VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum-590018, Karnataka

BANGALORE INSTITUTE OF TECHNOLOGY

K.R. Road, V.V.Puram, Bangalore-560 004

Department of Computer Science & Engineering

Dissertation on

Tra La La- Music Search

Submitted By:

USN	Name
1BI09CS027	**Dhanush T.D**
1BI09CS051	**Namitha Sindhu**
1BI09CS127	**Priyanka Manjunath**
1BI09CS128	**Alok Rao P**

for the academic year 2012-2013

Under the Guidance of

Prof. N. Thanuja

Asst. Professor

Department of Computer Science & Engineering

Bangalore Institute of Technology

Bangalore-560004

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum-590018, Karnataka

BANGALORE INSTITUTE OF TECHNOLOGY

Bangalore-560 004

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Certificate

This is to certify that the Project work entitled **"Tra La La – Music Search"** has been successfully completed by

USN	Name
1BI09CS027	**Dhanush T.D**
1BI09CS051	**Namitha Sindhu**
1BI09CS127	**Priyanka Manjunath**
1BI09CS128	**Alok Rao P**

students of VIII semester B.E. for the partial fulfillment of the requirements for the Bachelors Degree in Computer Science & Engineering of the **VISVESVARAYA TECHNOLOGICAL UNIVERSITY** during the academic year 2012-2013.

Prof. N. Thanuja
Internal Guide
Asst. Professor
Department of CS&E
Bangalore Institute of Technology
Bangalore - 560004.

Dr. S. NANDAGOPALAN
Professor and Head
Dept. of CS & E
Bangalore Institute of Technology
K. R. Road, V. V. Puram
Bangalore - 560004.

Examiners: 1.

2.

“Dedicated to our beloved Parents and Friends”

ACKNOWLEDGMENT

There have been a lot of people whose support made this project to be completed successfully and it is only appropriate to express our gratitude to them all.

Our first thanks goes to the University. We are grateful to the **Visveswaraya Technological University** for providing us with this opportunity to showcase our skillsets in the form of this project.

We thank **Dr. K. R. Suresh, Principal, BIT,** for providing us with excellent facilities and bringing us all together.

We thank **Dr. S. Nandagopalan, HOD, Department of Computer Science & Engineering,** for his valued co-operation in completion of this project.

We are thankful to our project coordinator, **Prof. S. K. Savitha**, and the entire **Department of Computer Science & Engineering,** for constant support and advice throughout the course of preparation of this project and the report.

Our heartfelt gratitude to our guide, **Prof. N. Thanuja**, **Department of Computer Science & Engineering,** for her constant support, advice and motivation. Thanks for being there for us.

We also sincerely thank **Roy Van Rijn** as his paper on finger-print matching helped us get a more granulated idea of our design.

Last, but not the least, we are deeply indebted to our beloved parents who stood by us at all times with their constant motivation and unflinching support. We also thank our friends for their constructive criticism and assistance.

Dhanush T.D

Namitha Sindhu

Priyanka Manjunath

Alok Rao P

ABSTRACT

In a society where everybody wants the world at their fingertips, instant access of data is one of the biggest design challenges for a software engineer. There is a lot of work done on text processing and searching but, when it comes to different mediums the domains remain quite new. Even today Google image search gets better result with the text string attached. The scenario is no different for other multimedia. When it comes to songs the only way to search for it on the internet is with its name. Hence we see that all forms of search in the end go back to text processing and string matching. We see quite clearly that there exists a need for an application that can search for a song taking the tune as its input.

It's noticeable though, matching a song by its actual song data presents its own set of disadvantages. First, there is the encoding that goes into most song formats then there is the obstacles of too much information the sound samples need to be mined and converted to give us useful characteristics that could be used to match against it. Then comes the complexity that could be associated with such a match. An algorithm which claims to perform song matching must surpass or solve these obstacles and problems. This is what we propose to do in this project.

This application will take a microphone input from the user and process the sound samples into wave forms. These waveforms are matched to, perceivably, a database on the server side or a library on the client side. The two big ideas which we have implemented are 'conversion of bare byte arrays of the input sound sample to spectrograms'. These spectrograms are a series of values which denote the three main factors that determine the uniqueness of any note in a song. The second idea is 'the pattern matching algorithm' whose complexity was got down to constant time for each note.

These two concepts have been integrated with a fully working music player for the ready-made database that it will provide us with. The music player also enhances the usability of such an application as a 'play and match' utility can be thought of as a natural add-on or an extension for a music player. This music application will be installable that can be run on any desktop computer.

The main motivation here is that the characteristic of a song presents us with more patterns that are distinctive than a speech recognition system. As the saying goes 'a song in its essence is universal'.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

Chapter 1

Introduction

INTRODUCTION

Twelve notes. Five sharp, and seven flat. Ten octaves and Infinite possibilities.

Of all the different things man has invented for his pleasure and peace, nothing is as sweet as music. It is said that music is as close as we can get to nirvana (except for one other). Other trends may come and go, but music lives on. Man learnt to sing before he learnt to talk, and he isn't likely to forget it anytime soon. Music, keeping up with the changes in time, has constantly reinvented, refined (and for a brief painful period remixed) itself.

1.1 Prelude

The first development was the creation of a language for music, giving it a means to live on. Creating notes so that it could be represented in more than just voice, and writing it down in more than just lyrics. Thus was invented sheet music. The motive behind this being the need to capture a good song and maybe replicate it again, just as we wanted to capture the great events of history in text and learn from it. There was, however, a downside. Paper alone, although it had its uses, had limitations. The tune was saved, yes, but it could not be listened to at leisure, quite contradictory to, say a storybook. We needed some way to store a song. To store it and play it.

Many, many years later we had vinyl records and magnetic tapes. Songs were sung, and recorded. Packaged and sold. And boy did they sell. The demand was so much that the total music industry in the world today is worth in hundreds of trillions. The number of songs composed and recorded are innumerable. This demand catapulted when music, as every other medium these days, left magnetic tapes for digital drives. (Which are also magnetic. How anti-climactic). This meant only one thing. Yes. Computers. Which also meant the World Wide Web. Which brought in its wake the grand era of music piracy, incidentally, and was the first instance of illegal activity that cropped up on the web (after the honorable Mr. Virus of course).

