDistributed(Flatten(),
                                  name='ConvBiGRNN_Reshape_Layer')(conv_bigrnn)

    conv_bigrnn = self.get_bigrnn_layers(conv_bigrnn, init)
    self.output = TimeDistributed(Dense(units=self.output_units,
                                        activation='softmax',
                                        kernel_initializer=init),
                                  name='ConvBiGRNN_Output_Layer')(conv_bigrnn)

    self.model = Model(inputs=self.inputs, outputs=self.output)

    opt = self.get_optimizer()

    self.model.compile(loss='categorical_crossentropy',
                       optimizer=opt,
                       metrics=['accuracy'])
```

Listing 8.4: Function to get the BiGRNN layers with the functional API.

8.4 Introduction to DenseNet

The main idea behind Dense Convolutional Networks (DenseNets) is, that convolutional neural networks can be significantly deeper, more accurate, and more efficient to train, if layers close to the input and close to the output have shorter connections [49]. Standard convolutional networks with L layers have one connection between each layer and its following layer (L connections in total). In DenseNets, each layer uses the feature maps from all previous layers as inputs and its own feature maps are the inputs to all succeeding layers, which leads to $L(L+1)/2$ connections [49]. The advantages of DenseNets are a mitigation of the vanishing gradient problem, they encourage feature reuse, they propagate features more strongly and reduce the number of parameters [49].

8.5 DenseNet Implementation in Keras

In general a DenseNet consists of dense blocks and transition blocks. The dense blocks consist of a number of so called composite functions $H_l(\cdot)$. This function consists of three consecutive operations: a batch normalization, a rectified linear unit and a 3×3 convolution. The composite function receives the concatenation of all preceding feature maps as input: $x_l = H_l([x_0, x_1, \dots, x_{l-1}])$. The transition blocks are needed because the concatenation is not feasible when the size of feature maps changes [49]. They consist of batch normalization, a rectified linear unit and a 1×1 convolution. The 2×2 average pooling layer proposed in [49] is not used here. The growth rate k is the number of feature maps produced by H_l. Due to the fact that each layer has access to the previous feature maps, the DenseNet can have very narrow layers, e.g. $k = 12$ [49]. To reduce the number of input feature maps and improve computational efficiency the bottleneck layer introduces a 1×1 convolution before each 3×3 convolution [49]. The bottleneck layer produces $4k$ feature maps. Additionally a compression factor θ with $0 < \theta \leq 1$ can be used in transition blocks to further reduce the number of feature maps. When $\theta = 1$, the number of feature maps stays unchanged [49].

The implementation of DenseNet is based on [57]. The `DenseNetBiGRNNModel` class inherits from the `BiGRNNModel` class and reuses the bidirectional recurrent network. This model essentially replaces the convolutional frontend from the `ConvBiGRNNModel` class with a DenseNet frontend. The `create_dense_net` method creates this DenseNet depending on the parameters given. The `depth` parameter defines the number of dense blocks, the `layers_per_block` determines how many composite functions to use per block. All layers are again initialized with orthogonal weights and wrapped in `TimeDistributed` layers. In order to connect to the recurrent network the output of the DenseNet needs to be flattened by a `Flatten` layer. The `build_model` method of the `DenseNetBiGRNNModel` is shown in Listing 8.5.

```
def build_model(self):
    self.inputs = Input(shape=self.input_shape)
    init = Orthogonal(gain=1.0, seed=42)

    densenet_bigrnn = self.create_dense_net(
        classes=self.output_units,
        input=self.inputs,
        include_output_layer=False,
        initializer=init,
        depth=self.depth,
        filters=self.feature_maps,
        layers_per_block=self.layers_per_block,
        dropout_rate=self.dropout_rate,
        bottleneck=self.bottleneck,
        reduction=self.reduction,
        weight_decay=self.weight_decay,
        use_bias=self.use_bias)
    densenet_bigrnn = TimeDistributed(Flatten())(densenet_bigrnn)

    densenet_bigrnn = self.get_bigrnn_layers(densenet_bigrnn, init)
    self.output = TimeDistributed(Dense(units=self.output_units,
          activation='softmax'))(densenet_bigrnn)

    self.model = Model(inputs=self.inputs, outputs=self.output)

    opt = self.get_optimizer()

