Stellenbosch University

Honours project

Music identification tool: Stelhound

Design document

Matthew Jenje

supervised by
Prof. Brink van de Merwe

April 30, 2017

Contents

List of Figures

Listings

1 Introduction

This design document describes the architecture and system design for the software development project "Stelhound". This document is intended to fulfill all requirements outlined in this projects requirements document. This document shall discuss data structures, algorithms and interfaces for this development project.

2 System architecture

The systems functions can be described by the following components:

1. Audio recording
2. Audio preprocessing
3. Audio interpretation and extraction
4. Creating the custom fingerprint
5. Database querying with custom fingerprint
6. Returning meaningful results to the user

The relationships between each of the components can be seen below in figure 1.

The audio recording will be initiated on input from the user interface and shall run concurrently with the search operation to return results in as little time as possible. The recording is a crucial part of the matching process and if done incorrectly, the generated fingerprints may be inaccurate and matches may never be found. The audio recording process starts the main functionality of the program.

The audio preprocessing is responsible for removing any unwanted noise from a recorded audio sample. This process should help reduce false positives in a conversion of the sound file into its fingerprint. This shall let the user record sounds in any environment without worrying that the background noise is interfering with their results.

The audio interpretation component is responsible for taking the raw sound data (after de-noising) as input and converting it via the Fast Fourier Transformation (FFT). This changes the sound data from the time domain into the frequency domain which assists in retrieving the pitch data responsible for determining the song's fingerprint.

The creation of the custom fingerprint involves sampling information many times a second and performing Fourier transformations on the data. This information is then checked and transformed into a custom fingerprint via a hashing

function. The custom fingerprint will be used to query the database and return the correct match. Fingerprints generated from partial recordings should be substring matches for a song's full fingerprint.

The database shall be queried with the generated fingerprint to return matches and partial matches. This process is the final step in the program and shall be done more than once with longer fingerprint references if too many matching results are returned by the database.

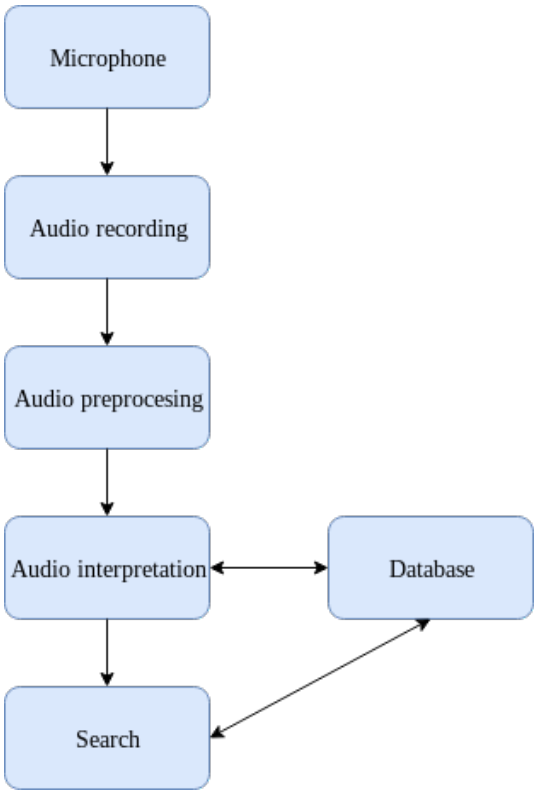

Figure 1: Relationship between main components

2.1 Audio recording

Audio recording in Java is a well know operation which produces high quality recordings and store them into `byte []` arrays. These are important in the process of turning the sound data into frequency data.