Today, 16% of the world's websites deal in music, which is the second highest share for any type of multimedia on the web (we will not go into what is first). With all

this music at a user's disposal comes in the happy problem of too much to handle. Huge amounts of data need structure and accessibility. That, exactly is where our app comes in.

1.2 The need for a better music search engine

Just as we are branded with names and USN's to tell us apart from the other thousands in the brood, so do we brand everything we create. We brand a song with a name, put these songs in an album, plaster the name of the man who sung it, the man who composed it, the company which taped it and released it. All these details we ship along with a song, and these are the only details which we have to identify a song among thousands of its own kind. Or are they?

In a world where we are making computers more and more clever, why can't we consider the possibility of a search engine which surpasses the limitation of asking for a text string to identify a song? Every time a user wants to search for a particular song, he needs to type in its name. This makes it no different from a text search. Which is a pity, as music offers so many other possibilities. Untapped and waiting to be put to use.

Our application aims at bringing justice to one of these. We propose to make an application, which takes in a microphone input of a song being played, and instantly searches for it from a database. No (text) strings attached.

In the next few pages, we wish to elaborate on how we went through the process of taking this seed of an idea and creating a music application which does just that. Before going into the details of it, however, we need to understand sound itself, and how it is stored in a computer system, what parts of a sound wave constitute its uniqueness, and how to mark these parts. This will be the subject of the rest of this topic

1.3 Sound

Sound is just a pressure wave that our mouths put out into the surroundings by compressing the air right next to our mouth. As a reflex, the air compresses the particles next to it in turn, and relaxes. These compressions and relaxations propagate all the way to the recipient and impinge upon their ears.

The two constituent components of a sound wave are called compressions and rarefactions. The amount of pressure given for a compression is the amplitude of that certain wave. The distance for a full wave is one wavelength, and the amount of such waves generated in one second is its frequency.

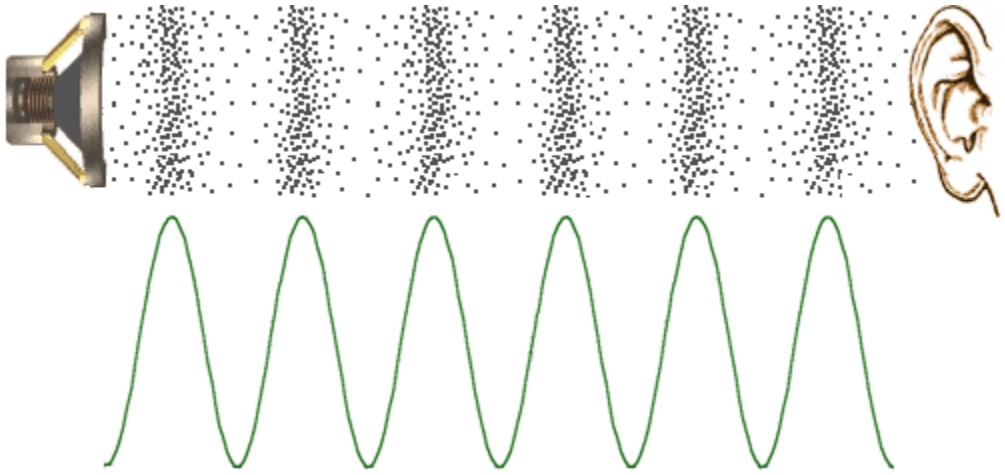

Fig. 1.1 Sound Transmission from a source to a reciever

In reality, the sound waves look much more jarred and incomplete.

The important observation that we need to take away from this is that three things uniquely determine any distinct point in a sound wave.

- Its amplitude or volume.
- Its frequency or pitch.
- Its time slice.

These three things are what will determine the uniqueness of a note at a point of time in a song. This is what we hope to draw out of a song file and use to compare the song with other samples that have to be searched on. For that we first need to take a look at what a song stored on disk looks like, what is the information we can directly get from it? What information requires manipulation? What are the time and space complexities of this manipulation? We begin to answer these questions by first looking at a sound wave in an electronic system.

1.4 Sound Samples

Now we need to look at how sound is stored in a digital format. If we dig deep into a song file and plot its numbers into a graph, we see something like this:

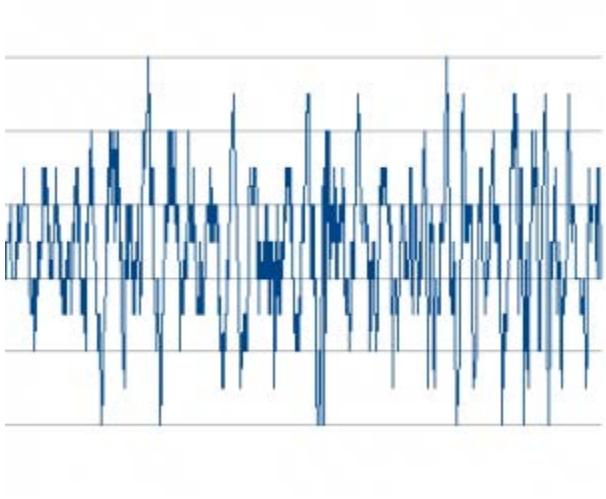

Fig 1.2 Sound Samples

These obviously make no sense by themselves. To understand this, we need to understand what sampling is. What exactly does it mean when they say a song has a sample rate of 8000Hz?

Sampling is basically taking measurements of the magnitude of the sound at different intervals of time. If it says its 8000Hz, it means per second, 8000 measurements are taken of the magnitude. It implies that more measurements increase the quality, which is true.

These, measurements therefore constitute a song. Our job is to take this data and convert it to get the three significant data points that we discussed in the previous section, namely time-amplitude-frequency. Before we do that, we have another hurdle. Song files are not simply these sound samples. They have a slightly different structure.

The structure varies from format to format as different formats use different encodings. Of the different formats that exist, one format which is the most popular today is the mp3 format.

