

AudioInspector

Moritz Vifian, Alexander Theler
Supervisors: Beat Pfister, Sarah Hoffmann

July 3, 2013

Abstract

The goal of this thesis was to create a cross platform application which can make annotations to an audio file. These annotations are visualized and can be edited by the user.

Contents

1	Introduction	2
2	Concept	3
2.1	Functionality	3
2.2	Programming Methods and Paradigms	3
2.3	Program structure	4
2.4	Back End	5
2.5	Front End	6
3	Implementation	8
3.1	Class Diagram (simplified)	8
3.2	Front End	8
3.3	Back End	15
3.4	Class Diagram	17
3.5	Problems	19
4	Conclusion	21
4.1	Outro	21
4.2	Outlook	21

Definitions

Tag is an annotation entity containing a beginning, possibly an ending and a text. The values of beginning and ending should be within the boundaries of the audio file

Frame is the smallest unit available, containing a sample for every channel in the audio file. It's directly linked to time via the frame rate ($[time] = [frame]/frameRate$)

Section is a certain portion of the audio file. A section has a beginning and an ending. We call these two *frameA* and *frameB*

Timeline is the axis we associate with the time dimension

Cursor can refer to a mouse cursor but also to the line indicating the actual position of the audio playback

Chapter 1

Introduction

When working with a large amount of audio files, one runs into the problem of how to make annotations inside these. Using a simple text editor and writing down the corresponding time values and notes becomes very tedious. As the number of annotations grows, one also completely loses the overview of where exactly in the audio the annotations lay. Furthermore when one wishes to share his or her notes with someone else, even more problems arise.

The goal of this group work therefore was to develop an application that assists the user with this sort of tasks. It should provide an intuitive user interface which allows quick access to the annotations and it should be useable for everyone with a little bit of experience with audio software.

Chapter 2

Concept

2.1 Functionality

As already roughly stated in the introduction, this is an overview of all features defined at the beginning of the thesis and represents, what our application should deliver.

- Opening audiofiles and display them. For a start we restrict ourselves to uncompressed PCM formats only (e.g. wav).
- Saving tags to a file. Possible file formats are:
 - htk label file format: this format is part of the htk speech recognition toolkit¹
 - srt: SubRip subtitle format which is easy to edit manually and widely supported²
- Loading tags from a file

2.2 Programming Methods and Paradigms

Object Oriented Programming

In order to have a good maintainability and extensibility of the code, the project was developed in a object oriented manner. The main advantage is that there can be an abstract outline of the classes involved (e.g. in form of a class diagram), how they interact and which methods are used.

¹<http://nesl.ee.ucla.edu/projects/ibadge/docs/ASR/htk/htkbook.pdf>, Chapter 6.2.1 HTK Label Files

²http://en.wikipedia.org/wiki/.srt#SubRip_text_file_format

GUI	Native Widgets	Java internal library
SWT	yes	no
Swing/AWT	(yes)	yes
Swing	no	yes
Qt Jambi	yes	no

Table 2.1: The shortlist of possible GUI Frame Works

Test Driven Development

Considering the back end, it is well possible to define certain functionality and according to that a set of functions which represents basically the interface to the front end. Also the classes not directly visible to the front end can be tested. For example the handling of certain tag formats can be tested explicitly.

Language

Because we want to deliver a cross platform application the choice fell on Java. Also beneficial for Java is that it provides a friendly environment for getting started with application programming since neither of us has ever programmed on a bigger project before.

Java implements most paradigms of the object oriented concepts. Using Eclipse as a development environment, testing is also included through JUnit.

GUI Framework

Our choice fell on using SWT which uses native system widgets. This means that we can have all the normal control elements that the user expects on his platform (e.g. file dialogs and even the OS X menubar). Our first impression of SWT and its structure was quite positive too.

2.3 Program structure

We divide the program into two parts. The back end deals with all data and the files on the machine while the front end provides the user interface.

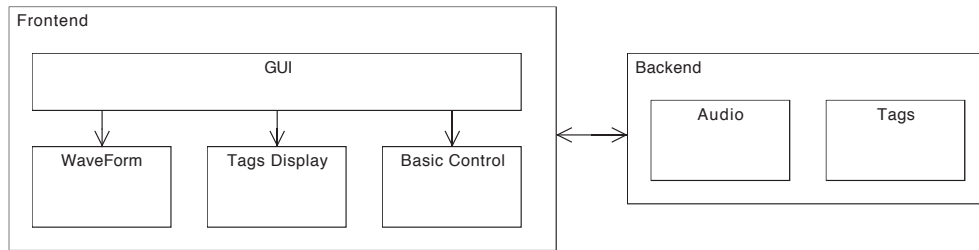


Figure 2.1: Front/back end

2.4 Back End

Tags

The basic entity dealt with here is a tag. It constitutes: a value in form of a text, a start and an end (optional). This part of the back end has the following tasks:

- Handling the creation and deletion of tags
- Reading tags from a file and/or saving them to a file
- Providing a basic undo/redo functionality

Audio

The Audio has the following tasks:

- Opening audio files from the filesystem
- Delivering sample data to the front while taking care of buffering
- Providing audio playback functionality

Downsampling

When downsampling a signal by a large factor compared to its nominal frequencies, the user expects to see the envelope rather than the waveform itself. As an example: A file encoded at a frame rate of 44 100 kHz and with frequencies between 50 Hz and 5 kHz with a duration of 4 minutes contains roughly 1 million samples. When displayed with a width of 1000 pixels it has to be scaled by a factor of 1000 which leaves the highest frequency at 44 Hz which is unlikely to be a frequency of interest.