In order to have consistent recordings, several parameters need to be set which tell the system information such as how many times to sample a sound or how many channels the sound is recorded on. Luckily, Java comes with a library `javax.sound.sampled` which will be used to set up the recording information and make sure that recordings are consistent. These same values will need to be used when generating the custom fingerprint as differences in information such as sampling rates would cause the same song to have different fingerprints. Below is the Java implementation of how audio shall be formated when recordings are made through the microphone.

```
protected static AudioFormat audioFormating () {
     final float sampleRate = 44100.0f;
     final int sampleSizeInBits = 16;
     final int channels = 1;
     final boolean signed = true;
     final boolean bigEndian = true;
     AudioFormat format = new AudioFormat(sampleRate, sampleSizeInBits,
         channels, signed, bigEndian);

     return format;
}
```

Listing 1: Audio formating for recordings

There are several steps required in order to record audio in Java:

1. Get the information of the data-line: the data-line is formated using the `javax.sound.sampled` library to get information about the recording peripherals that are present on the device running the program. If no recording devices are detected an exception is thrown.

2. Opening the data-line: this step is done provided that there is a recording device available. By opening the data-line the program is now given access to the information being generated by the recording device. This information is recorded and sampled with the same parameters as defined in the listing above.

3. Saving the recordings: a loop is created which runs until a stop signal is received. For every iteration of the loop, information is read into a buffer and sent into a `ByteArrayOutputStream()`. This stream then stores the entire recording which will first be cleaned of noise and used to generate the custom fingerprint.

Recordings shall run for a minimum of 3 seconds in order to generate enough information for accurate matching as matches in less time may be unreliable.

Audio recordings shall continue until either the user stops the program or enough information is gathered to determine that there is no corresponding database entry. If the user decides to stop the recording, no results shall be returned.

Table 1: The message directory for the audio recording component

Action	Req	Description	Args	Return
recordSound	R1	Records the sound and put it in a byte array.	None	Byte Array
playAudio	R1	Plays the recorded audio file. Used as a quality checker	Byte Array	None
audioFormating	R1	Provides the audio formating for the microphone when recording	None	Format

2.2 Audio preprocessing

In order to have a sound file ready for processing, it needs to be cleaned of as much background noise as possible. This is achievable in several ways. One way is to take a short recording of the room tone and then subtract the common frequencies from the audio the user wishes to record. In this way, there is a noise baseline which is know to not be in the desired sound and can be used to clean the audio up afterwards. The issues with this method is that it requires a recording of the room without the desired sound first and there is no guarantee that the noise in a room is always uniform. The other way is to use a low passing filter which removes the highest frequencies in a sound file. This effectively removes unwanted information and leaves the relevant parts of the file. The issue with this technique is that the filter may remove information pertaining to the desired sound, removing important data. Both of these methods have issues which may create an imperfect fingerprint and thus will not return correct results when used in a search.

One potential option is to not do any noise reduction. The process used to extract the fingerprint from the recording involves the use of Fourier transforms which separates the sound into its frequency components. These frequencies are then analyzed and only the highest frequencies are used to create the hash values for the song's fingerprint. This process of only choosing the frequencies with the highest amplitude will mean that only the loudest sounds in a room will become part of the fingerprint, which automatically removes a large amount of noise information. In cases where the frequencies with the highest amplitude in the room are not the song, the program will not return any matches. The program will thus only work on the condition that the song is louder than the average noise level in the room.

Another potential option is to use source separation techniques. Source separation involves listening to an audio recording and separating the audio into different parts accurately, i.e. being able to determine which sounds in a recording are the noise and which sounds are the music we wish to identify. By doing source separation we can eliminate the noise in the room and only use the desired audio signal.

One possible technique is Blind Signal Separation (BSS) which is an attempt to recover unobserved signals from several observed mixtures [8]. This technique is quite complex and often involves more than one microphone picking up different sets of signals and then mixing those two signals together to get some sense of distance and location. It is possible to do BSS with a single microphone, however the margin of error is higher than if there were more recording sources. One approach to doing BSS involves taking the Singular Value Decomposition (SVD) of the data by placing the data in a matrix and drawing conclusions based on how the singular values correlate. This is an extremely complicated process and some experimentation is required in order to determine whether to use BSS or to only use Fourier analysis.