    self.model.compile(loss='categorical_crossentropy',
                       optimizer=opt,
                       metrics=['accuracy'])
```

Listing 8.5: Function to build the DenseNetBiGRNN model with the functional API.

8.6 Results

Table 8.2 shows the results for the Keras implementation of the different models. The scores are the micro-average scores of the seven folds. Figures 8.5-8.8 show the corresponding losses over epochs for the different models. Compared to the results in Table 8.1 the performance of the different models deviates from the expected results.

Model	$P_+(\%)$	$Se(\%)$	$F_1(\%)$
MLP	100.0	48.1	65.0
BiGRNN	95.7	81.5	88.0
ConvBiGRNN	88.9	83.0	85.8
DenseNetBiGRNN	78.2	50.4	61.3

Table 8.2: Evaluation of the four models.

MLP Model

The best performance for the MLP baseline model was achieved with a learning rate of 10^{-5} and a batch size of 32. The rest of the parameters was taken from [1]. This implementation slightly outperforms the implementation from [1]. The precision score of 100% is not really significant because often the network predicted all subjects as healthy. The training and validation loss is shown in Figure 8.5.

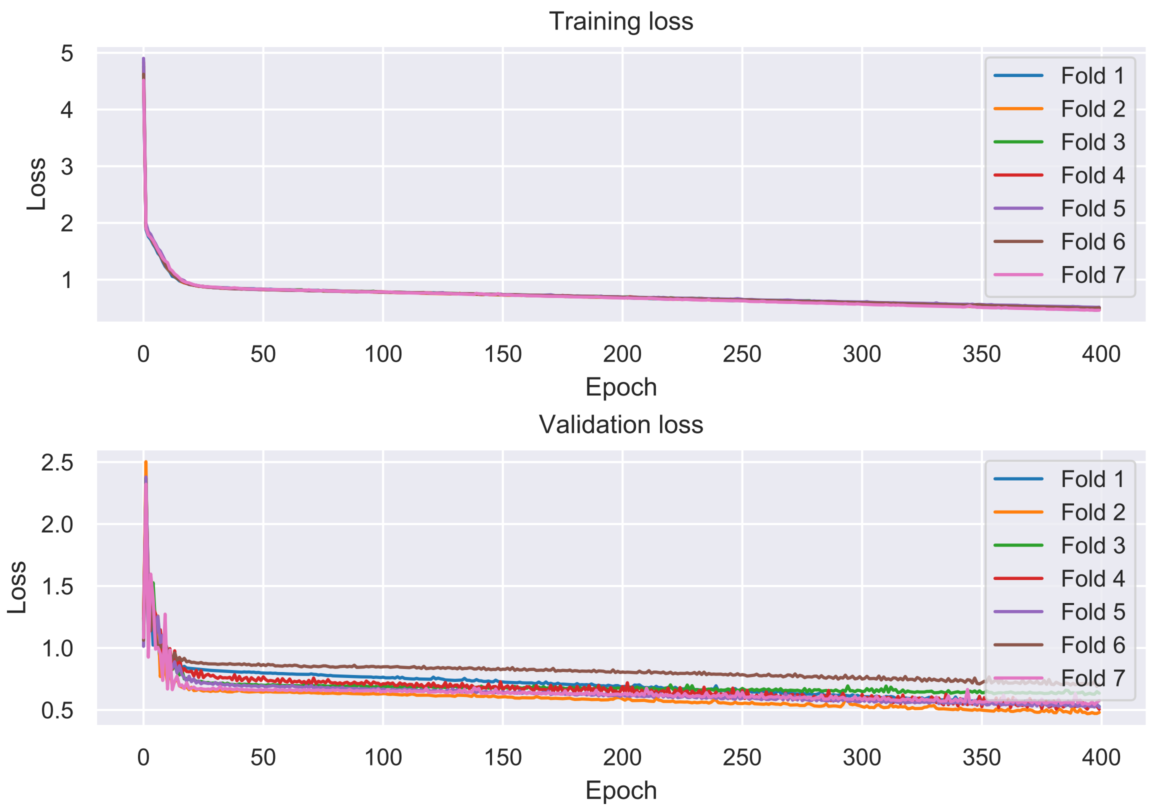


Figure 8.5: Training and validation loss of the MLP model.

BiGRNN Model

The best BiGRNN model uses a learning rate of 10^{-6} and a batch size of 32. The forward and backward paths of the bidirectional recurrent layers are concatenated. Other parameters are taken from [1]. The training process is shown in Figure 8.6. This model also slightly outperforms the model from [1]. The small differences in performance could be resulting from internal implementation differences of the Keras layers compared to the implementation in [1]. Also this model uses the Cuda optimized layers, which internally uses a hyperbolic tangent activation function instead of the rectified linear unit.

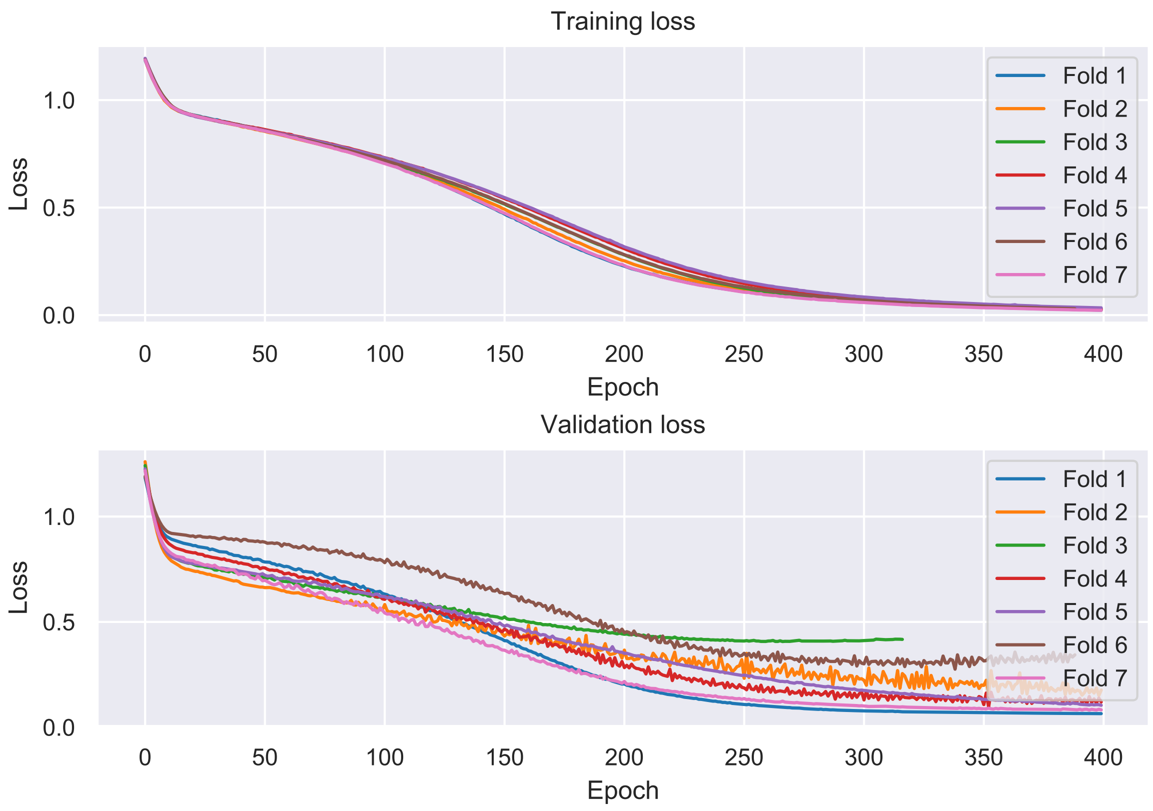


Figure 8.6: Training and validation loss of the BiGRNN model.

ConvBiGRNN Model

The ConvBiGRNN uses a learning rate of 10^{-5} and a batch size of 32. The bidirectional recurrent part is the same as for the BiGRNN model, the other parameters are taken from [1]. Conversely this model performs significantly worse than the model in [1], and even worse than the BiGRNN model. The training process is shown in Figure 8.7. The validation losses of fold 3 and fold 6 are increasing again very early on. Although the models with the lowest validation loss for each fold are used for the evaluation, the performance could not be met. Experiments with different activations, optimizers and learning rates could also not improve the performance. The performance gap could be connected to implementation errors and is most likley tied to the `Conv2D` layers, as the rest of the model is identical to the BiGRNN model.

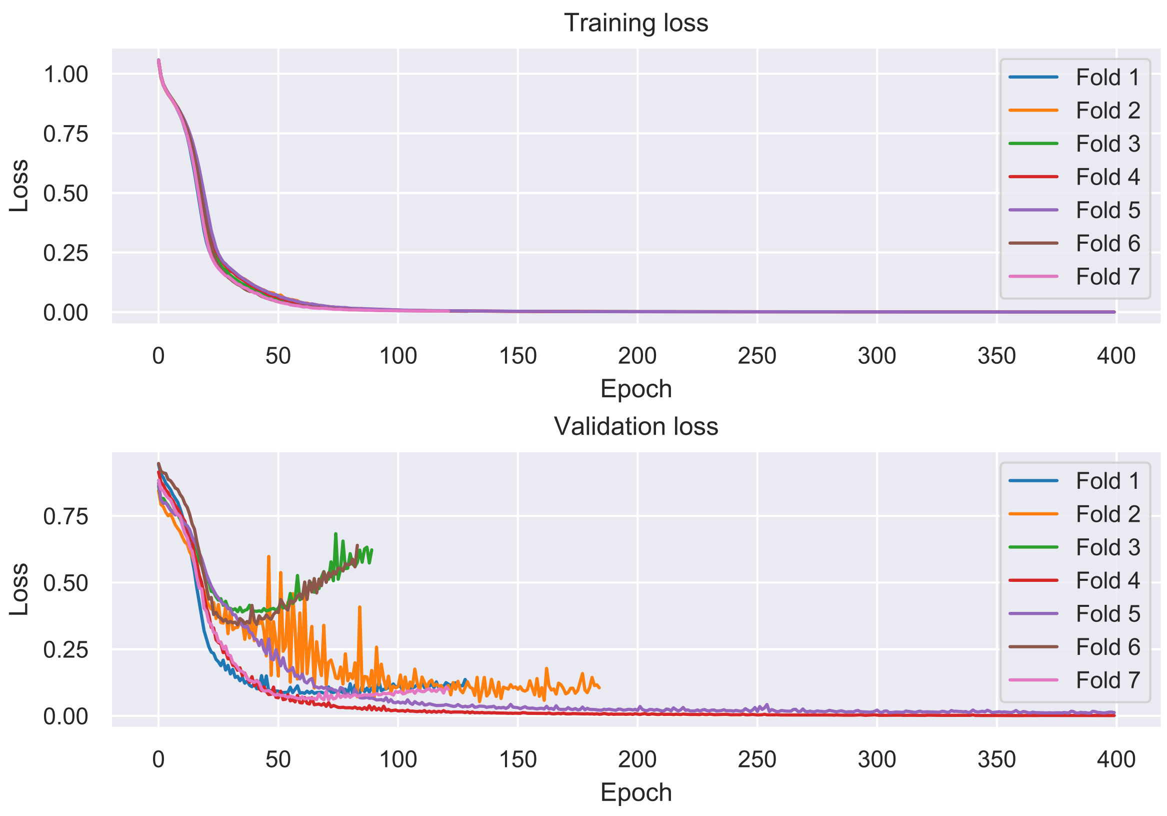


Figure 8.7: Training and validation loss of the ConvBiGRNN model.

DenseNetBiGRNN Model

The DenseNetBiGRNN model uses a learning rate of 10^{-5} and a batch size of 32. The DenseNet frontend consists of three dense blocks with three layered composite functions in each dense block. The dense blocks are connected via transition blocks. The initial number of feature maps is 30, the growth rate is $k = 12$ and bottleneck layers are enabled. The training and validation losses are shown in 8.8. The performance of this model could not meet the performance of the other three models. This is most likely because the parameters were not obtained via a proper grid search and thus the optimal set of parameters was not found. However, it is questionable if a model architecture like this could achieve better results with the given data. The image size of the data is only 4×4, which does not leave much space for pooling operations and a really deep network like the DenseNet.

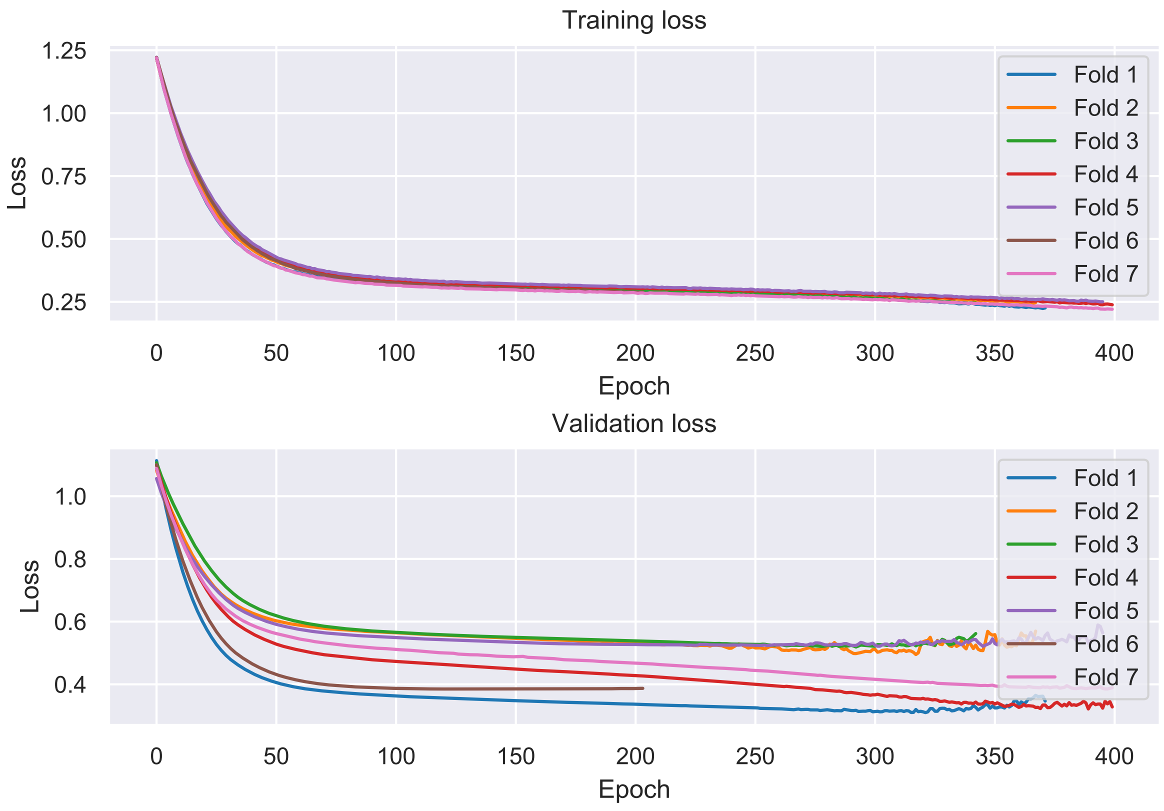


Figure 8.8: Training and validation loss of the DenseNetBiGRNN model.

9 Conclusion

For this thesis a modern audio software application was developed using the state-of-the-art C++ frameworks for cross-platform audio applications, JUCE and Tracktion Engine. The research for this thesis included the in depth study of those two frameworks, which encompassed a lot of source code analysis because of the Tracktion Engine's lack of documentation. Following the core architectural concepts of the JUCE framework, a module based structure was derived for the software. Each module has its own responsibilities and a robust system for inter-object communication based on the JUCE data structure `ValueTree` was implemented. The multithreading capabilities of JUCE were used to push calculations onto background threads wherever possible to keep the application responsive to the user. The JUCE DSP module was used extensively in various places for FFT calculations and filter design and implementations. The overall project structure and UML class diagrams for individual classes were presented. A brief overview of the newly developed hardware including the audio interface, the microphones and the auscultation pad was given. Furthermore, the deep neural network models for multi-channel lung sound classification developed in [1] were re-implemented in the machine learning framework Keras and the performance was compared to the original low-level implementations. Additionally, an attempt was made to use a dense convolutional neural network architecture for the lung sound classification.

A Abbreviations

ADAT	Alesis Digital Audio Tape
ALSA	Advanced Linux Sound Architecture
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIO	Audio Stream Input/Output
BiGRNN	Bidirectional Gated Recurrent Neural Network
CI	Continuous Integration
CNN	Convolutional Neural Network
ConvBiGRNN	Convolutional Bidirectional Gated Recurrent Neural Network
DAW	Digital Audio Workstation
DenseNet	Dense Convolutional Network
DOM	Document Object Model
DSD	Direct Stream Digital
DSP	Digital Signal Processing
DTFT	Discrete Time Fourier Transform
FFT	Fast Fourier-Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FNN	Feedforward Neural Network
GRNN	Gated Recurrent Neural Network
GRU	Gated Recurrent Unit
I2S	Inter-IC Sound
IC	Integrated Circuit
IPF	Idiopathic Pulmonary Fibrosis
JUCE	Jules' Utility Class Extensions
LSTM	Long Short-Term Memory
MCLSR	Multi Channel Lung Sound Recorder
MEMS	Micro-Electrical-Mechanical System
MLP	Multilayer Perceptron
MVC	Model-View-Controller
PCB	Printed Circuit Board
PDM	Pulse Density Modulation
RNN	Recurrent Neural Network
SNR	Signal to Noise Ratio
SPDIF	Sony/Philips Digital Interface
SPSC	Signal Processing and Speech Communications
STFT	Short-Time Fourier-Transform
TDM	Time Domain Multiplexing
te	`tracktion_engine`, namespace of the Tracktion Engine
TOSLINK	TOShiba LINK
UML	Unified Modeling Language

B
Folder and File Structure