1.5 MP3

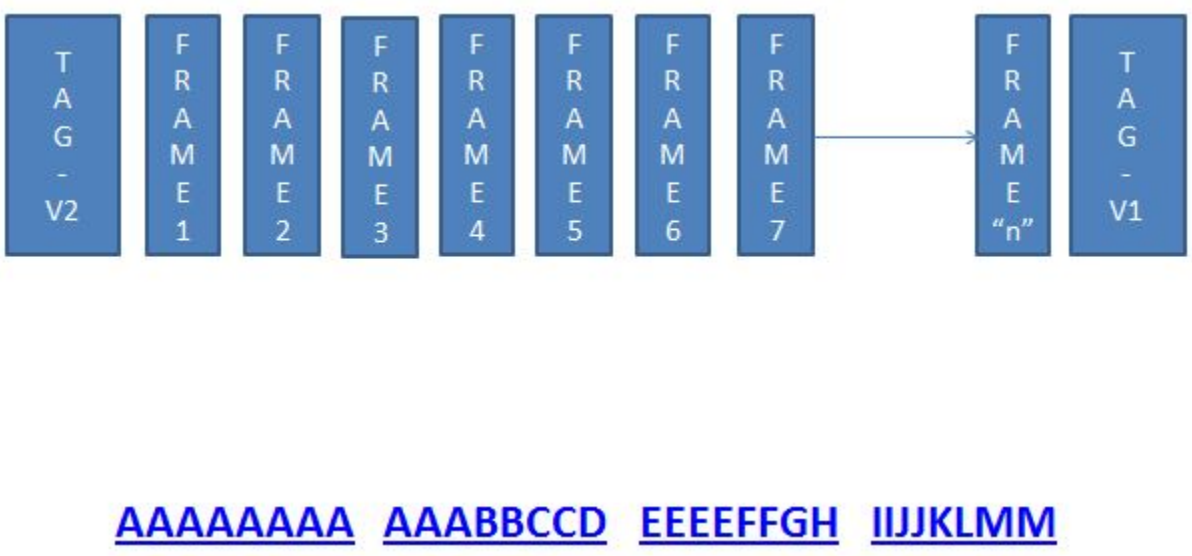

Fig 1.3 Structure of an mp3 file

This is the structure of an mp3. It is enclosed by tags or metadata. After them come the frames. Each frame has a frame header. The header gives details about each frame. After each header comes the song data. To extract the sound samples discussed earlier, we need to:

- Get access to the bare bytes of an mp3.
- Check if tags exist, and if they do, skip over them.
- Skip over the frame header and copy the actual data.
- Get the details of the frame header and apply the necessary decoding technique.
- Join all the decoded numbers to form a set of samples.

This procedure will get us the raw byte array.

It seems that we have been going more and more backwards in explaining the process which we intend to undertake. Now that we have a brief view of the details, let's look at the procedure step by step.

- Get the song, decode it.
- Use the samples to get the required values of the song.
- Convert these samples into the constituent data points that determine each note in the song.
- Save this information for each song in the database.
- Do the same conversion for a microphone input from the user.
- Match with these stored data points to determine which song is the one played.

With this introduction, let us look at some related work done on this topic.

Chapter 2

Literature Survey

LITERATURE SURVEY

After doing some research it was found that there were a few systems which used acoustic fingerprint matching to match sound. An acoustic fingerprint is a condensed digital summary, deterministically generated from an audio signal that can be used to identify an audio sample or quickly locate such similar items in an audio database.

Practical uses of acoustic fingerprinting include identifying songs , melodies, tunes and video file identification. Media identification using acoustic fingerprints can be used to monitor the use of specific musical works and performances on radio broadcasts, records ,Cd's and peer to peer networks. This identification has been used in copyright compliance, licensing, and other monetization schemes. The most notable applications were Shazam, SoundHound(Midomi), Gracenote by musipedia, MusicID, and Play by yahoo.

2.1 Shazam

Shazam is a commercial mobile phone based music identification service, with its headquarters in London,England. The company was founded in 1999 by Chris Barton, Philip Inghelbrecht, Avery Wang and Dhiraj Mukherjee.

The service was launched initially just in the UK in 2002 and was known as 2580, since customers dialed the short code from their mobile phone to get music recognized. The phone would automatically hang up after 30 seconds. A result was then sent to the user in the form of a text message containing the song title and artist name. At a later date, the service also began to add hyperlinks in the text message to allow the user to download the song online.

Shazam launched in the US on the AT&T Wireless network in 2004 in a joint offering with Musicphone, a now defunct San Francisco based company. The company has seen exponential growth since its inceptions due to its unique theme and growing user base that has given a boost to its development. This by far is the best in the domain at the moment.

Shazam uses a mobile phone's built-in microphone to gather a brief sample of music being played. An acoustic fingerprint is created based on the sample, and is compared against a central database for a match. If a match is found, information such as the artist, song title, and

album are relayed back to the user. Relevant links to services such as iTunes, Youtube , Spotify or Zune are incorporated into some implementations of Shazam.

As of September 2012, Shazam has raised $32 million in funding.

2.2 Audio Fingerprinting

Another paper that we came across describes the development of an audio fingerprint called AudioDNA designed to be robust against several distortions including those related to radio broadcasting. A complete system, covering also a fast and efficient method for comparing observed fingerprints against a huge database with reference fingerprints is described. The promising results achieved with the first prototype system observing music titles as well as commercials are presented.

In a general Audio Fingerprinting scheme, the system generates a unique fingerprint of audio material based on an analysis of the acoustic properties of the audio itself. The fingerprint is compared against a database of fingerprints to identify a song.

Fingerprinting, or content-based identification (CBID), technologies work by extracting acoustic relevant characteristics of a piece of audio content and storing them in a database. When presented with an unidentified piece of audio content, characteristics of that piece are calculated and matched against those stored in the database. Using complex matching algorithms and acoustic fingerprints different versions of a single recording can be identified as the same music title.

A figure outlining the process of audio finger printing is depicted clearly in the figure below. The main blocks of the process are what has just been explained. The implementation details vary from person to person. This is just the outline of how things happen when an audio sample is converted and optimized and the music info is extracted from it. A much deeper explanation of the same process will be given in the implementation part when we get down to doing the fingerprinting ourselves.