2.3 Audio interpretation

Audio needs to be processed with a Fourier transform in order to convert the audio from the time domain, into the frequency domain. This is useful in determining which frequencies are most prominent in an audio sample and these prominent frequencies can then be used to determine the custom fingerprint for an audio file. The Fourier transform variant which will be used in this program is the Fast Fourier Transform (FFT). The FFT can compute a Fourier transform far quicker than a standard Fourier analysis, computing results in $\mathcal{O}(nlog_n)$ time compared to $\mathcal{O}(n^2)$, where n is the number of samples being transformed.

2.3.1 Fourier analysis

Fourier analysis needs to be done many times a second in order to extract every dominant frequency from an audio segment. The frequencies with the highest amplitude are then selected and used as the primary factor in the hash function which generates the fingerprint. Doing a Fourier analysis it pivotal in determining a fingerprint for a song.

The Fourier analysis cannot be done at even time spaces if the goal is to only extract the melody. This is due to the nature of melodies as they do not necessarily conform to the the same rhythm in each bar throughout a song. Rhythms can also change as a song progresses, meaning that if a pattern could be established in the rhythm, there is no guarantee that it will hold for the rest of the song. This problem can be solved in one of two ways: a custom fingerprinting system can be used which incorporates song sampling at even time intervals, or a system needs to be put in place which allows the program to only do Fourier analyses over areas in the recording where the melody is present which would produce a melodic contour. Both of these solutions have problems associated with them and these will be explained bellow, with section **2.3.1.1** referring to the first method and section **2.3.1.2** referring to the second.

2.3.1.1 Custom fingerprint

Generating a custom fingerprinting system requires the use of a custom built database, preventing the use of online web services such as Musipedia in the programs solution. The database would at least need to be constructed with a custom fingerprint field, song title, song name and duration. These fingerprints would be generated with a function taking a full `.wav` file as input, and producing a set of hash values as output. The database would contain the full fingerprint of each song and searches would look for substring matches in each fingerprint.

This method of fingerprinting will be able to identify multiple genres of music, including instrumental. The fingerprinting process looks at chunks of the song without singling out any information and generates the hash. This means that as long as the hash has been stored in the database, it can be found again

if it is searched for with a recording.

This method is quite similar to method **2.3.1.2** in how it executes, differing in that the fingerprinting method uses the entire song as a reference without singling out the melody, whereas method **2.3.1.2** focuses solely on the melody.

- Pros
  - Allows for a custom search algorithm to be used
  - Allows the developer to have full control over the database and its contents
  - Should provide higher accuracy in matches
- Cons
  - Will take a substantial amount of time populating the database for practical use
  - Will require more development time than method **2.3.1.2**

2.3.1.2 Melodic contour

Determining whether a small section of a song is part of the melody or part of the background is challenging for a program to do. Given an audio file we can only gather three pieces of information: the time, amplitude and frequency and this is not enough information to always determine if a sound sample is part of the melody. In general, the melody of a song is the most noticeable sound in a section of audio, which normally means that the frequency of the sound with the highest amplitude is part of the melody. This however is not strictly true. In some percussive songs where the drum is the dominant sound, the melody will not be distinguishable to a computer program as the amplitude of the melody is lower than the amplitude of the percussion. This is the first challenge faced in determining the melody.

The second problem in determining a songs melody is changing the size of the sampling window for the Fourier analyses so as not to sample sections without the melody in them, or to sample sections with more than one note of the melody in them.

One possible solution to both of the above problems is to determine an average amplitude over the duration of the recording and only accept frequencies which are above this average. This method will still have the drawback of including some tones which are not actually part of the melody and this could skew the accuracy of results.

This method would use Parsons code (a representation of a songs melody which looks at each pitch relative to the one before it and builds a contour) to determine whether a song is a match or not. This has the benefit of having the

Musipedia web-service database already in place so no database population is required.