However, there is an approach that allows us to tackle both problems at once. By taking the minimum and the maximum over a subset of samples. Although this method works the signal is overlaid with a ripple of the subsampling frequency

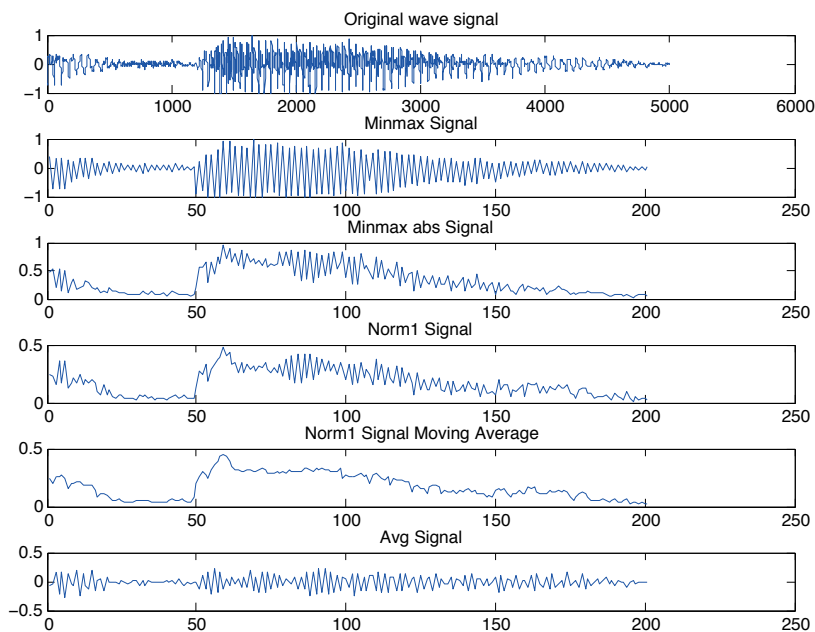


Figure 2.2: Different methods to flatten wave data for visualization.

which produces sometime not so nice outputs at certain zoom levels. But as the algorithm is very efficient and can be easily done in a recursive way, we selected this as our first method to be implemented. The outcome of the different methods is illustrated in figure 2.2.

To obtain a more beautiful image of the wave shape the two view modes must be taken into account:

- Envelope: Each subsample is the average over a norm function because average converges to zero for a DC free signal. To rescale by a factor of N $\frac{1}{N} \sum_{i=1}^N |y[i]|$ or using the power rated average $\frac{1}{N} \sqrt{\sum_{i=1}^N y^2 [i]}$. These can be combined with a moving average as a lowpass filter.
- For a detailed view one can use the normal average

2.5 Front End

The front end contains the whole graphical user interface divided into three main areas:

- Visualize the wave form of the audio file
- Show the tags

- Basic audio control

Furthermore, we created a mockup (figure 2.3) of how we expected our application to look like. This mockup also served as a helpful orientation for the GUI programming.

Main design aims for the GUI are:

- Navigate inside the audio file in an intuitive and fast way
- Visualize tags and edit them in multiple ways

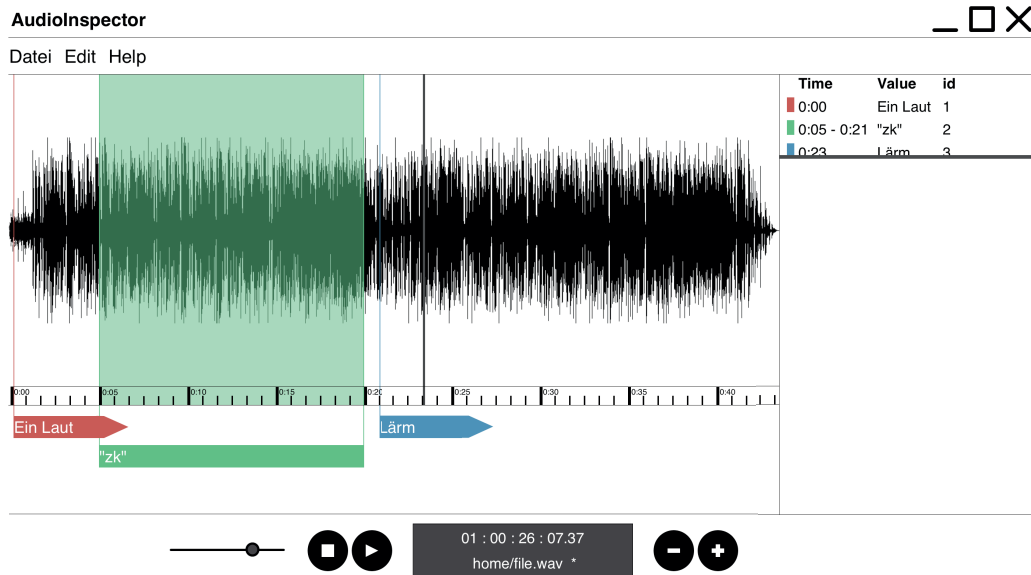


Figure 2.3: Mockup of the AudioInspector

Chapter 3

Implementation

3.1 Class Diagram (simplified)

3.2 Front End

From Frames to Pixels

On the one hand we are working with audiofiles which consist of frames and deal with time. On the other hand we have the "pixel" domain. So we need a method for switching back and forth between frames and pixels. These two domains are linked by a linear transfer function which we can be constructed in the following way.

$$[pixel] = a \cdot [frame] + b$$

$$\begin{aligned} a &= \frac{W}{frameB - frameA} \\ b &= -a \cdot frameA \end{aligned}$$

Where W is the available display space, $frameA$ the left and $frameB$ the right border of the current audio section.