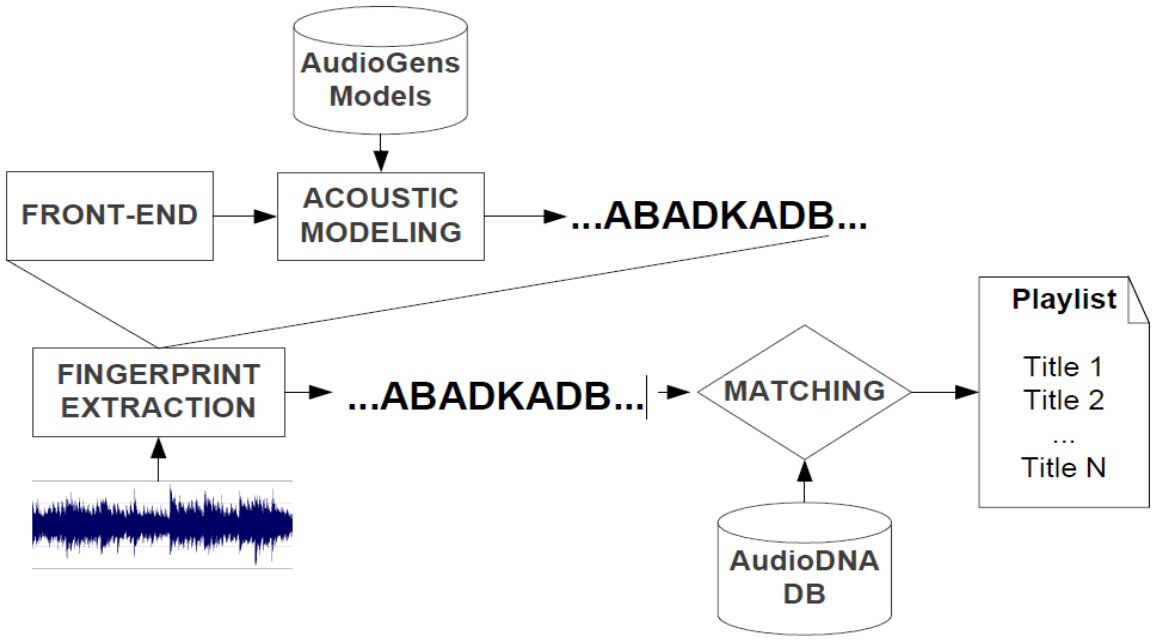

Fig 2.1 Audio Finger Printing

2.3 Audio Genes

AudioGenes have additional time information, which is a significant difference to standard string applications, but this information can be exploited in the matching algorithm. The above diagram shows the most important cases beside the identical match, which may occur when comparing observed with original AudioDNA fingerprints:

- identical genes are detected, but their time borders are different
- parts having different genes
- additional genes are present in the same time interval
- the average length of genes is different (typical case with pitching effects)

A graph of the same is seen in the next page. The audio version is the y-axis and the time is x-axis.

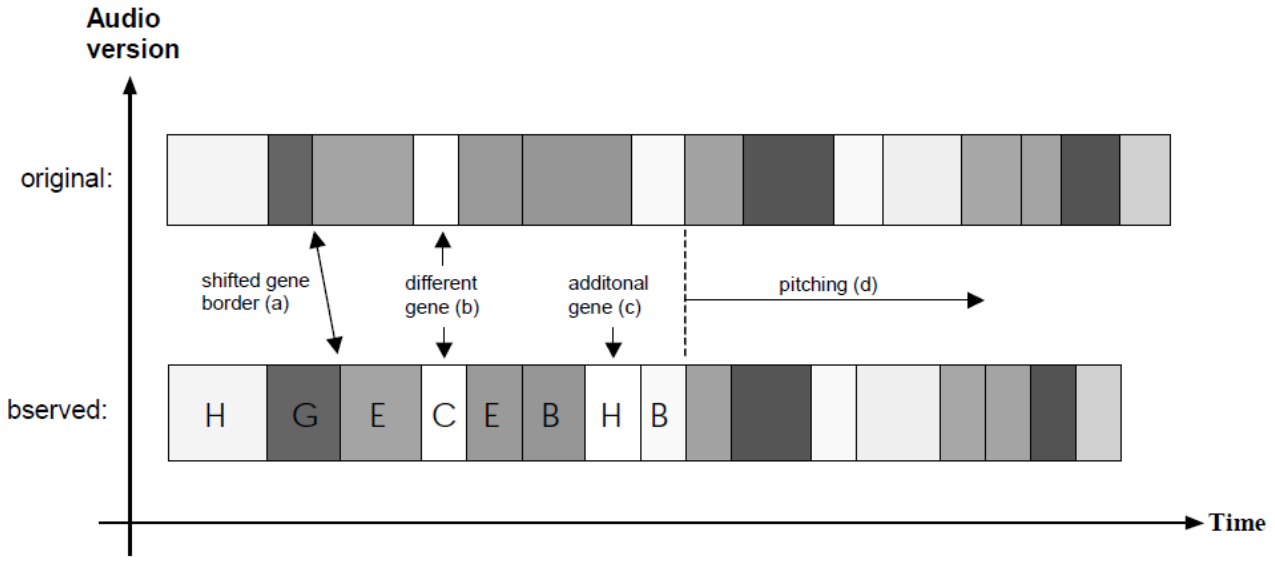

Fig 2.2 Audio Gene Graph

2.4 Matching Process

Short subsequences of AudioDNA from an observed audio stream are continuously extracted and compared with the fingerprints in the database. If an identical subsequence is found, the matching result will be stored in a balanced tree data structure for further processing steps. One node of this tree contains a list of exact matching results of one title. Only results appearing in the right chronological order will be appended to a node.

In addition the time length information is used to discard non-promising nodes. If a node contains a certain amount of exact matching results, an approximate matching method is applied to detect similarities of longer sequences starting at the position of the exact matches. The calculation of the similarity of sequences is a standard application of dynamic programming. Again the embedded time information of audio genes is used to apply a very fast and simple approximate matching method.

2.5 SoundHound

SoundHound (known as Midomi until December 2009) is a mobile phone service that allows users to identify music by playing a recorded track. The service was launched by Melodis Corporation (now SoundHound Inc), under chief Executive Director Keyvan Mohajer in the year 2007.

Sound matching is achieved through the company's 'Sound2Sound' technology. The app then returns the lyrics, links to videos on YouTube, links to iTunes, ringtones, the ability to launch Pandora radio as well as recommendations for other songs. A feature called LiveLyrics displays a song's lyrics in time with the music, if they are available. Double-tapping on those lyrics moves the music to that point in the song. It is also possible for users to play music from their iPhone's iPod library through the app. If lyrics are available for a song, it will show them as it plays.