- Pros
  - Database already in place
  - Generated Parsons codes can easily be checked for validity
- Cons
  - No control over what information is in the database
  - No choice in the method used to search through the database
  - Parsons codes do not contain enough unique information to accurately pick one song out of a database

From the above comparison between option **2.3.1.1** and **2.3.1.2**, a decision was made to implement option **2.3.1.1**. By having control over the database it is possible to create the fields needed to return meaningful results to the user, create a search algorithm which is much faster than that of the web service and there should be a higher match accuracy. For these stated reasons, option **2.3.1.2** will not be implemented.

Table 2: The message directory for the audio interpretation component

Action	Req	Description	Args	Return
convertToDouble	RD1.1	Changes the audio to a double array	audio	doubleAudio
makeWindow	RD1.1	Creates a sample window	None	None
fft	RD1.1	Performs a FFT	Real and imaginary parts of the audio	newReal, newImaginary
beforeAfter	RD1.1	Prints the result for debugging	Real and imaginary parts of the audio	None

2.4 Creating a custom fingerprint

Creating a fingerprint for a song is one solution to the problem of uniquely identifying songs based on their frequency composition. The fingerprint for every song will be different which allows for both partial and exact matches without duplication.

The fingerprint of a song will consist of a long list of hash values, created by segmenting each second of a song roughly 20 times and generating a hash value for each segment. Some experimentation is required to determine whether this is the optimal density for audio recording. This creates a large number of hash values and ensures that each song has a unique fingerprint.

The benefit of using this fingerprint method is that the Fourier analysis of a songs data can be computed at even time intervals, allowing the program to have repeatability and not rely on an algorithm to always correctly sample a song on the melody.

Fingerprinting of songs will take place in two different areas: one in generating the entire fingerprint of a song before it gets entered into the database, and the other when recording on the microphone, generating part of the song's fingerprint for every second of the recording. The idea is that when recording a song through the microphone, no matter at which point in the song the recording is started, there should be a substring match between the database fingerprint and the recorded fingerprint. The longer the recording runs, the more unlikely it is to find songs with the same set of fingerprint values and so a song should eventually be singled out.

There are a few simple steps to creating the fingerprint:

1. Receive the `byte array` of the sound we wish to generate the fingerprint of, either through recording or reading in a file
2. Convert the `byte array` into a `double array`
3. Segment the `double array` into even sized chunk's
4. Run a Fast Fourier transformation on each chunk to get the frequency distribution of the chunks
5. Find the highest frequencies in each frequency band [1] for each chunk
6. Use the highest frequencies to generate a hash value with a hash function
7. Repeat the above process until every chunk in the recording has been assigned a hash value.

[1] The frequency bands in this case are <200Hz, 200-400Hz, 400-600Hz, 600z-1000Hz and 1000-1200Hz. These bands cover most of the main frequencies in a song for low, mid and high tones.

An important step in the above process is the frequency hashing. The hash is used to give the frequency data for each chunk a consistent size, as well as to do minor error-correction with the use of a "Fuzz factor". Below is a listing showing the hashing function implemented in this program[9].

```
private static long hash(long p1, long p2, long p3, long p4) {
     return (p4 - (p4 % FUZ_FACTOR)) * 100000000
          + (p3 - (p3 % FUZ_FACTOR)) * 100000
          + (p2 - (p2 % FUZ_FACTOR)) * 100
          + (p1 - (p1 % FUZ_FACTOR));
  }
```

Listing 2: Hash function used in the fingerprinting process

Once the fingerprint is generated it will either be inserted into the database if it is for an entire song, or it will be used as a lookup value to search through the database. If the fingerprint is used as a lookup the fingerprinting process will happen multiple times until only a single result is returned back to the user.