GUI Structure

The structure of the front end is closely related to structure of the GUI itself (see figures 3.2 and 3.1). In SWT all control elements have a parent element to which they are chained to. The root of this tree is the **Shell** element which resides inside the **Frontend** class. Most of the other classes inherit, in some way or an other, from a SWT class (e.g. **Canvas** and **Composite**).

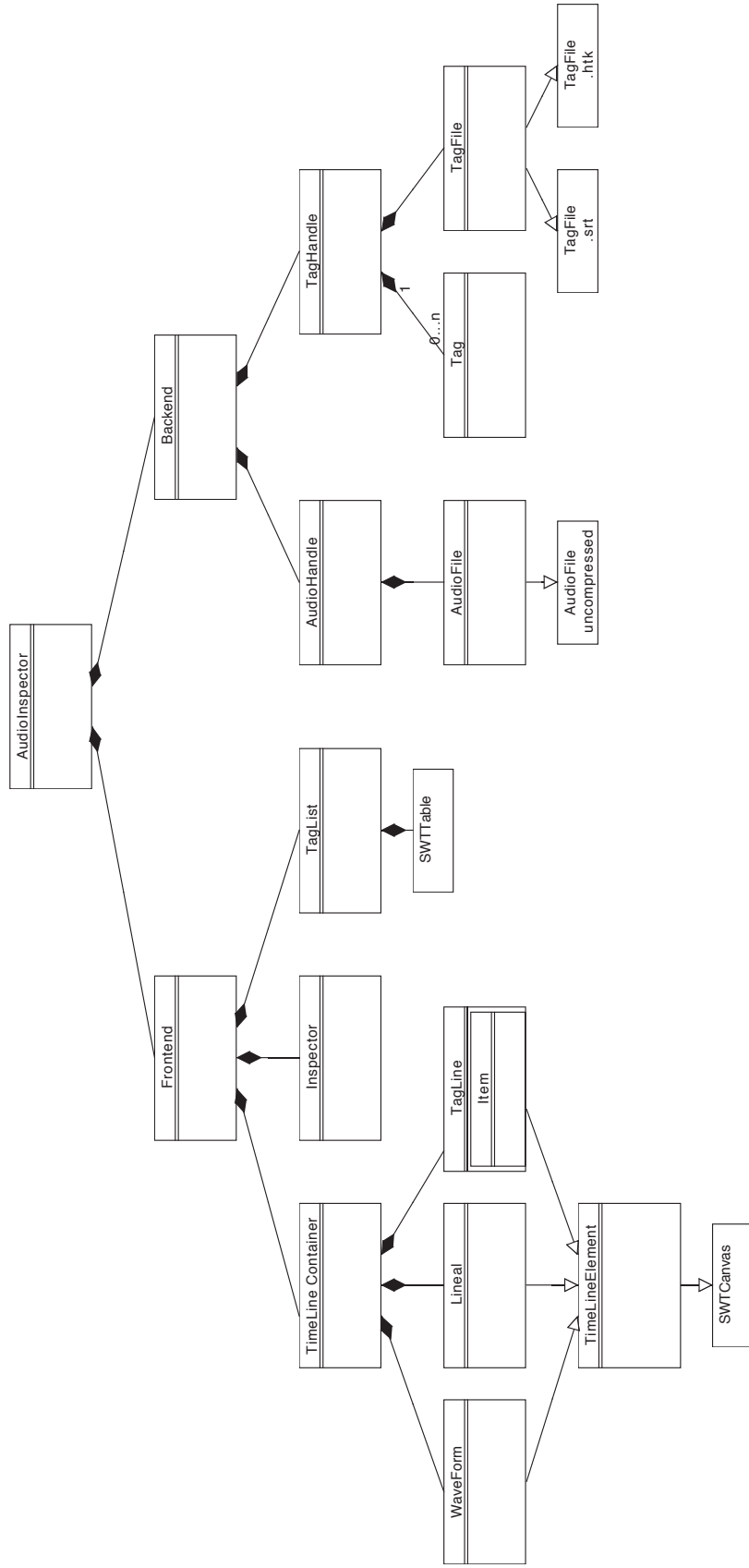


Figure 3.1: Class diagram simplified

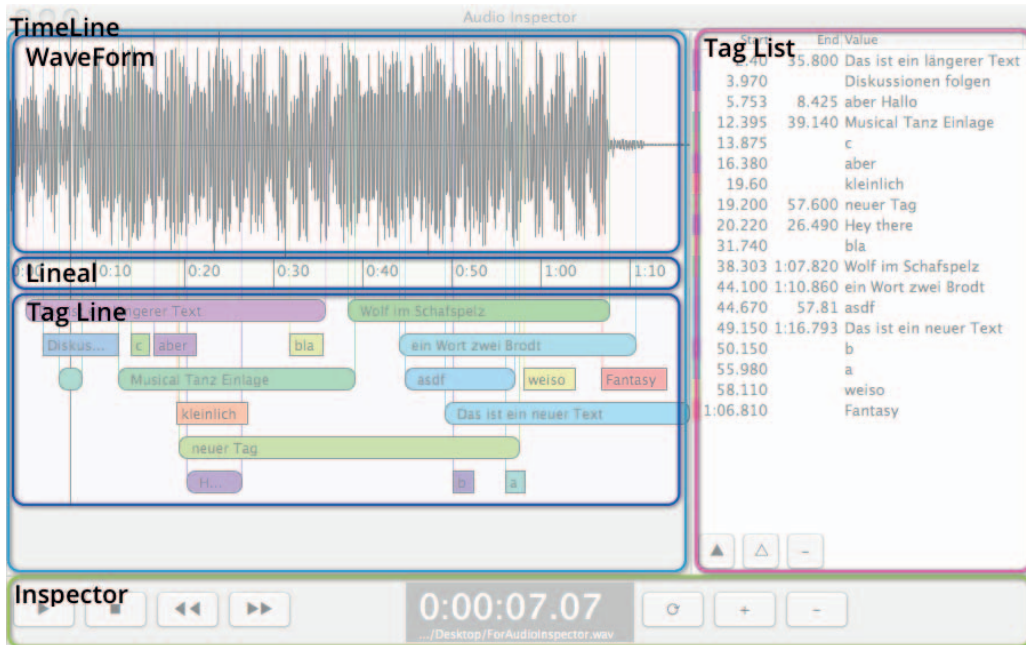


Figure 3.2: Screenshot with GUI elements

Two Custom Events

We created two Java events for the front end, called `SectionChangedEvent` and `TagsChangedEvent`. Both are triggered by user actions such as scrolling inside the audio file or deleting a tag. These two events are the driving force behind the front end/GUI.

`SectionChangedEvent` is triggered whenever the displayed section changes. That is the value of $frameA$, $frameB$ or both changes. This Event is only relevant for the control elements with a timeline and notifies these to redraw themselves.

`TagsChangedEvent` is triggered whenever a tag was created or deleted and elements displaying tags have to redraw themselves.

Zooming

Zooming in or out is realized by changing the section borders $frameA$ and $frameB$. One could simply add or subtract a certain constant amount c to each boundary.

$$\begin{aligned} newFrameA &= frameA + \alpha \cdot c \\ newFrameB &= frameB - \alpha \cdot c \end{aligned}$$