Fig 2.3 Sound Hound Snap

The tap here button when pressed enters a mode where the user inputs the song through a microphone(Phone's inbuilt mic) and the artists and the songs are displayed on the right side along with an option to look up lyrics.

Chapter 3

System Requirements

SYSTEM REQUIREMENTS

Below is a table outlining the basic requirements on both sides.

Developer side	Client side
• IDE: Netbeans • JDK(Java Development Kit) • InstallShield(For deployment) • 512MB RAM • 3GB Hard Disk space • Microphone	• OS: Win XP or Higher • Processor: Pentium or Higher • Microphone • 512MB RAM • 500MB Hard Disk space

Table 3.1 Requirements table

These are the main things of importance that will be needed as resources throughout the coding of this project.

Chapter 4
Architecture

ARCHITECTURE

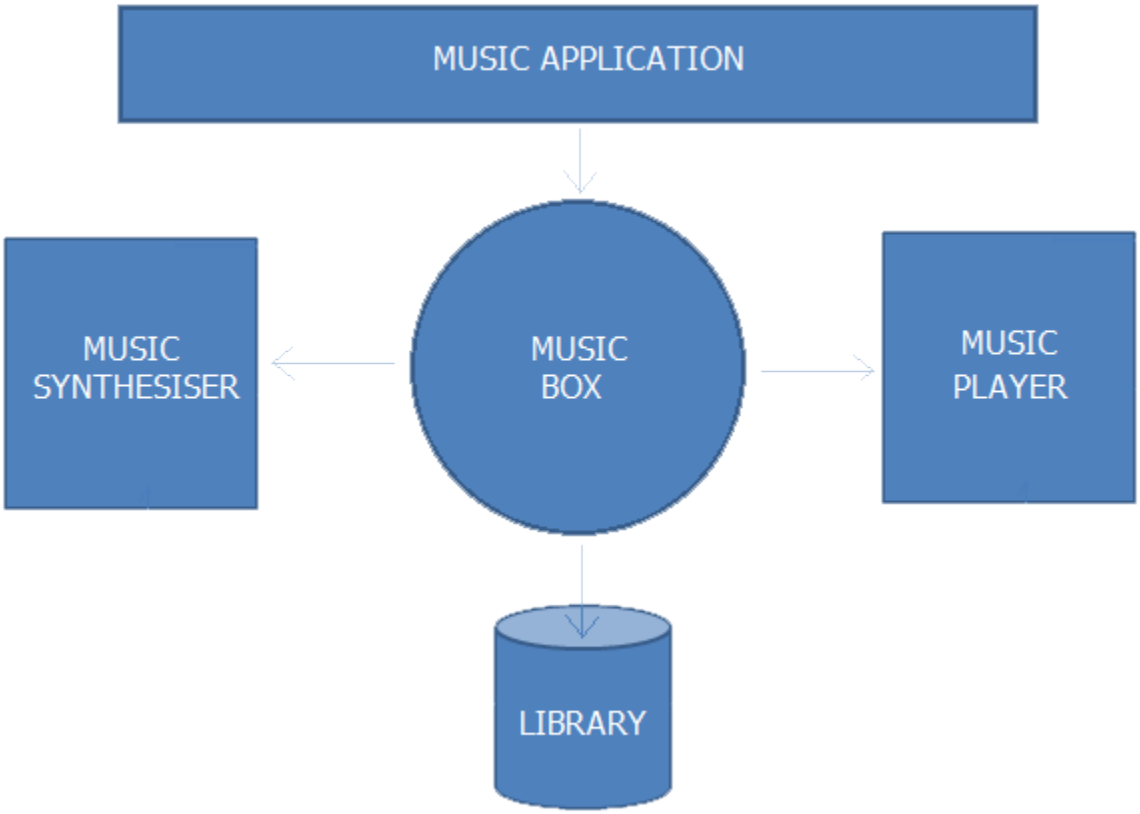

Fig 4.1 High level design

High level design gives an overview of the entire application. The design used can be compared to that of a computer system where MusicApp forms the front end and is analogous to a monitor, synthesizer to a supplier, Player to a processor, MusicBox to a CPU and music library to a Hard disk.

MusicBox is like a central manager which is responsible for sub tasking to the other modules. Each module has been explained in further detail in the lower level design module.

Chapter 5

System Design

SYSTEM DESIGN

Our first decision of course, was to divide the design into modules. For a proper MVC structure, we cut the project up into the following parts.