Table 3: The messaging directory for the fingerprinting component

Action	Req	Description	Args	Return
fingerprint	RD1.2	Creates the fingerprint	None	fingerprint
getIndex	RD1.2	Gets the index of a frequency	freq	index
hash	RD1.2	Hash's the input	(p1, p2, p3, p4): the top frequencies in each frequency band	hash value
convertToDouble	RD1.1	Changes the audio to a double array	audio	double array
audioFormating	RD1.1	Provide audio formating for the microphone	None	None

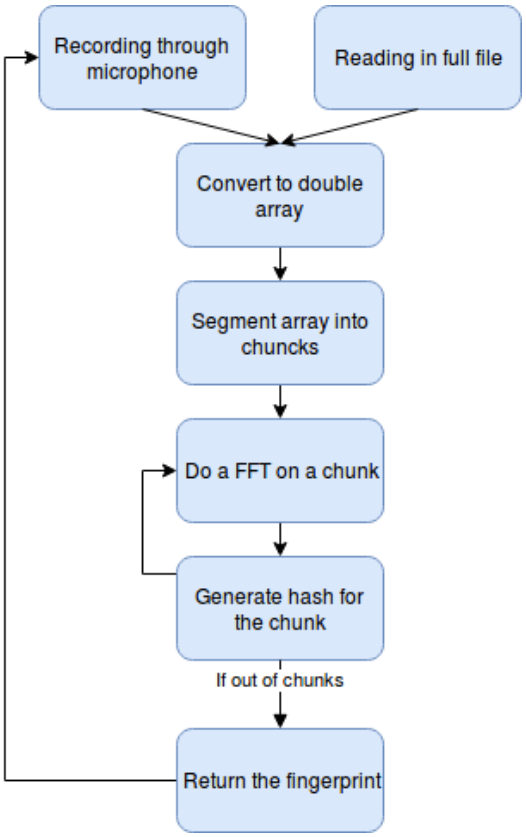

Figure 2: Diagram of the fingerprinting process, done for database management and sound recognition

2.5 Database querying and design

A database needs to be designed which takes all the information we might want for a song and stores it into individual fields which make lookups easy and efficient. The database will not be populated by users using the program, but will be pre-populated by the administrator using a converter which takes songs in mp3 format and extracts their fingerprints, song titles, contributing artists, duration and album art if it is available.

The fingerprint generated by a recording will be used to search through all the entries in the database, looking for substring matches in each fingerprint. Searches will be conducted multiple times, each time adding information to the search about which entries in the database are still viable and which can be eliminated outright. By keeping track of valid entries per search, the algorithm should get faster and faster for every second that goes by in a recording.

2.5.1 Database schema

For the program prototype only one table will be used. This is to simplify updates and searches and the information will not take up much more disk space on the small scale with less than 500 tracks. Once the program is shown to work, the database will be optimized and redesigned.

The schema has the following columns:

1. idmusic_table INT. This column is used as the primary key and shall help in returning a result by just doing a select.
2. music_hash MEDIUMTEXT. This column contains the fingerprint of the entire song, stored as one long array. This column will have searches done on its contents, looking for substring matches within the length of the fingerprint.
3. artist VARCHAR(45). This column contains the name of the artist relating to a specific song and fingerprint. Values in this column will be used to display meaningful results upon a successful search.
4. song_title VARCHAR(45). This column contains the title of the song relating to the generated fingerprint. Values in this column will be used to display meaningful results upon a successful search.
5. duration INTEGER. This column stores the length of each song in seconds. Values in this column will be used to display meaningful results upon a successful search. Values in this column may also be used to limit results in a search.
6. album_art BLOB. This column is used to store the album art associated with each song. Values in this column will be used to display meaningful results upon a successful search.

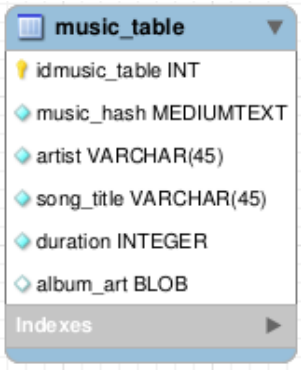

Figure 3: Schema for the database

The database needs to be linked to the Java program and to do this the JDBC library is used [3]. JDBC is a library developed by MySQL which assists in connecting a Java program to a MySQL database. The code which is used to connect to a database can be found online[5].