However, this leads to a somewhat irregular behavior since the amount of zooming changes absolutely and not relatively to the zooming level. In order to fix that, we could relate the amount we add or subtract to the size of the displayed section.

$$\begin{aligned}
diff &= frameB - frameA \\
newFrameA &= frameA + \alpha \cdot diff \\
newFrameB &= frameB - \alpha \cdot diff
\end{aligned}$$

This results in a more intuitive behavior since the percentage of change is equal on every zooming level. However, in the end we decided for yet another method. The idea is that we have one point P , a certain frame that acts as a fix point and doesn't change its position on the display. This leads to a situation related to the Intercept Theorem (see 3.3).

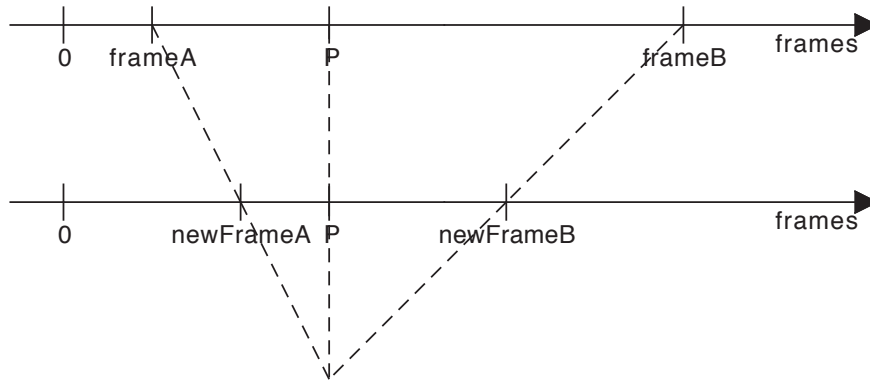


Figure 3.3: Zoom in

Given Point $P \in [frameA, frameB]$ we can calculate the new frame values $newFrameA$ and $newFrameB$ as follows:

$$\begin{aligned}
newFrameA &= frameA + \alpha \cdot (P - frameA) \\
newFrameB &= frameB - \alpha \cdot (frameB - P)
\end{aligned}$$

Where α is arbitrary constant which influences the zoom intensity. For $\alpha > 0$ we zoom in and for $\alpha < 0$ we zoom out.

Ruler aka Lineal

For orientations we have a time ruler (3.4) which connects a location on the display with a time. This isn't further difficult to accomplish. But problems arise when we consider different zoom levels and the ruler becomes cluttered with numbers or we got a lot of whitespace. What we want is a flexible way of displaying marking lines and numbers. The density of information should be equal regardless of how much we zoomed into the audio file.