```
mcls_core
├── mcls_core.cpp
├── mcls_core.h
├── BinaryData
│   ├── MCLSBinaryData.cpp
│   └── MCLSBinaryData.h
└── Core
    ├── ApplicationCommandHandler.h
    ├── ApplicationState.cpp
    ├── ApplicationState.h
    ├── CommandIDs.h
    ├── Constants.h
    ├── ConstrainedValue.h
    ├── Delegate.h
    ├── EditManager.cpp
    ├── EditManager.h
    ├── FlaggedAsyncUpdater.h
    ├── Identifiers.h
    ├── Progress.h
    ├── ProgressList.h
    ├── RenderingTaskRunner.h
    └── Utilities.h
```

Figure B.1: File structure of the core module

```
mcls_domain
├── mcls_domain.cpp
├── mcls_domain.h
└── Domain
    ├── FrequencyAnalyzer.h
    ├── HighPassFilterPlugin.cpp
    ├── HighPassFilterPlugin.h
    ├── SampleRateConversionScheduler.cpp
    ├── SampleRateConversionScheduler.h
    ├── SampleRateConverter.cpp
    ├── SampleRateConverter.h
    ├── Spectrogram.cpp
    ├── Spectrogram.h
    ├── SpectrogramScheduler.cpp
    ├── SpectrogramScheduler.h
    ├── TracksAndInputManager.cpp
    ├── TracksAndInputManager.h
    ├── TransportManager.cpp
    └── TransportManager.h
```

Figure B.2: File structure of the domain module

```
mcls_model
├── mcls_model.cpp
├── mcls_model.h
└── Model
    ├── Converters.cpp
    ├── Converters.h
    ├── MetaDataModel.cpp
    ├── MetaDataModel.h
    ├── MetaDataValidationService.cpp
    ├── MetaDataValidationService.h
    ├── ModelBase.h
    ├── ProjectModel.cpp
    ├── ProjectModel.h
    ├── SpectrogramSettingsModel.cpp
    ├── SpectrogramSettingsModel.h
    ├── ViewModel.cpp
    └── ViewModel.h
```

Figure B.3: File structure of the model module

```
mcls_application
├── mcls_application.cpp
├── mcls_application.h
├── Components
│   ├── Clips
│   │   ├── AudioClipComponent.cpp
│   │   ├── AudioClipComponent.h
│   │   ├── AudioClipComponentBase.cpp
│   │   ├── AudioClipComponentBase.h
│   │   ├── AudioClipDetailComponent.cpp
│   │   ├── AudioClipDetailComponent.h
│   │   ├── ClipComponent.cpp
│   │   ├── ClipComponent.h
│   │   ├── ClipComponentFactory.h
│   │   ├── RecordingClipComponent.cpp
│   │   ├── RecordingClipComponent.h
│   │   ├── SpectrogramComponent.cpp
│   │   └── SpectrogramComponent.h
│   ├── MetaData
│   │   ├── CoreMetaDataComponent.cpp
│   │   ├── CoreMetaDataComponent.h
│   │   ├── MetaDataComponent.cpp
│   │   ├── MetaDataComponent.h
│   │   ├── MetaDataComponentBase.cpp
│   │   └── MetaDataComponentBase.h
│   ├── Recorder
│   │   └── ⋮
│   ├── Settings
│   │   └── ⋮
│   ├── Tracks
│   │   └── ⋮
│   ├── Widgets
│   │   └── ⋮
│   └── ComponentBase.h
├── Infrastructure
│   └── ⋮
└── LookAndFeel
    ├── MCLSLookAndFeel.cpp
    └── MCLSLookAndFeel.h
```

Figure B.4: File structure of the application module

```
mcls_application
├── mcls_application.cpp
├── mcls_application.h
├── Components
│   ├── Clips
│   │   └── ⋮
│   ├── MetaData
│   │   └── ⋮
│   ├── Recorder
│   │   ├── HeaderComponent.cpp
│   │   ├── HeaderComponent.h
│   │   ├── MainComponent.cpp
│   │   ├── MainComponent.h
│   │   ├── MicrophoneComponent.cpp
│   │   ├── MicrophoneComponent.h
│   │   ├── PlayheadComponent.cpp
│   │   ├── PlayheadComponent.h
│   │   ├── ProjectControlsComponent.cpp
│   │   ├── ProjectControlsComponent.h
│   │   ├── RecorderComponent.cpp
│   │   ├── RecorderComponent.h
│   │   ├── ResizerComponent.cpp
│   │   ├── ResizerComponent.h
│   │   ├── TimelineComponent.cpp
│   │   ├── TimelineComponent.h
│   │   ├── TransportComponent.cpp
│   │   └── TransportComponent.h
│   ├── Settings
│   │   ├── DeviceSelectorComponent.cpp
│   │   ├── DeviceSelectorComponent.h
│   │   ├── SettingsComponent.cpp
│   │   ├── SettingsComponent.h
│   │   ├── SettingsContainerComponent.cpp
│   │   ├── SettingsContainerComponent.h
│   │   ├── SpectrogramSettingsComponent.cpp
│   │   └── SpectrogramSettingsComponent.h
│   ├── Tracks
│   │   └── ⋮
│   ├── Widgets
│   │   └── ⋮
│   └── ComponentBase.h
├── Infrastructure
│   └── ⋮
└── LookAndFeel
    ├── MCLSLookAndFeel.cpp
    └── MCLSLookAndFeel.h
```

Figure B.5: File structure of the application module

```
mcls_application
├── mcls_application.cpp
├── mcls_application.h
├── Components
│   ├── Clips
│   │   └── ⋮
│   ├── MetaData
│   │   └── ⋮
│   ├── Recorder
│   │   └── ⋮
│   ├── Settings
│   │   └── ⋮
│   ├── Tracks
│   │   ├── TrackComponent.cpp
│   │   ├── TrackComponent.h
│   │   ├── TrackComponentFactory.h
│   │   ├── TrackContainerComponent.cpp
│   │   ├── TrackContainerComponent.h
│   │   ├── TrackHeaderComponent.cpp
│   │   ├── TrackHeaderComponent.h
│   │   ├── TrackHeaderWithButtonsComponent.cpp
│   │   ├── TrackHeaderWithButtonsComponent.h
│   │   ├── TrackLabelComponent.cpp
│   │   ├── TrackLabelComponent.h
│   │   ├── TrackLabelsContainerComponent.cpp
│   │   ├── TrackLabelsContainerComponent.h
│   │   ├── TrackToolsComponent.cpp
│   │   ├── TrackToolsComponent.h
│   │   ├── TrackToolsContainerComponent.cpp
│   │   ├── TrackToolsContainerComponent.h
│   │   ├── TrackViewComponent.cpp
│   │   └── TrackViewComponent.h
│   ├── Widgets
│   │   └── ⋮
│   └── ComponentBase.h
├── Infrastructure
│   └── ⋮
└── LookAndFeel
    ├── MCLSLookAndFeel.cpp
    └── MCLSLookAndFeel.h
```

Figure B.6: File structure of the application module