2.5.2 Queries and searching

In order to query the database efficiently, both the fingerprinting system and the searching algorithm need to be as fast as possible when creating the fingerprint and using it to find matches in the database. For the fingerprinting system optimization is fairly simple as the main time cost comes with the Fourier analysis which can be sped up by using a FFT instead of a Discrete Fourier Transform (DFT). Optimizing the database searching is slightly trickier. If the program were to search through the database every time, limiting the number of results it would take far too long. A better approach is to return many results from the database into the program, search through them, determine whether the current fingerprint is located in the first batch of returned results and if not, repeat the process with another set of results.

The main benefit of doing searching this way is that the program can quickly identify which songs contain the first substring and store them in a data object with key and value pairs where the key would be the primary key of the database and the value would be the fingerprint. A suitable object for this task would be a Hash Map. Once an initial pass has been completed through the database there should be a far reduced number of items to search through which should speed up subsequent searches. Based on the previous statement it is safe to assume that the searches should subsequently get faster and faster the longer the program runs.

Doing the search on the data efficiently is going to be a very important part of this program and because the search is going to be for substrings and not identical matches, standard matching algorithms will not work efficiently. Some pattern matching algorithms include doing the brute force method `String.contains()`, the **Aho-Corasick** algorithm, the **Boyer Moore** algorithm or the **Rabin-Karp** algorithm. The time complexity of each of these algorithms are:

- `String.contains()`: $\mathcal{O}(nm)$
- Aho-Corasick: $\mathcal{O}(n+m)$
- Boyer-Moore: $\mathcal{O}(nm)$ in the worst case and $\Omega(n/m)$ in the best case
- Rabbin-Karp: average case is $\Theta(n+m)$, worst case is $\Theta(n-m)m)$

From the above time complexity analysis it is clear that `String.contains()` is the worst of the pattern matching algorithms and so can be excluded, however it is not so easy to exclude any of the other algorithms as they all have relatively similar time complexities. The only way to decide which algorithm to use is to check the use case of each against the problem at hand.

The search algorithm needs to be able to search through a large set of strings, looking for exact matches with the current fingerprint. The list of exact matches will then be searched through again repeatedly until only one song remains. Aho-Corasick's algorithm works by having a dictionary of words which it wants to match to a large text and count how many instances of each entry in the dictionary occur. This searching algorithm is best when there are multiple items being searched for. This does not fit the needs of this program, therefor Aho-Corasick can be removed. This leaves us with Rabin-Karp and Boyer-Moore.

Rabbin-Karp is similar to Aho-Corasick in that it also searches for instances of multiple words contained in a dictionary and fits the same use profile as Aho-Corasick, meaning we can eliminate Rabin-Karp's algorithm. This leaves the Boyer-Moore algorithm.

Boyer-Moore is the best algorithm for this program as it specialises in finding substrings where the pattern is much shorter than the text in which it persists and this fits the use case for this project[6].

Boyer-Moore's algorithm works back to front and eliminates the need to search through an entire string by knowing that it can skip back by the length of the string if it does not find the last character. This makes Boyer-Moor a very efficient algorithm and its implementation should work well for this project. Bellow is the pseudo code of the Boyer-Moore algorithm[7] which will be used in this project.

Algorithm 1 Psuedocode for the Boyer-Moore algorithm

```
1:  procedure COMPUTE FUNCTION LAST
2:  i ← m − 1
3:  j ← m − 1
4:  Repeat:
5:      if P[j] = T[i] then
6:          if j = 0 then return i
7:          else
8:              i ← i − 1
9:              j ← j − 1
10:     else
11:         i ← i + m − min(j, 1 + last[T[i]]
12:         j ← m − 1
13: until i > n − 1
14: return "no match"
```

During the execution of the searching algorithm, a list of results matching the current recordings fingerprint will be stored in a data structure. This data structure will keep updating until only one result is returned and once that happens, the previous set of matches will be displayed to the user as similar matches. If there are no similar matches in the database, the user shall be notified.