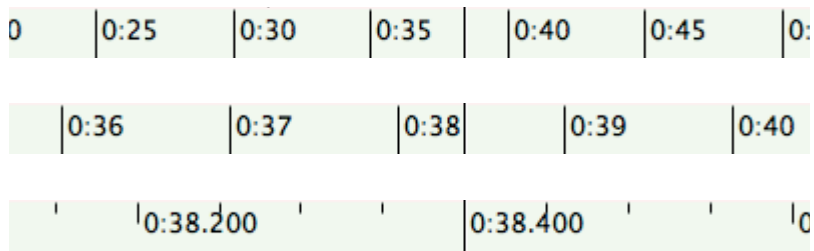


Figure 3.4: Time Lineal Screenshots with different zoom levels

Our solution for this problem goes as follows. We differ between two types of lines: *major* lines occurring at multiples of 1 second (the "whole" line) and *minor* lines occurring between the major lines (smaller lines from the top). For the *major* lines we define the possible intervals manually like 1s, 2s, 5s, 10s, and we have a minimum spacing for each type of line. We also have an array *factors* with possible denominators for getting "nice" fractions of the *major* interval (e.g. like 1/2, 1/4 etc...). Once we have a suitable minor spacing we calculate all points inside a *major* interval on which a *minor* line lays.

Text with time data is drawn on all major lines. But it is only drawn on a minor line if it doesn't collide with the previously set text.

Algorithm 3.1 Ruler: find proper spacing

```

while(major_min_space < major_interval):
    try next major_interval;

dxMajor = major_interval;

while(minor_min_space < major_interval / factor)
    try next factor;

dxMinor = major_interval / factor;

minor = new int[factor + 1];
for i = 0:1:minor.length:
    minor[i] = round(dxMinor * i);

```

Tagline Layout

We wanted to display tags along the timeline but by simply doing this we get a result like we see in figure 3.5. This representation does not help at all.

In order to solve this we put the tags on different rows according to algorithm 3.2. Tags which don't have an ending have a minimal and a maximal width (which is given by the space a text occupies). So there is some spacing flexibility for this kind of tag and the algorithm takes advantage of that. The result can be seen in figure 3.6.

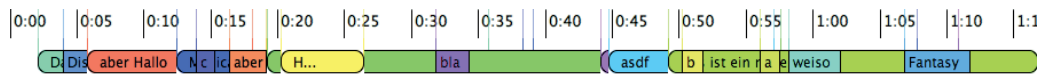


Figure 3.5: TagLine with bad layout

Algorithm 3.2 TagLine Tag Layout

```

for every tag:
    put tag on the first row;
    if collision with previously set tag:
        if collision tag hasn't an ending:
            try to resize it;
            if resize succeeded:
                continue with next tag;
        try next row;

```

Waveform Navigation

The main goal here was to provide an intuitive and quick way for navigating within the audio waveform. We decided to go for the following approach, pioneered in various proprietary audio software.¹: When the mouse is pressed the user can perform a drag gesture. Movements in the vertical direction result in zooming in/out, whereas movements in the horizontal direction in scrolling left/right. The scrolling is implemented in such a way that the clicked point is always right under the mouse cursor. This gives a feeling like moving a paper around. Both movements are possible at the same time hence quick navigation is possible.

Here is the procedure in a simplified manner. `Listener1` only listens to the first `MouseDown` event and `Listener2` is active for every `MouseMove` event.

¹e.g. Ableton Live - www.ableton.com

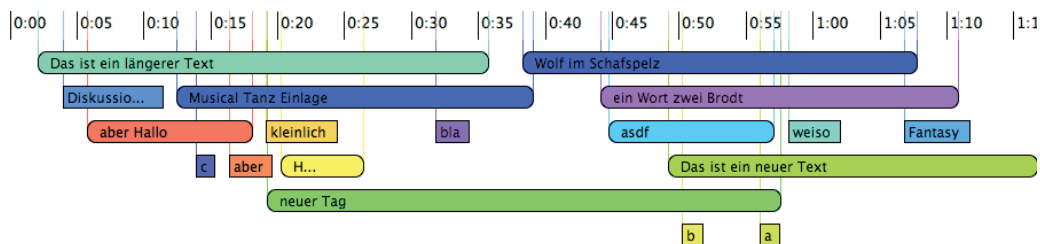


Figure 3.6: TagLine with a better layout

Algorithm 3.3 Mouse listeners for drag zoom/scroll

```
Listener1(MouseDown){
    x0 = mouse.x;
    y0 = mouse.y;
}

Listener2(MouseMove){
    if(MouseDown):
        dx = mouse.x - x0;    // delta x
        dy = mouse.y - y0;    // delta y

        if(dx!=0):
            scroll(a*dx);      // a is some constant
        if(abs(dy)>2):
            zoom(x0, b*dy);   // b is some constant

        x0 = mouse.x;
        y0 = mouse.y;
}
```

Coloring

The coloring system is implemented inside the `Helferlein` class. Every tag should have a unique color. So we generate a set with all the possible colors. Then we assign a color to each tag id. But a problem arises because of the way we handle tags: Since tags can only be deleted or created, editing a tag results in a new tag with a different id. We solved this by extending the `TagsChangedEvent` so it's possible to pass on the original id with the newly created tag.

Algorithm 3.4 color assignment

```
map: id -> color

for every newTag:
    if newTag has previousID:
        addToMap(newTag.getID(), map(previousID));
    else:
        addToMap(newTag.getID(), getRndColor());
```

Image Buffering

Having a moving playback cursor makes it necessary to redraw the `timeLine` area extensively, whereas most part of the image stays the same besides the playback cursor. All `timeLine` elements have therefore a buffer in which they paint their content when a `Section` change occurs (or in the case of the tagline also when the tags

change). For the periodic cursor redraw we can then take the pre-rendered image from the buffer and paint the cursor over it. This gives a massive performance boost.