```
mcls_application
├── mcls_application.cpp
├── mcls_application.h
├── Components
│   ├── Clips
│   │   └── ⋮
│   ├── MetaData
│   │   └── ⋮
│   ├── Recorder
│   │   └── ⋮
│   ├── Settings
│   │   └── ⋮
│   ├── Tracks
│   │   └── ⋮
│   ├── Widgets
│   │   ├── FrequencyResponseComponent.cpp
│   │   ├── FrequencyResponseComponent.h
│   │   ├── IconPushButton.cpp
│   │   ├── IconPushButton.h
│   │   ├── IconToggleButton.cpp
│   │   ├── IconToggleButton.h
│   │   ├── LevelMeterComponent.cpp
│   │   ├── LevelMeterComponent.h
│   │   ├── ProgressBarComponent.cpp
│   │   ├── ProgressBarComponent.h
│   │   ├── TransportClock.cpp
│   │   └── TransportClock.h
│   └── ComponentBase.h
├── Infrastructure
│   ├── IconHelper.cpp
│   ├── IconHelper.h
│   ├── Icons.h
│   ├── NotifyingViewport.h
│   ├── SelectionState.cpp
│   ├── SelectionState.h
│   ├── TrackState.cpp
│   ├── TrackState.h
│   ├── TrackToolsState.cpp
│   ├── TrackToolsState.h
│   └── UIBehaviour.h
└── LookAndFeel
    ├── MCLSLookAndFeel.cpp
    └── MCLSLookAndFeel.h
```

Figure B.7: File structure of the application module

C Class Diagrams

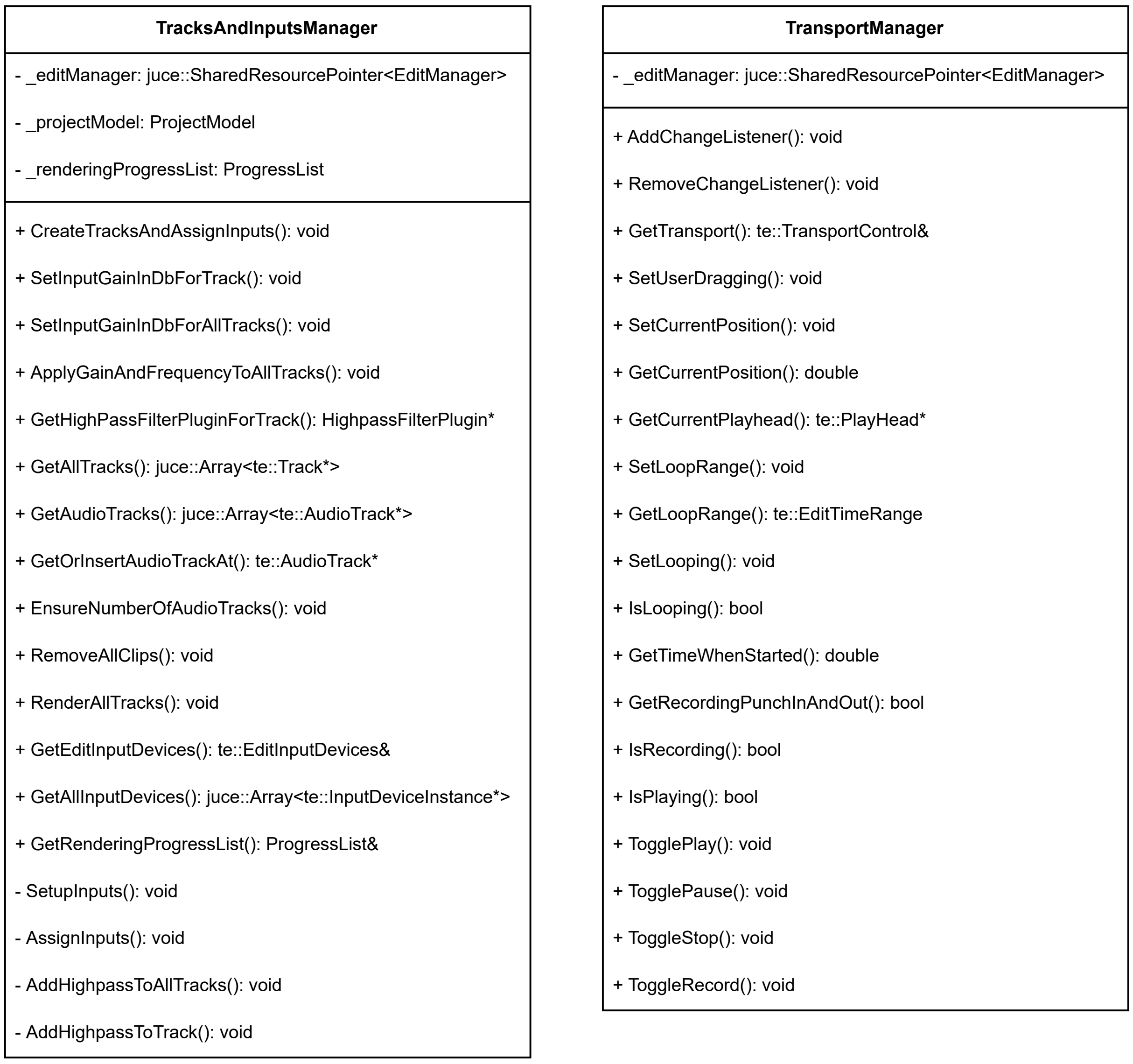

TracksAndInputsManager
- _editManager: juce::SharedResourcePointer<EditManager>
- _projectModel: ProjectModel
- _renderingProgressList: ProgressList
+ CreateTracksAndAssignInputs(): void
+ SetInputGainInDbForTrack(): void
+ SetInputGainInDbForAllTracks(): void
+ ApplyGainAndFrequencyToAllTracks(): void
+ GetHighPassFilterPluginForTrack(): HighpassFilterPlugin*
+ GetAllTracks(): juce::Array<te::Track*>
+ GetAudioTracks(): juce::Array<te::AudioTrack*>
+ GetOrInsertAudioTrackAt(): te::AudioTrack*
+ EnsureNumberOfAudioTracks(): void
+ RemoveAllClips(): void
+ RenderAllTracks(): void
+ GetEditInputDevices(): te::EditInputDevices&
+ GetAllInputDevices(): juce::Array<te::InputDeviceInstance*>
+ GetRenderingProgressList(): ProgressList&
- SetupInputs(): void
- AssignInputs(): void
- AddHighpassToAllTracks(): void
- AddHighpassToTrack(): void
TransportManager
- _editManager: juce::SharedResourcePointer<EditManager>
+ AddChangeListener(): void
+ RemoveChangeListener(): void
+ GetTransport(): te::TransportControl&
+ SetUserDragging(): void
+ SetCurrentPosition(): void
+ GetCurrentPosition(): double
+ GetCurrentPlayhead(): te::PlayHead*
+ SetLoopRange(): void
+ GetLoopRange(): te::EditTimeRange
+ SetLooping(): void
+ IsLooping(): bool
+ GetTimeWhenStarted(): double
+ GetRecordingPunchInAndOut(): bool
+ IsRecording(): bool
+ IsPlaying(): bool
+ TogglePlay(): void
+ TogglePause(): void
+ ToggleStop(): void
+ ToggleRecord(): void

Figure C.1: Class diagram of the `TracksAndInputsManager` and `TransportManager` classes

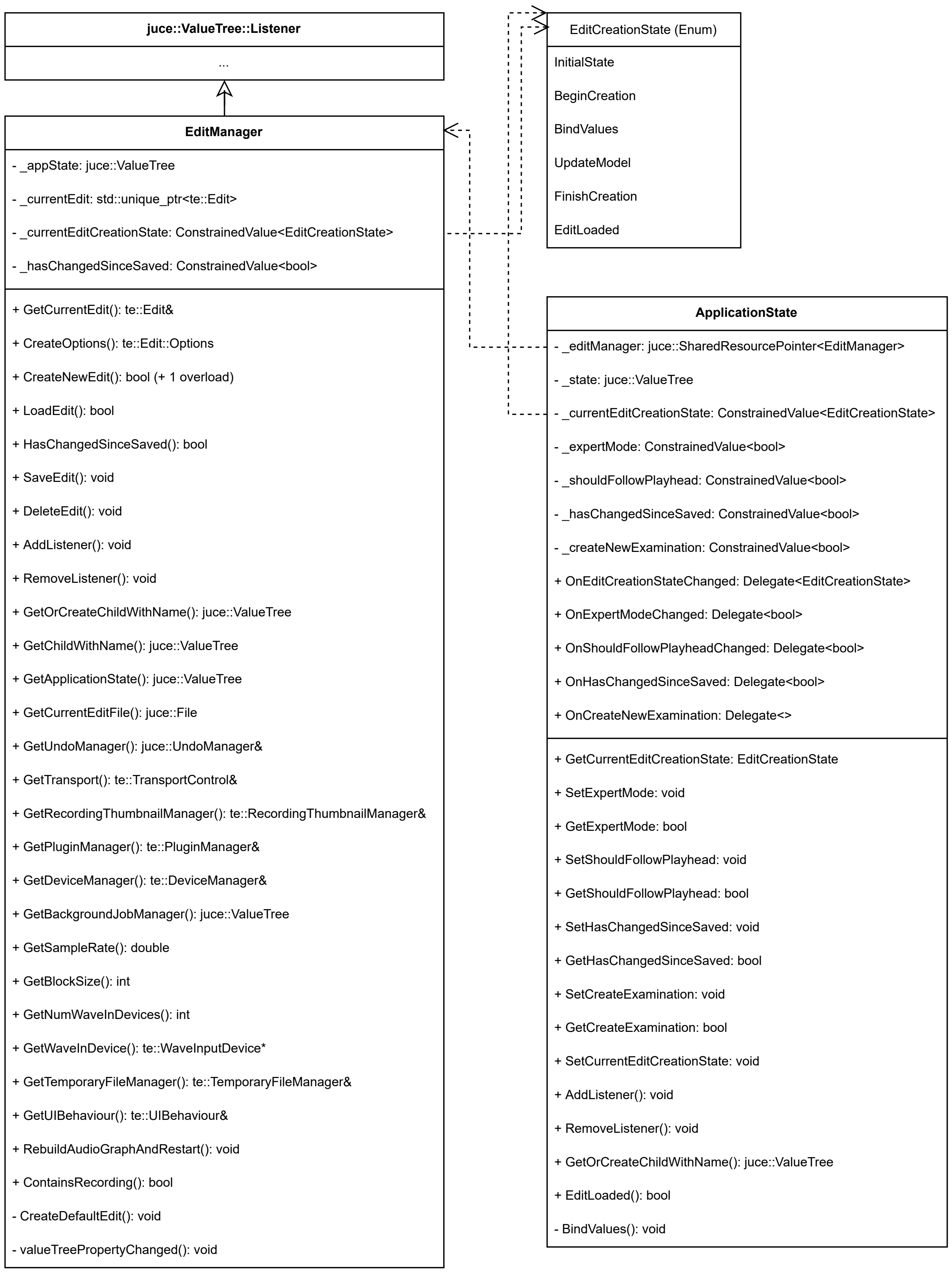

juce::ValueTree::Listener
...
EditManager
- _appState: juce::ValueTree
- _currentEdit: std::unique_ptr<te::Edit>
- _currentEditCreationState: ConstrainedValue<EditCreationState>
- _hasChangedSinceSaved: ConstrainedValue<bool>
+ GetCurrentEdit(): te::Edit&
+ CreateOptions(): te::Edit::Options
+ CreateNewEdit(): bool (+ 1 overload)
+ LoadEdit(): bool
+ HasChangedSinceSaved(): bool
+ SaveEdit(): void
+ DeleteEdit(): void
+ AddListener(): void
+ RemoveListener(): void
+ GetOrCreateChildWithName(): juce::ValueTree
+ GetChildWithName(): juce::ValueTree
+ GetApplicationState(): juce::ValueTree
+ GetCurrentEditFile(): juce::File
+ GetUndoManager(): juce::UndoManager&
+ GetTransport(): te::TransportControl&
+ GetRecordingThumbnailManager(): te::RecordingThumbnailManager&
+ GetPluginManager(): te::PluginManager&
+ GetDeviceManager(): te::DeviceManager&
+ GetBackgroundJobManager(): juce::ValueTree
+ GetSampleRate(): double
+ GetBlockSize(): int
+ GetNumWaveInDevices(): int
+ GetWaveInDevice(): te::WaveInputDevice*
+ GetTemporaryFileManager(): te::TemporaryFileManager&