Table 4: The messaging directory for the searching component

Action	Req	Description	Args	Return
search	S1.1	searches for matches	fingerprint	index
dbReturn	S1.2	returns database info	None	list
boyerMoore	S1.1	performs Boyer-Moore	fingerprint, hash	true / false

2.6 Interface and program feedback

By this point in the program, it is expected that there should be one correct result and several similar results returned from the database.

Once the results have been thinned out enough, the user shall get a visual feedback showing the exact match of the fingerprint to a song. Similar matches shall be displayed in a separate panel to the suspected matches. The user shall be able to see the song name and artists of all returned matches and similar/ suspected matches (I2.4) The user shall also be able to see the album artwork of the matched song provided that it is present in the database. If there is no album artwork associated with the returned song, a default image shall appear stating "The album artwork is unavailable at this time".

If there is more than one suspected match, the user shall be able to select the correct song from the list. If the user chooses not to select a song from the list of suspected matches no song shall be selected as a correct match. If there is only one suspected match, it shall automatically be selected as the correct match. Correct matches shall be saved and displayed on the user interface, in order of most to least recent (I2.2).

The figure below shows the wire frame of the user interface. The interface shall use various Java swing components in its design, including animations in the processing of microphone information to give users a visual feedback that the program is working to try identify the song. If the song is not present in the database, a message is displayed saying "No results were found, please try again. If you receive this message more than once it may be because the song is not in our database."

The interface shall have two screens with the following components:

- Screen one: Home screen

    1. A search button which allows the user to record a short piece of music. Once the search button has been pressed the text shall toggle from "Search" to "Stop". If the user chooses to stop the search, no results shall be returned.
    2. A previous matches section which shall display music pieces which were previously matched by the application. This information will be stored locally and ordered from most recent to least recent. Only the last five matches shall be stored.

3. A loading animation which will let the user know that the program is working on finding the match.

- Screen two: Results screen

1. A center panel which displays the returned results of a successful query. This component shall also be used to display error messages and instruction to the user in cases where there were no matches or too many matches.
2. A similar matches component shall display a list of matches with similar fingerprints. If there are no similar matches, the message "This song is unique!" shall be displayed instead.
3. There will be a back button component which allows the user to go back to the home screen and do another search with.

Table 5: The messaging directory for the user interface component

Action	Req	Description	Args	Return
prevSearch	I2.3	Displays previous matches	list of prev matches	None
similarMatch	I2.4	Displays similar matches	List of similar songs	None
animate	I4.	Animates search	Comonent	None
components	I1.	GUI elements	match, pmatch	None
reportError	I3.	Error reporting	Error code	Error message