5.1 Module 1-Music player

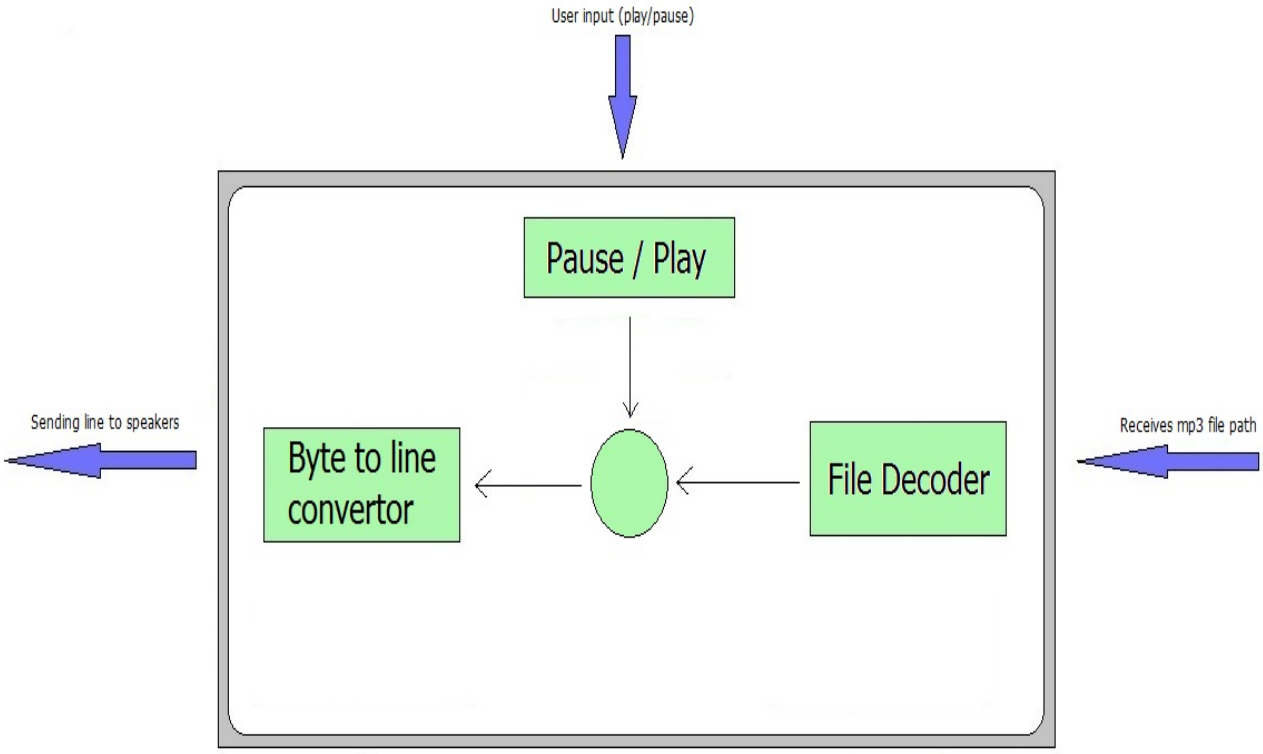

Fig 5.1 Music Player

To be brief Music player takes in a request , gets the requested file and plays the requested song. User requests the music player by pressing a button(front end element) which is either a play or a pause. If the button pressed is play, the song's file path is obtained and sent to a file decoder.

A mp3 file is of no use since we need to use the data stream and accessing the data stream in an mp3 is not possible since it is encoded. Hence the need of a file decoder which decodes an mp3 file, thus enabling us to access the bare byte stream of the sound

file. Playing a song via speakers means putting these byte stream into the output line stream of the speakers which does the required D/A conversion and hence play the song.

5.2 Module 2-Music Library

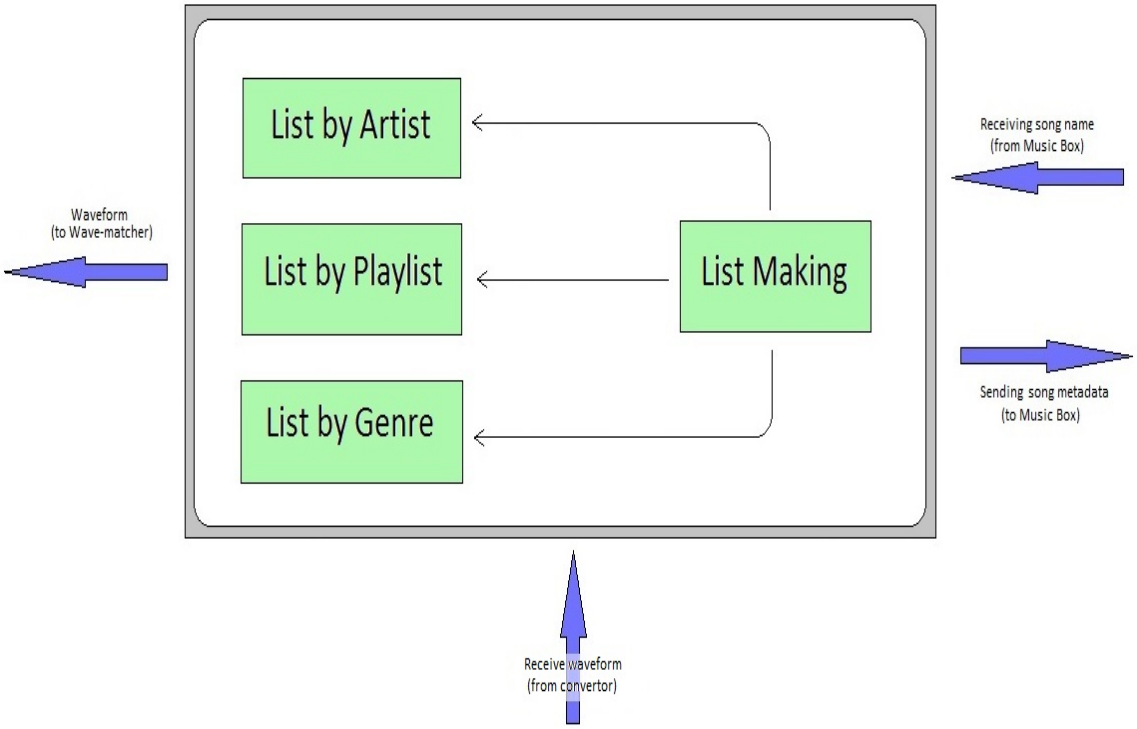

Fig 5.2 Music Library

Music Library forms the database of the Application. It holds all the data of the Song library made by the user. It contains hash maps for the list of Playlists, list of all the albums in the Library, and the list of songs based on the genre.

Each of these lists are made and updated as and when a song or album is added to the library. Music Library responds to the requests made by the musicBox for a song by sending the metadata of the song to the musicBox.

Further every time a song is added, it is sent to the converter where the song data is converted into waveform data and stored in a hash table.

Finally when a user enters the humming mode and enters the input, music library sends the necessary waveform data for comparision.