+ GetUIBehaviour(): te::UIBehaviour&
+ RebuildAudioGraphAndRestart(): void
+ ContainsRecording(): bool
- CreateDefaultEdit(): void
- valueTreePropertyChanged(): void
EditCreationState (Enum)
InitialState
BeginCreation
BindValues
UpdateModel
FinishCreation
EditLoaded
ApplicationState
- _editManager: juce::SharedResourcePointer<EditManager>
- _state: juce::ValueTree
- _currentEditCreationState: ConstrainedValue<EditCreationState>
- _expertMode: ConstrainedValue<bool>
- _shouldFollowPlayhead: ConstrainedValue<bool>
- _hasChangedSinceSaved: ConstrainedValue<bool>
- _createNewExamination: ConstrainedValue<bool>
+ OnEditCreationStateChanged: Delegate<EditCreationState>
+ OnExpertModeChanged: Delegate<bool>
+ OnShouldFollowPlayheadChanged: Delegate<bool>
+ OnHasChangedSinceSaved: Delegate<bool>
+ OnCreateNewExamination: Delegate<>
+ GetCurrentEditCreationState: EditCreationState
+ SetExpertMode: void
+ GetExpertMode: bool
+ SetShouldFollowPlayhead: void
+ GetShouldFollowPlayhead: bool
+ SetHasChangedSinceSaved: void
+ GetHasChangedSinceSaved: bool
+ SetCreateExamination: void
+ GetCreateExamination: bool
+ SetCurrentEditCreationState: void
+ AddListener(): void
+ RemoveListener(): void
+ GetOrCreateChildWithName(): juce::ValueTree
+ EditLoaded(): bool
- BindValues(): void

Figure C.2: Class diagram of the `EditManager` and `ApplicationState` classes

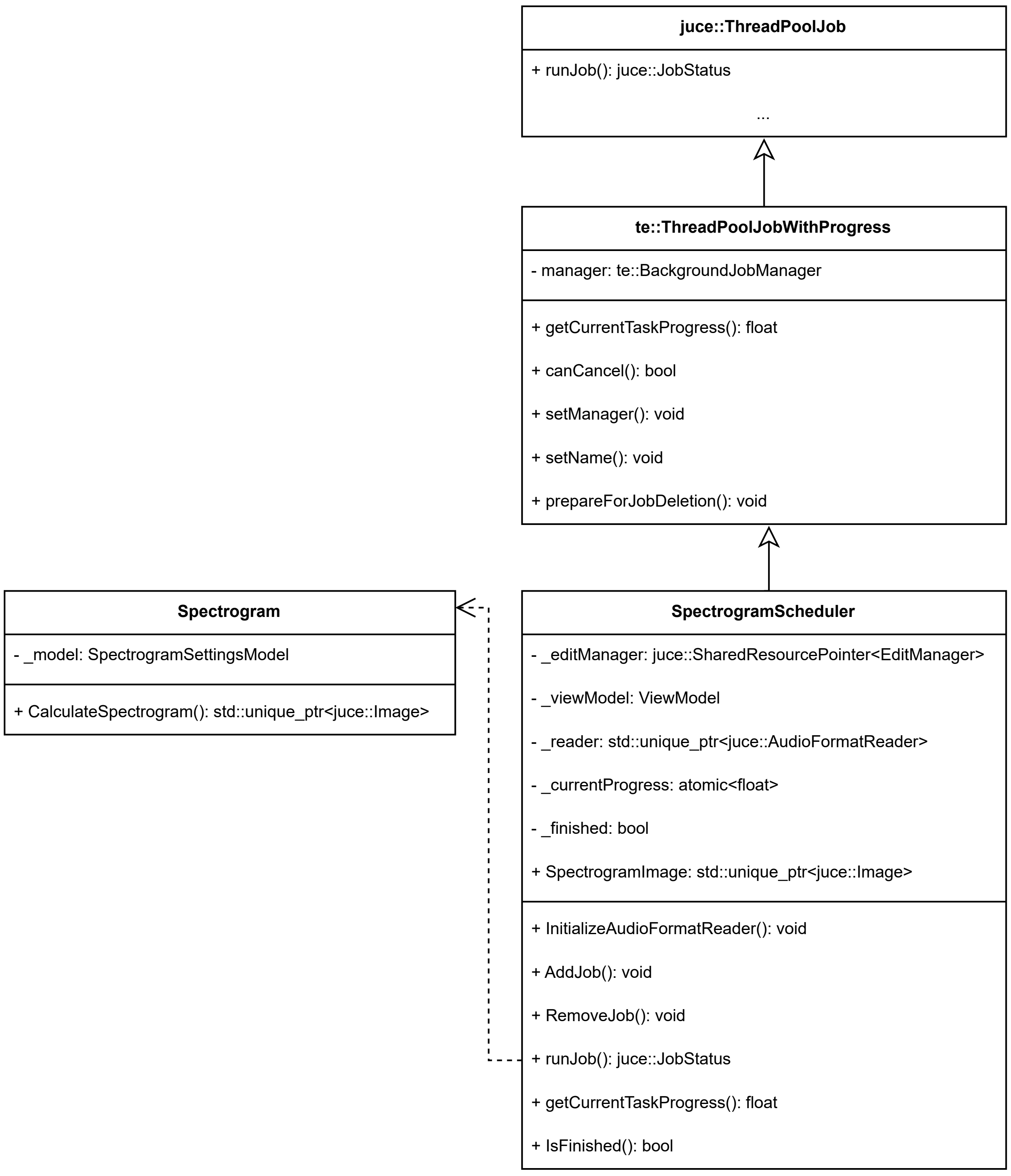

juce::ThreadPoolJob
+ runJob(): juce::JobStatus
...
te::ThreadPoolJobWithProgress
- manager: te::BackgroundJobManager
+ getCurrentTaskProgress(): float
+ canCancel(): bool
+ setManager(): void
+ setName(): void
+ prepareForJobDeletion(): void
Spectrogram
- _model: SpectrogramSettingsModel
+ CalculateSpectrogram(): std::unique_ptr<juce::Image>
SpectrogramScheduler
- _editManager: juce::SharedResourcePointer<EditManager>
- _viewModel: ViewModel
- _reader: std::unique_ptr<juce::AudioFormatReader>
- _currentProgress: atomic<float>
- _finished: bool
+ SpectrogramImage: std::unique_ptr<juce::Image>
+ InitializeAudioFormatReader(): void
+ AddJob(): void
+ RemoveJob(): void
+ runJob(): juce::JobStatus
+ getCurrentTaskProgress(): float
+ IsFinished(): bool

Figure C.3: Class diagram of the `Spectrogram` and `SpectrogramScheduler` classes

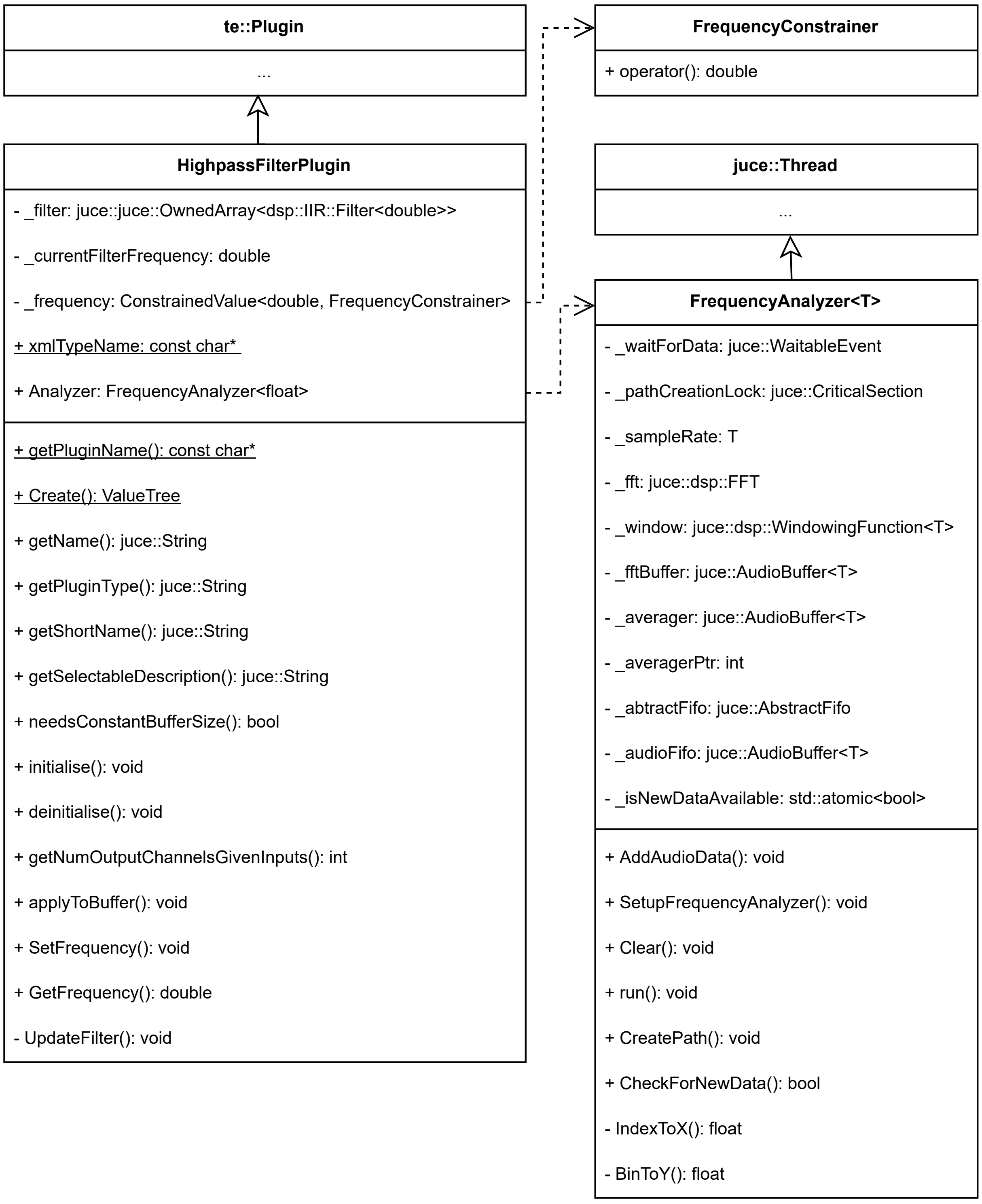

te::Plugin
...
FrequencyConstrainer
+ operator(): double
HighpassFilterPlugin
- _filter: juce::juce::OwnedArray<dsp::IIR::Filter<double>>
- _currentFilterFrequency: double
- _frequency: ConstrainedValue<double, FrequencyConstrainer>
+ xmlTypeName: const char*
+ Analyzer: FrequencyAnalyzer<float>
+ getPluginName(): const char*
+ Create(): ValueTree
+ getName(): juce::String
+ getPluginType(): juce::String
+ getShortName(): juce::String
+ getSelectableDescription(): juce::String
+ needsConstantBufferSize(): bool
+ initialise(): void
+ deinitialise(): void
+ getNumOutputChannelsGivenInputs(): int
+ applyToBuffer(): void
+ SetFrequency(): void
+ GetFrequency(): double
- UpdateFilter(): void
juce::Thread
...
FrequencyAnalyzer<T>
- _waitForData: juce::WaitableEvent
- _pathCreationLock: juce::CriticalSection
- _sampleRate: T
- _fft: juce::dsp::FFT
- _window: juce::dsp::WindowingFunction<T>
- _fftBuffer: juce::AudioBuffer<T>
- _averager: juce::AudioBuffer<T>
- _averagerPtr: int
- _abtractFifo: juce::AbstractFifo
- _audioFifo: juce::AudioBuffer<T>
- _isNewDataAvailable: std::atomic<bool>
+ AddAudioData(): void
+ SetupFrequencyAnalyzer(): void
+ Clear(): void
+ run(): void
+ CreatePath(): void
+ CheckForNewData(): bool
- IndexToX(): float
- BinToY(): float

Figure C.4: Class diagram of the `HighpassFilterPlugin` and `FrequencyAnalyzer` classes

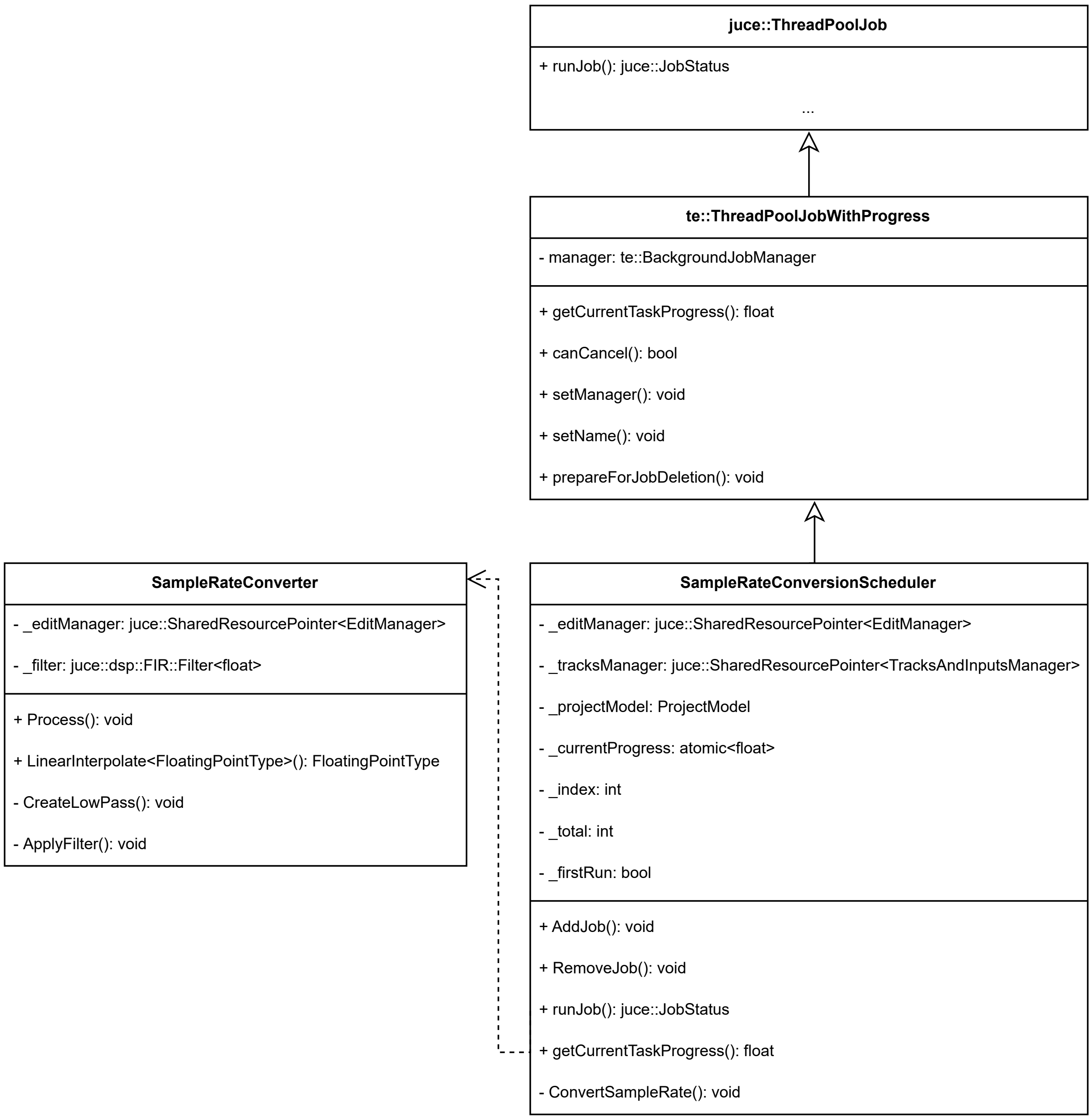

juce::ThreadPoolJob
+ runJob(): juce::JobStatus
...
te::ThreadPoolJobWithProgress
- manager: te::BackgroundJobManager