3.3 Back End

Test Driven Development

Sample Buffer Testing As of Junit4, it's possible to use parameterized tests. This was used for testing the buffering of the different zoom levels. As an input there was a ramp generated ($y[x] = x$) where it is easy to predict what to expect from the buffering. Another advantage of the ramp is that every value is unique and it is easy to debug and see where something went wrong. Additionally, it was aimed not to have a multiple of 2 as a number of samples to test also the behavior if a sample can't be divided in two equal parts.

The test will then be executed for every element in the collection

```
@RunWith(Parameterized.class)
public class BufferTestingFile {
    @Parameterized.Parameters
    public static Collection expBuffers() {
        int larraysize =(int)IntMath.pow(2, 15) + 23201;
        linearArray = new int[larraysize];
        for (int i=0; i<larraysize; i++) {
            linearArray[i] = i;
        } } }
```

Testing Subtitle files For testing the file handling when exporting or importing tags there were expected files prepared manually and also a set of tags. One known limitation but still being tested is that there can't be a tag with a position more than 2^{32} frames (which corresponds to $27h$ at a frame rate of $44.1kHz$). In this case the tags are supposed to be neglected.

Audio Buffering

In a first approach the buffering is done through loading the entire sample into RAM.² It is then recursively calculated into smaller samples with less resolution.

In a second step the aim is to avoid loading the entire sample at once. It should be possible to constrain the buffer size (2 MB for example) that is then used for the creation of the pre-calculated data. This leads to an upper limit from which on the samples would be interpolated. This method is only used if there is not enough memory (e.g. the audio file is too large or the JVM has only little memory).

²This works also for large files (50MB) even though they cannot be played by javax.sound's sampled.clip

In a third step it should be possible to load the more close up views of the waveform directly from the hard disk. Although in the current design the samples are delivered through blocking method to the GUI. This wouldn't work too well if waiting for a hard disk. One possible solution would be to deliver low resolution samples immediately and triggering an event after reading from disk has finished.

Audio Player

The `javax.sound` library itself provides the `.sampled.clip` class which is nice to play short samples. Although this method already fails for a wav file of 4 minutes (40 MB) as it tries to load the whole file into the play buffer. Therefore the implementation of routine that loads a portion of PCM data from the input streams and writes it out to line. In our approach the player inherits from the `Thread` class. The writing to the line (plays the audio) is done in the `run` method. Later on we noticed that implementing the player is not as simple as at first glance: Even a simple action (like stopping the player) could lead to a `raise` condition.

To get a better idea how a player could be realized we took a closer look in to the `BasicPlayer` from `javazoom`³. One main difference from our approach is that `BasicPlayer` has a `STOP` state where the audio resources are freed. This is realized by implementing a `Runnable` and shutting down the thread completely in the `STOP` state.

The main reason for not just using the `BasicPlayer` was that it has no loop function. In our player is realized by wrapping the `AudioInputStream` around a `BufferedInputStream`. Thus it is possible to set a mark at input stream to which the buffer can be reset. The audio then reread from the marks position is read from the buffer instead of the original stream (file on the hard disk⁴).

Tag writing and reading from file

Both file formats supported so far are time based: `srt` down to 1 ms; `htk` down to 100 ns. Since the position of the tag is stored in frames internally the writer of the tag file must know the frame rate to convert to a time.

If there is need to support a new format this can be achieved by implementing the `TagFile` interface and add the new file extension to the `TagFileFactory`.

Basic history

With the limitation that tags can only be created and deleted, but not modified (modifying is done by deleting and then creating a new one) the history can be implemented fairly simple. Every time a tag gets removed from the tag list, it is pushed to an undo stack.

³<http://www.javazoom.net/projects.html>

⁴Most Operating System would cache the file read from the disk, but using a `BufferedStream` it's certain that data accessed in narrow section will be read from RAM

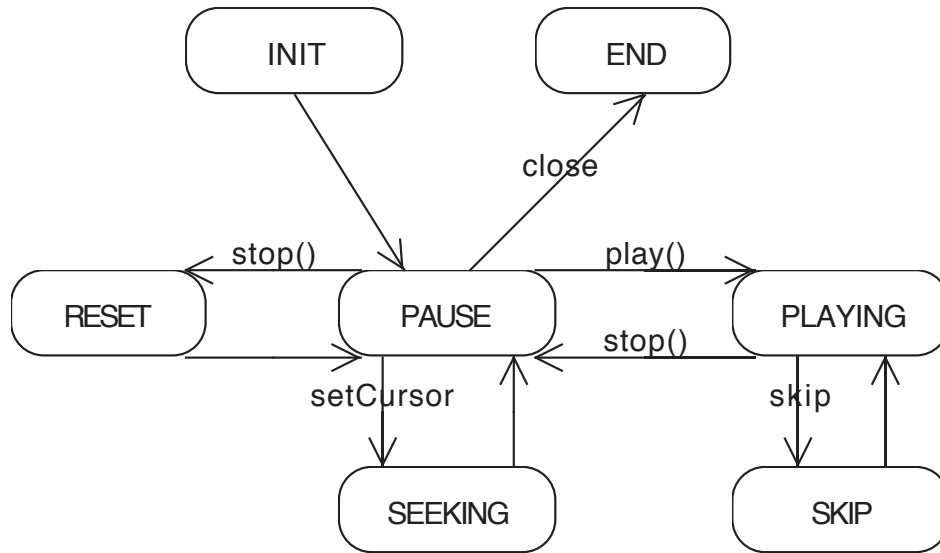


Figure 3.7: The different states of the player

A flaw of this concept (as we noticed later on) is that a drag operation of the tag consists of two operations: Deleting the old tag and creating the new one. The problem is now that this usually happens immediately and is not visible for the user. But when using the undo function, the user will have press the undo button twice.