5.3 Module 3-Music App

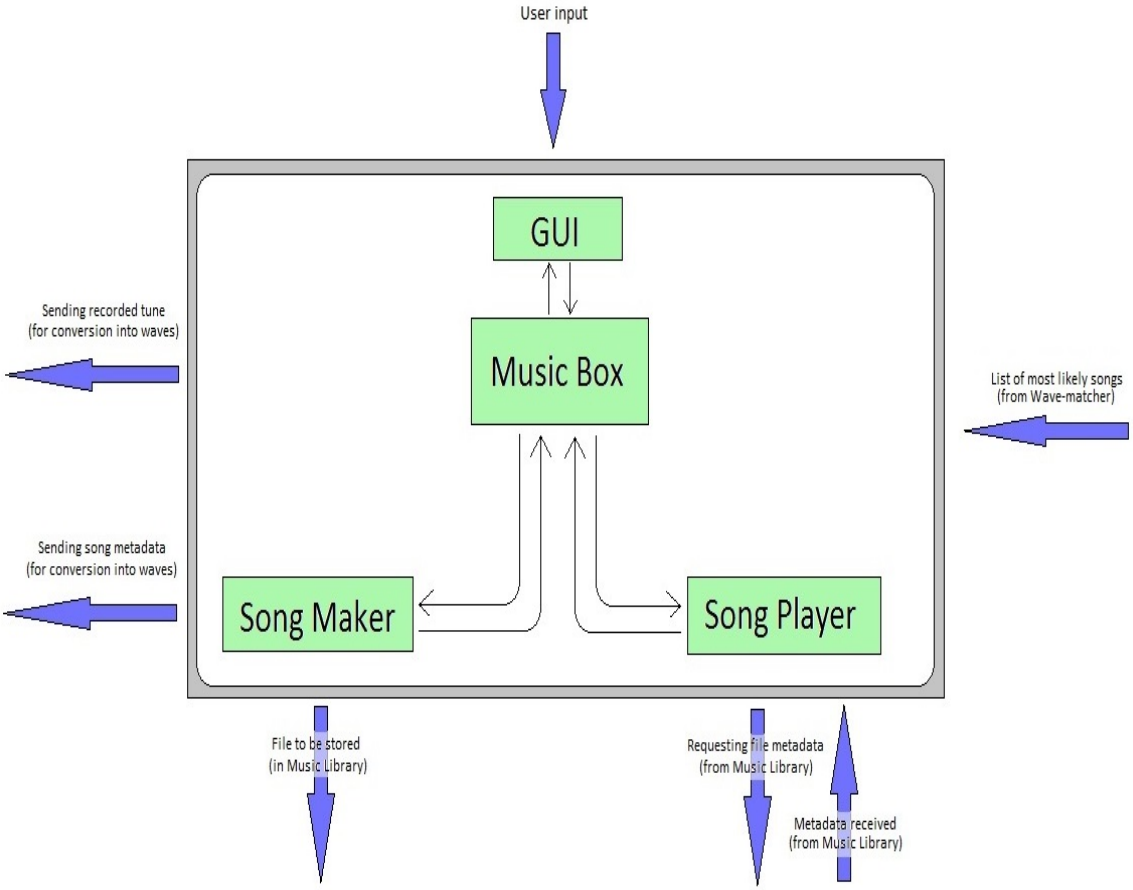

Fig 5.3 Music App

MusicBox is an intermediate class to mediate between all the other components of the app. Any class that has to get any work done outside the scope of itself, will ask the Music Box.

The music box forwards the “addSong” , “findSong”, “getSongMetaData” methods to the library. The library furnishes the Box with all the data. The Box converts it into a plain text form for the GUI class to understand, and forwards the answers to it.

It also has other subsidiary classes to extract the meta data of the song, pass the next song on to the player.

It sends the recorded tune for conversion to the converter and when a play is requested, it sends the song to the player.

MusicBox gets a list of most likely hits from the waveform matcher and sends it to the GUI to display it to the user.

5.4 Module 4-Music synthesizer: converter

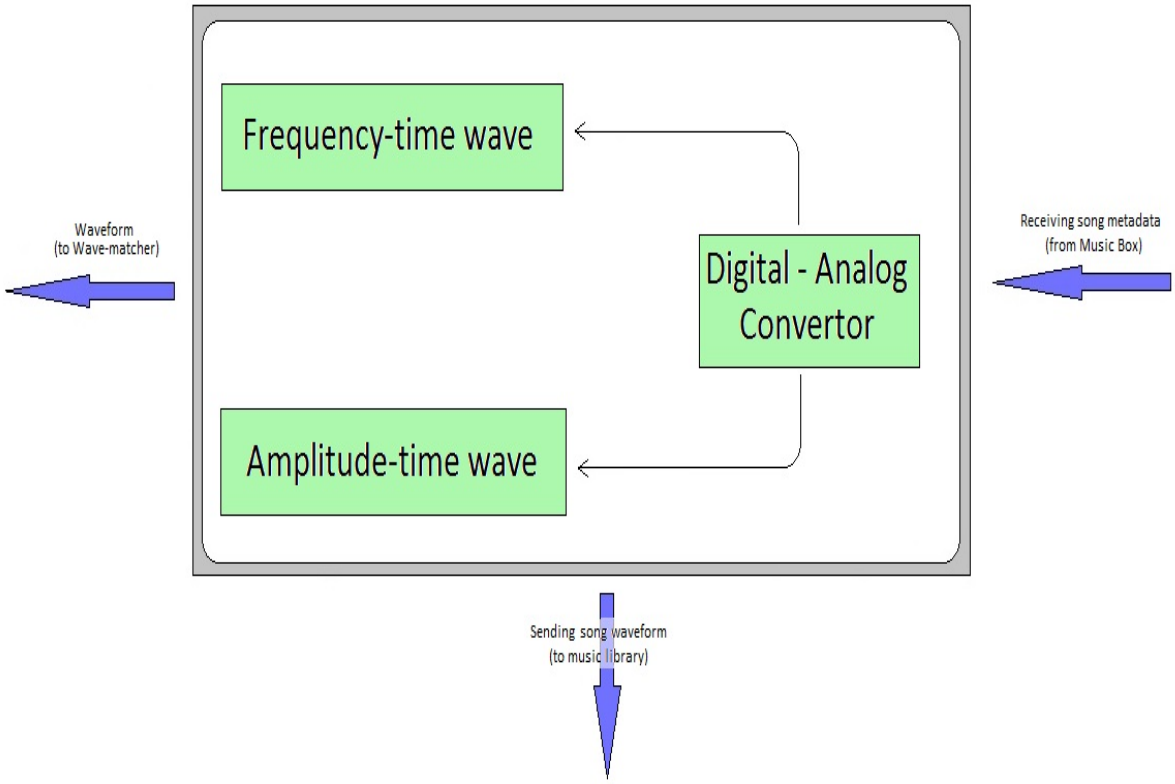

Fig 5.4 Music Synth 1

The job of a converter is to convert the song data into a usable form and store it in a hash table. On the whole it involves a Digital to Analog conversion of the song and storing the song as a frequency time wave and an amplitude-time wave.

It receives the song from MusicBox and sends the song either for comparison to the waveform matcher or simply stored in the library.