+ getCurrentTaskProgress(): float
+ canCancel(): bool
+ setManager(): void
+ setName(): void
+ prepareForJobDeletion(): void
SampleRateConverter
- _editManager: juce::SharedResourcePointer<EditManager>
- _filter: juce::dsp::FIR::Filter<float>
+ Process(): void
+ LinearInterpolate<FloatingPointType>(): FloatingPointType
- CreateLowPass(): void
- ApplyFilter(): void
SampleRateConversionScheduler
- _editManager: juce::SharedResourcePointer<EditManager>
- _tracksManager: juce::SharedResourcePointer<TracksAndInputsManager>
- _projectModel: ProjectModel
- _currentProgress: atomic<float>
- _index: int
- _total: int
- _firstRun: bool
+ AddJob(): void
+ RemoveJob(): void
+ runJob(): juce::JobStatus
+ getCurrentTaskProgress(): float
- ConvertSampleRate(): void

Figure C.5: Class diagram of the `SampleRateConverter` and `SampleRateConversionScheduler` classes

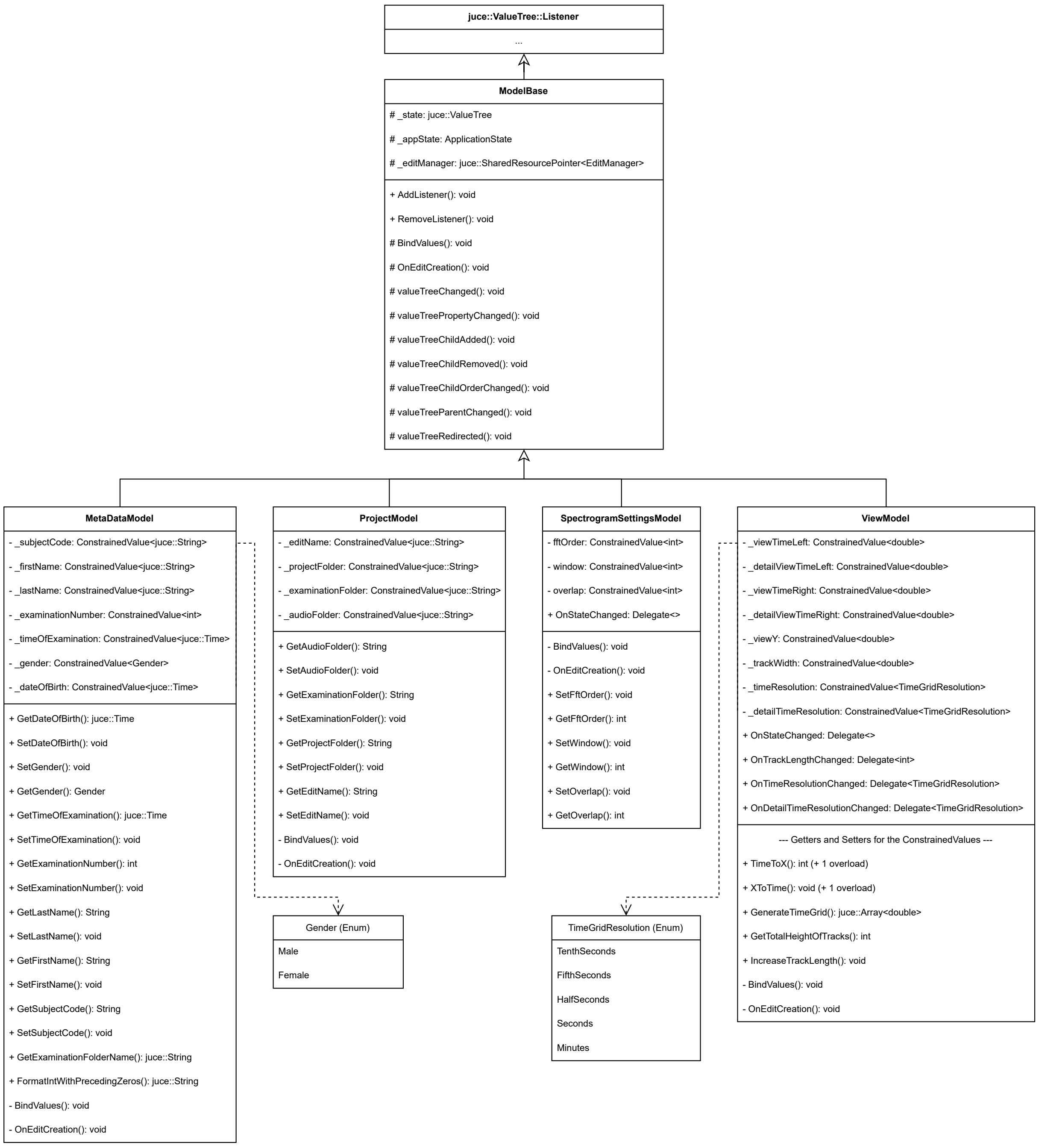

juce::ValueTree::Listener
...
ModelBase
# _state: juce::ValueTree
# _appState: ApplicationState
# _editManager: juce::SharedResourcePointer<EditManager>
+ AddListener(): void
+ RemoveListener(): void
# BindValues(): void
# OnEditCreation(): void
# valueTreeChanged(): void
# valueTreePropertyChanged(): void
# valueTreeChildAdded(): void
# valueTreeChildRemoved(): void
# valueTreeChildOrderChanged(): void
# valueTreeParentChanged(): void
# valueTreeRedirected(): void
MetaDataModel
- _subjectCode: ConstrainedValue<juce::String>
- _firstName: ConstrainedValue<juce::String>
- _lastName: ConstrainedValue<juce::String>
- _examinationNumber: ConstrainedValue<int>
- _timeOfExamination: ConstrainedValue<juce::Time>
- _gender: ConstrainedValue<Gender>
- _dateOfBirth: ConstrainedValue<juce::Time>
+ GetDateOfBirth(): juce::Time
+ SetDateOfBirth(): void
+ SetGender(): void
+ GetGender(): Gender
+ GetTimeOfExamination(): juce::Time
+ SetTimeOfExamination(): void
+ GetExaminationNumber(): int
+ SetExaminationNumber(): void
+ GetLastName(): String
+ SetLastName(): void
+ GetFirstName(): String
+ SetFirstName(): void
+ GetSubjectCode(): String
+ SetSubjectCode(): void
+ GetExaminationFolderName(): juce::String
+ FormatIntWithPrecedingZeros(): juce::String
- BindValues(): void
- OnEditCreation(): void
ProjectModel
- _editName: ConstrainedValue<juce::String>
- _projectFolder: ConstrainedValue<juce::String>
- _examinationFolder: ConstrainedValue<juce::String>
- _audioFolder: ConstrainedValue<juce::String>
+ GetAudioFolder(): String
+ SetAudioFolder(): void
+ GetExaminationFolder(): String
+ SetExaminationFolder(): void
+ GetProjectFolder(): String
+ SetProjectFolder(): void
+ GetEditName(): String
+ SetEditName(): void
- BindValues(): void
- OnEditCreation(): void
Gender (Enum)
Male
Female
SpectrogramSettingsModel
- fftOrder: ConstrainedValue<int>
- window: ConstrainedValue<int>
- overlap: ConstrainedValue<int>
+ OnStateChanged: Delegate<>
- BindValues(): void
- OnEditCreation(): void
+ SetFftOrder(): void
+ GetFftOrder(): int
+ SetWindow(): void
+ GetWindow(): int
+ SetOverlap(): void
+ GetOverlap(): int
TimeGridResolution (Enum)
TenthSeconds
FifthSeconds
HalfSeconds
Seconds
Minutes
ViewModel
- _viewTimeLeft: ConstrainedValue<double>
- _detailViewTimeLeft: ConstrainedValue<double>
- _viewTimeRight: ConstrainedValue<double>
- _detailViewTimeRight: ConstrainedValue<double>
- _viewY: ConstrainedValue<double>
- _trackWidth: ConstrainedValue<double>
- _timeResolution: ConstrainedValue<TimeGridResolution>
- _detailTimeResolution: ConstrainedValue<TimeGridResolution>
+ OnStateChanged: Delegate<>
+ OnTrackLengthChanged: Delegate<int>
+ OnTimeResolutionChanged: Delegate<TimeGridResolution>
+ OnDetailTimeResolutionChanged: Delegate<TimeGridResolution>
--- Getters and Setters for the ConstrainedValues ---
+ TimeToX(): int (+ 1 overload)
+ XToTime(): void (+ 1 overload)