There is no workaround. The back end cannot possibly know whether the new tag has been created by the user or by moving an existing tag; the only way would be to guess by comparing the text of the last deleted and the newly created tag. But in case of a file where the vowels are marked this is likely to not behave as expected (as the probability of deleting an “e” and creating an “e” by the user exists)

Therefore would it be necessary to have a transaction based history. The stacks for creation and deletion of tags can still be used but when modifying a tag, the backend needs to make a deep copy of the old values of the tag and store it in a way that the values can then be associated with right tag when restoring at an undo/redo event.

3.4 Class Diagram

3.8 is a complete class diagram of our program with the most import methods and fields.

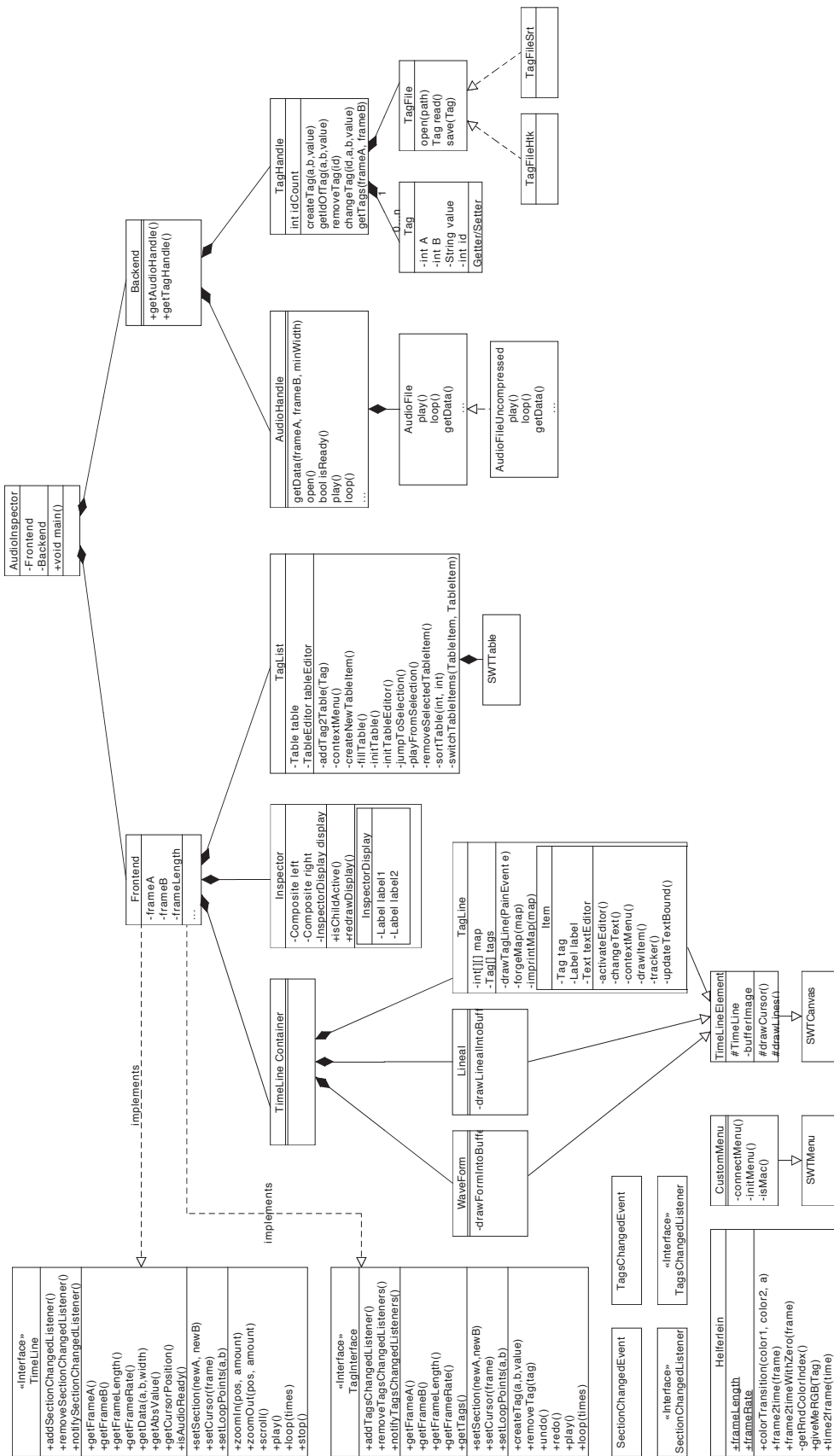


Figure 3.8: Class Diagram of Audioinspector

3.5 Problems

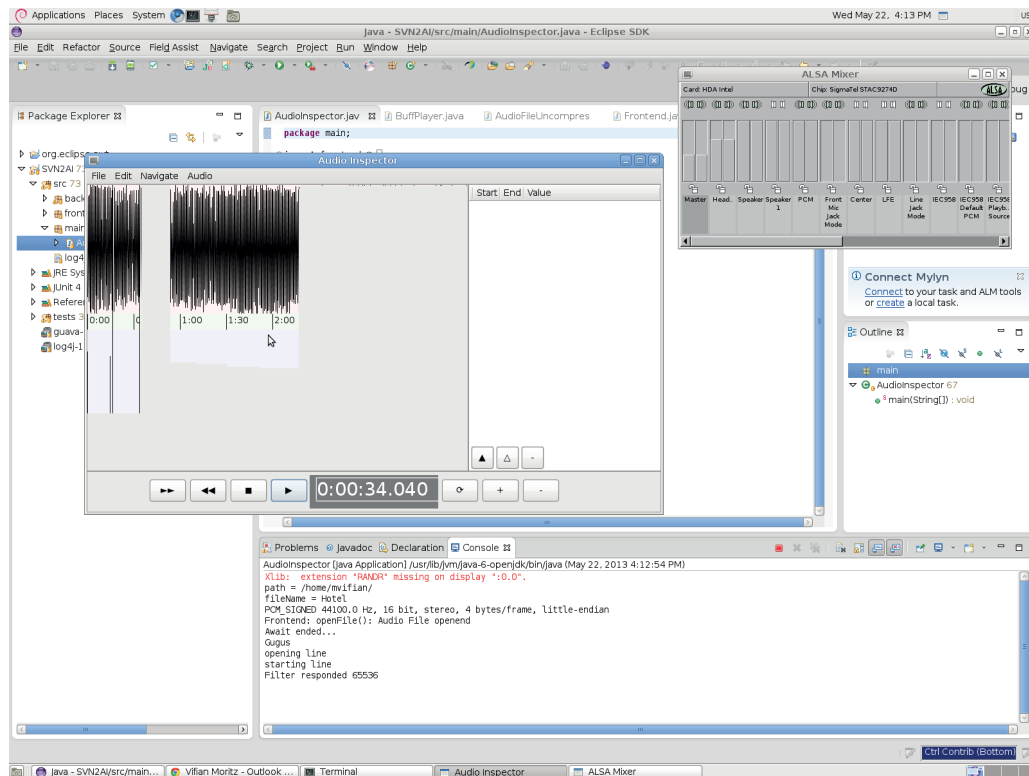
Timers in SWT

SWT doesn't support a runnable to be executed in a certain period. Although there is the method `display.syncExec(runnable)` that is used for runnables that need access to any swt related classes. A periodic scheduled function can thus be implemented in two ways

1. Create a runnable that does something, waits for some time (using `thread.sleep`) and executes itself again recursively from within the runnable. The advantage is that it is ensured that the runnable cannot run more than once.
2. Create a scheduled thread that calls `display.syncExec()` only for the part where access to the GUI is needed.

Differences between OS X and Xorg

As in the cocoa implementation (or OS X in general) each Window has a buffer where it is drawing onto. Therefore a window that has been covered by another one and gets uncovered again does not need be redrawn (if nothing has changed). This behavior is easy to observe when program crashes or is stopped with `ctrl+z`: The content of the window is still visible. When we first tested our grown program on a linux machine it really behaved strangely. There seemed to be a bug that the content wasn't redrawn at certain events. But on OS X it is impossible to detect.



Platform dependent Widget availability