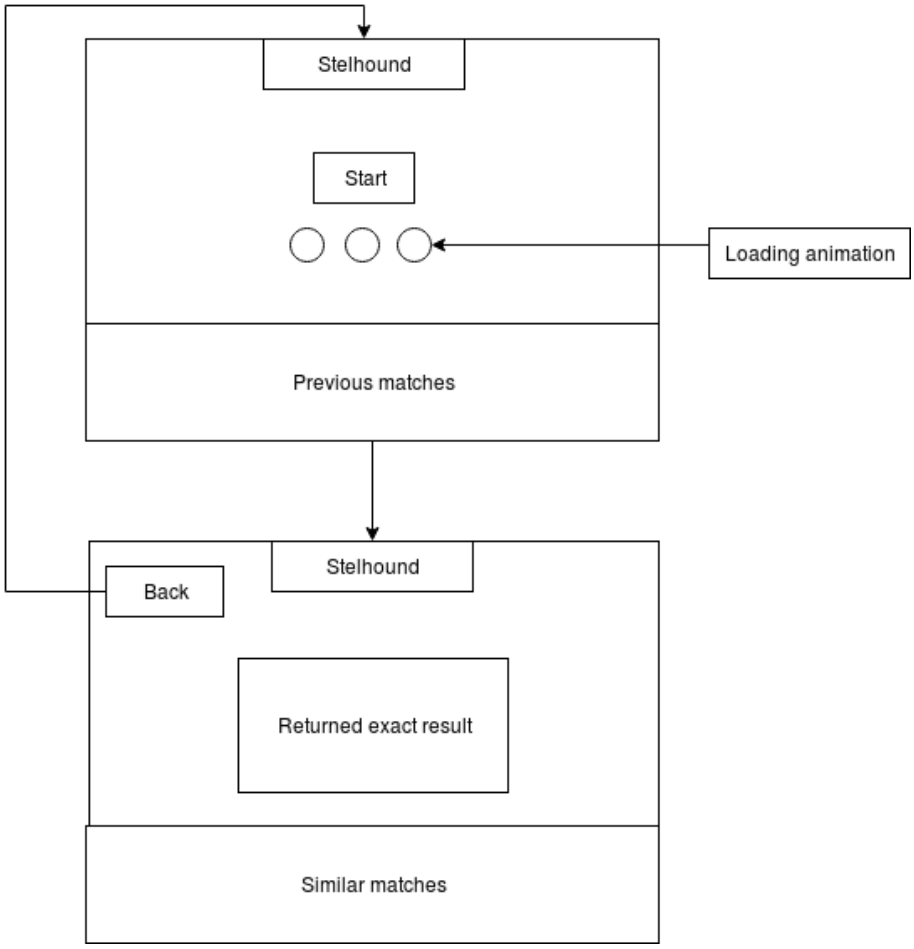

Figure 4: Wireframe for the user interface

3 External library integration

This project will make use of three different external libraries and an independent Java file, all of which are pivotal in the implementation of this project.

The initial process of converting a microphone input into frequency data requires the use of a Fast Fourier Transform which a well documented function with many equally valid interpretations. This project has very little to do with the implementation of the FFT and so an external Java file is used from Columbia University [1].

In order to add music and track data to the database the mp3agic library

[2] is used to extract information about each song. Mp3agic is a Java library used to read and manipulate ID3 tags in songs and this library will help speed up database population immensely.

In order to connect to a MySQL database through Java a connector is needed. There is a freely available library created by MySQL which assists in this and which shall be used in the project to insert into and query the database[3].

The last library included in the project is the SWT library from eclipse[4]. This library will be used to help design the user interface. SWT provides more functionality and cleaner code then the standard Swing containers normally used in Java.

References

[1] Columbia University. "FFT.java Source File". *http://www.ee.columbia.edu/ ronw/code/MEAPsoft/doc/html/FFT_8java-source.html* [Online; accessed April 2017]

[2] Patricios, M "mp3agic". *https://github.com/mpatric/mp3agic* [Online; accessed April 2017]

[3] MySQL "Download Connector/J" *https://dev.mysql.com/downloads/connector/j/5.1.html* [Online; accessed April 2017]

[4] eclipse "SWT: The Standard Widget Toolkit" *https://www.eclipse.org/swt/* [Online; accessed April 2017]

[5] StackOverflow "Connect Java to a MySQL database" *http://stackoverflow.com/questions/2839321/connect-java-to-a-mysql-database* [Online; accessed April 2017]

[6] Boyer, S. 1977, "A Fast String Searching Algorithm". *Communications of the ACM*, Volume 20, Issue 10, Pages 762-772

[7] Roahid Bin Muhammed "Boyer-Moore Algorithm" *http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm* [Online; accessed April 2017]

[8] Cardoso, J. 2003."Blind Signal Separation: statistical Principles" *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.462.9738&rep=rep1&type=pdf*

[9] Roy van Rijn, "Creating Shazam in Java" *http://royvanrijn.com/blog/2010/06/creating-shazam-in-java/* [Online; accessed April 2017]