+ GenerateTimeGrid(): juce::Array<double>
+ GetTotalHeightOfTracks(): int
+ IncreaseTrackLength(): void
- BindValues(): void
- OnEditCreation(): void

Figure C.6: Class diagram of the model classes

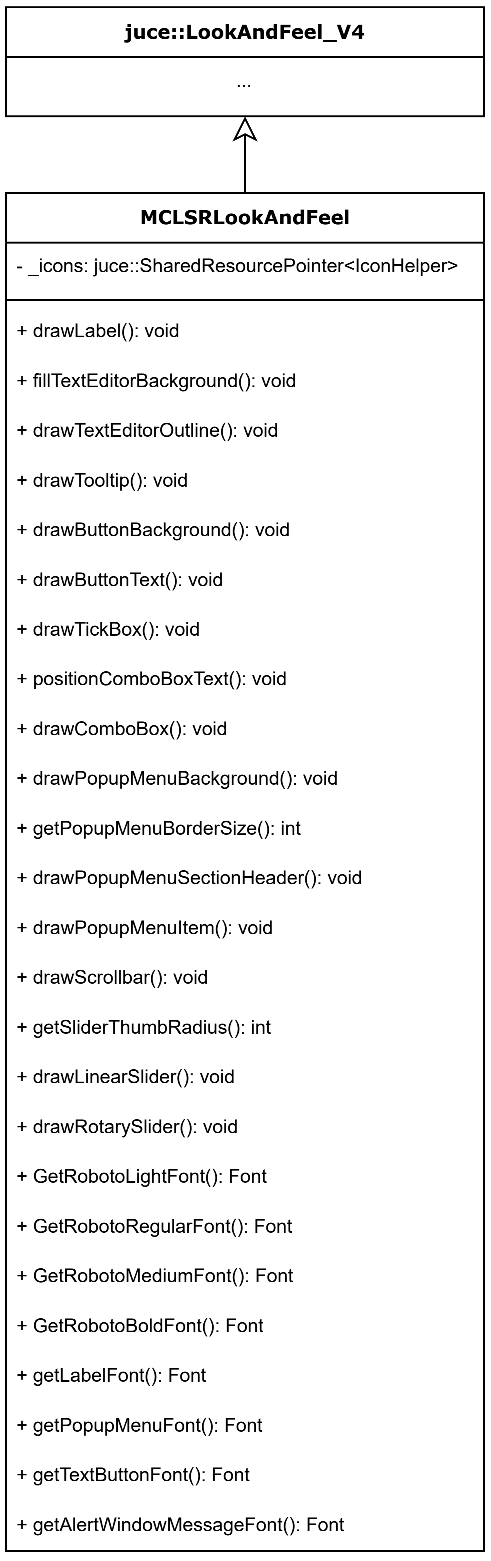

juce::LookAndFeel_V4
...
MCLSRLookAndFeel
- _icons: juce::SharedResourcePointer<IconHelper>
+ drawLabel(): void
+ fillTextEditorBackground(): void
+ drawTextEditorOutline(): void
+ drawTooltip(): void
+ drawButtonBackground(): void
+ drawButtonText(): void
+ drawTickBox(): void
+ positionComboBoxText(): void
+ drawComboBox(): void
+ drawPopupMenuBackground(): void
+ getPopupMenuBorderSize(): int
+ drawPopupMenuSectionHeader(): void
+ drawPopupMenuItem(): void
+ drawScrollbar(): void
+ getSliderThumbRadius(): int
+ drawLinearSlider(): void
+ drawRotarySlider(): void
+ GetRobotoLightFont(): Font
+ GetRobotoRegularFont(): Font
+ GetRobotoMediumFont(): Font
+ GetRobotoBoldFont(): Font
+ getLabelFont(): Font
+ getPopupMenuFont(): Font
+ getTextButtonFont(): Font
+ getAlertWindowMessageFont(): Font

Figure C.7: Class diagram of the `MCLSLookAndFeel` class

Bibliography

[1] E. Messner, "A Holistic Approach to Multi-channel Lung Sound Classification," Ph.D. dissertation, Graz University of Technology, 2019.

[2] R. X. A. Pramono, S. Bowyer, and E. Rodriguez-Villegas, "Automatic adventitious respiratory sound analysis: A systematic review," *PLOS ONE*, vol. 12, no. 5, pp. 1–43, 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177926

[3] "Web Audio API." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

[4] "JUCE." [Online]. Available: https://juce.com/

[5] "Tracktion Engine - open source audio software DAW." [Online]. Available: https://www.tracktion.com/develop/tracktion-engine

[6] "JUCE - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/JUCE

[7] "JUCE: JUCEApplication Class Reference." [Online]. Available: https://docs.juce.com/develop/classJUCEApplication.html

[8] "JUCE: DocumentWindow Class Reference." [Online]. Available: https://docs.juce.com/develop/classDocumentWindow.html#details

[9] "JUCE: Component Class Reference." [Online]. Available: https://docs.juce.com/develop/classComponent.html#details

[10] "JUCE: Thread Class Reference." [Online]. Available: https://docs.juce.com/master/classThread.html#details

[11] "JUCE: ThreadPool Class Reference." [Online]. Available: https://docs.juce.com/master/classThreadPool.html

[12] "JUCE: ThreadPoolJob Class Reference." [Online]. Available: https://docs.juce.com/master/classThreadPoolJob.html#details

[13] "JUCE: AsyncUpdater Class Reference." [Online]. Available: https://docs.juce.com/master/classAsyncUpdater.html#a4eece806c6ba9f591382fed54c5983b2

[14] "JUCE: ValueTree Class Reference." [Online]. Available: https://docs.juce.com/master/classValueTree.html#details

[15] "XML." [Online]. Available: https://en.wikipedia.org/wiki/XML

[16] "Using JUCE ValueTrees and Modern C++ to Build Large Scale Applications." [Online]. Available: https://github.com/drowaudio/presentations/blob/master/ADC2017-UsingJUCEValueTreesandModernC%2B%2BtoBuildLargeScaleApplications/UsingJUCEValueTreesandModernC%2B%2BtoBuildLargeScaleApplications.pdf

[17] "JSON." [Online]. Available: https://www.json.org/json-en.html

[18] "tracktion_Engine.h." [Online]. Available: https://github.com/Tracktion/tracktion_engine/blob/master/modules/tracktion_engine/utilities/tracktion_Engine.h

[19] "tracktion_Edit.h." [Online]. Available: https://github.com/Tracktion/tracktion_engine/blob/master/modules/tracktion_engine/model/edit/tracktion_Edit.h

[20] "Tracktion Engine PlaybackDemo." [Online]. Available: https://github.com/Tracktion/tracktion_engine/blob/master/tutorials/01-PlaybackDemo.md

[21] "tracktion_SelectionManager.h." [Online]. Available: https://github.com/Tracktion/tracktion_engine/blob/master/modules/tracktion_engine/selection/tracktion_SelectionManager.h