SWT uses (wherever possible) the native widgets from the OS it is running on. In comparison to AWT the API is the same for all platforms. This is nice for the programmer but has the severe drawback that some widgets (especially on linux) are not available from the the native toolkits (Motif or GTK2) and are implemented in Java. This leads to some ugly glitches in the rendering of those widgets and some oscillating/flickering texts.

Chapter 4

Conclusion

4.1 Outro

All in all it was an interesting experience. First, both of us have learned a lot while realizing this project concerning Java, OOP, GUI in general but mostly how import a good conceptualization can be. It is quiet impressive to realize how much effort lays behind a fairly small program like ours and to divine how much this effort would scale up regarding a "real", commercial program.

Second we are happy that we reached all the goals we set out to realize. The basic concept of the program is good but for next time we would rethink the way of how the front and back end's communicate with each other.

We don't think that our program meets all the expectations of the speech lab yet, but hopefully it can figure as a basis for such an application in the future. Moreover we hope that we could deliver a few useful inputs and concepts regarding an intuitive user interface, that possibly will prosper in another way.

4.2 Outlook

Possible extensions/improvements for the overall program could be:

Waveform display: Another approach for displaying the waveform could be to find the envelop of the form and then fill it. Also displaying stereo files could be useful.

Undo/Redo: The bug of our implementation, where the user has to press Undo/Redo twice could be fixed by adding a editing function besides create/delete.

Using Existing Audio Player: Wrapping our interface around an an existing Player would add support for more audio formats.

FFT-View: Alongside the waveform display one could add a frequency related view where the x-axis still represents time but the y-axis stands for frequency and the color of a dot could represent the intensity.

Front/Back end communication: This could be extended. Exceptions should be caught by the front end. The back end could provide listeners (e.g. cursor movement, audio buffering progress)

Stability: The overall stability could be improved with better error handling and user notification

Better performance: Tagline has to be fixed so we can visualize more than, let's say 100 tags.

"Helper functions": Add program functions that assist the user with making annotations (e.g. transient detection).