[22] "JUCE Repository." [Online]. Available: https://github.com/WeAreROLI/JUCE

[23] "Googletest." [Online]. Available: https://github.com/google/googletest

[24] "CMake." [Online]. Available: https://cmake.org/

[25] "FRUT documentation." [Online]. Available: https://frut.readthedocs.io/en/latest/

[26] "Model-View-Controller." [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[27] "JUCE Module Format." [Online]. Available: https://github.com/WeAreROLI/JUCE/blob/master/modules/JUCEModuleFormat.txt

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*, 1996.

[29] "JUCE: SharedResourcePointer<SharedObjectType> Class Template Reference." [Online]. Available: https://docs.juce.com/master/classSharedResourcePointer.html#details

[30] "JUCE: Listeners and Broadcasters." [Online]. Available: https://docs.juce.com/master/tutorial_listeners_and_broadcasters.html

[31] "RAII." [Online]. Available: https://en.cppreference.com/w/cpp/language/raii

[32] "JUCE: dsp::FFT Class Reference." [Online]. Available: https://docs.juce.com/master/classdsp_1_1FFT.html

[33] J. O. Smith, *Spectral Audio Signal Processing*, 2011.

[34] "Kaiser Window." [Online]. Available: https://www.mathworks.com/help/signal/ug/kaiser-window.html

[35] "Material Design Guidelines." [Online]. Available: https://material.io/design/guidelines-overview/#addition

[36] "Material Design Color Tool." [Online]. Available: https://material.io/resources/color/#!/?view.left=0&view.right=1&primary.color=2e4a66

[37] "JUCE: LookAndFeel Class Reference." [Online]. Available: https://docs.juce.com/master/classLookAndFeel.html#details

[38] "CSS Grid." [Online]. Available: https://css-tricks.com/snippets/css/complete-guide-grid/

[39] "CSS Flexbox." [Online]. Available: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

[40] "JUCE: Grid Class Reference." [Online]. Available: https://docs.juce.com/master/classGrid.html#details

[41] "Material Design Icons." [Online]. Available: https://material.io/resources/icons/?style=baseline

[42] "MCHStreamer Features and Specification." [Online]. Available: https://www.minidsp.com/images/documents/MCHStreamerUserManual.pdf

[43] "IM69D120 High performance digital MEMS microphone Description." [Online]. Available: https://www.infineon.com/dgdl/Infineon-IM69D120-DataSheet-v01_00-EN.pdf?fileId=5546d462602a9dc801607a0e41a01a2b

[44] A. Woehrer, "16 Channel USB 2.0 Sound Card for Digital MEMS Microphones," Ph.D. dissertation, Graz University of Technology, 2018.

[45] "MCHStreamer Multichannel audio interface." [Online]. Available: https://www.minidsp.com/products/usb-audio-interface/mchstreamer

[46] "Keras Documentation." [Online]. Available: https://keras.io/

[47] "Why use Keras." [Online]. Available: https://keras.io/why-use-keras/

[48] "First public version of Keras." [Online]. Available: https://github.com/keras-team/keras/commit/37a1db225420851cc668600c49697d9a2057f098

[49] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 2261–2269, 2017.

[50] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[51] "Keras Sequential API." [Online]. Available: https://keras.io/getting-started/sequential-model-guide/

[52] "Keras Functional API." [Online]. Available: https://keras.io/getting-started/functional-api-guide/

[53] "Keras Predicting." [Online]. Available: https://keras.io/models/model/#predict

[54] "Keras Layers." [Online]. Available: https://keras.io/layers/about-keras-layers/

[55] "Keras Callbacks." [Online]. Available: https://keras.io/callbacks/

[56] "Keras Project Template." [Online]. Available: https://github.com/Ahmkel/Keras-Project-Template

[57] "DenseNet." [Online]. Available: https://github.com/titu1994/DenseNet/blob/master/densenet.py