5.5 Module 5-Music Synthesizer: Pattern matcher

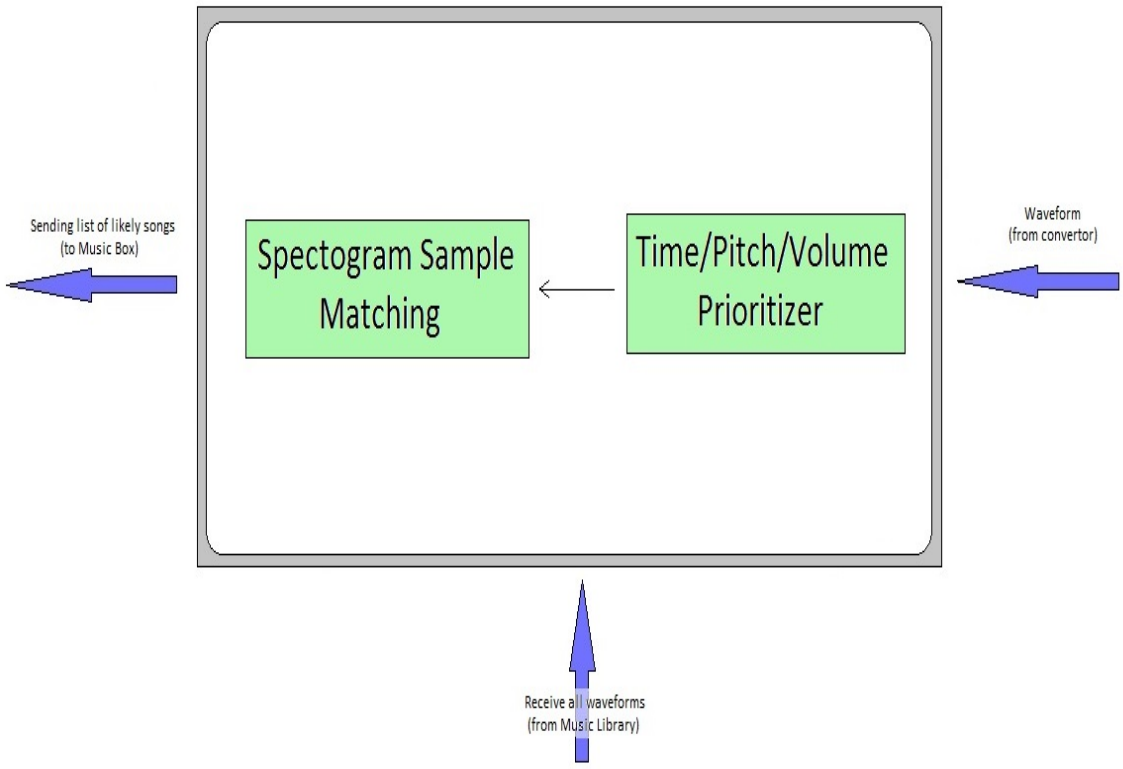

Fig 5.5 Music Synth 2

Pattern matcher does the actual pattern matching. The inputs to this module are the waveform of the input and the list of all the waveforms from the library. It uses a Time/Volume/Pitch prioritizer to create spectrograms and these spectrogram samples are further matched to get a result of most likely hits which is then sent to the MusicBox.

Chapter 6

Implementation

IMPLEMENTATION

Of the 5 modules that we've seen in the design part, we will elaborate on the conversion of the conversion and pattern matching models and briefly push through the music player, music application and the music library modules. We start with :

6.1 Music Player

This was the first obvious choice for a module implementation. The procedure for making our own music player is quite boiler plate. In brief, our procedure for implementing the music player was as follows.

- We had to get a stream of the bytes from an mp3 into our program. For this, we took the path of an mp3, put it in a File object, fed it to a file input stream, and got all the output from it into a byte array.
- We checked for existence of tags, skipped over them. Extracted the first 32 bits of every frame and used its values to decode the mp3.
- The raw decoded data was appended to another byte's array which was going to be a list of sound samples. This list would be nothing more than a standard .wav file.
- The data thus obtained was fed into an audio input stream by specifying the right format. (16 bit stereo with 4 frames per second and big endian.)
- A line was obtained from the speaker to which the audio input stream was fed. With this, we had the barebone player ready.
- Implementing the stop functionality was easy enough as the speaker had a close function.
- For pause and resume, you had to keep track of the position of the filestream when the speaker was closed so that when resuming, we could do so from that position in the filestream.

6.2 Music Library

We needed a database of our own inside the application for the user to store his/her songs in an ordered fashion. We chose hashmap from Java Collections and made the appropriate arrangements to store playlists and albums. The library module also had other getter, setter and modifier methods to access and manipulate its contents.

We implemented serialization so that the users' library could persist beyond the life of the application process in the run. The music library class used helper classes like a SongMaker to build its basic object: Song.

The Song object was an instance of a class we created to hold the data required for a song, namely:

- It's tags, that is, Artist, Album, etc
- The song's length in seconds.
- The path of the song file
- Its CoverArt

6.3 Music Application and Music Box

The Music Application module was designed to take care of the front end and manage the GUI thread. The music application of which our humming mode was going to be a subsidary feature, had to always have a constant GUI thread which had to be responsive to user input at all times. Since we were coding in Java, we used the standard Swings GUI design and awt event listeners. We handled scrolling events with the appropriate timers and spawned threads when necessary.

The Music box was a manager or an intermediatary agent, or class (whatever) which would interact with each module on the other's behalf. For the sake of good OO, we decided to centralize all communication that happens in the Music Application through the Music Box. This meant that the Music Box had to do most of the heavy lifting which included conversions and delegations.

With these three modules together, we had a complete Music Player which was eligible enough to be an independent application by itself. We had the front end and the database ready for our music search engine. The next two modules were going to be the crux of the project.

6.4 Music Synthesizer Part I (Song conversion)

By making the music player, we'd already got a solid procedure to extract a byte array from an mp3. This byte array, as explained in the introduction, was a list of sound samples which now had to be converted into a list or a table of amplitude-frequency-time values. As these samples were just measurements of the magnitude with relation to time, we can safely assume that this is a time domain graph. What we had to do was convert it into a frequency-time domain. We knew that to convert any time domain data into frequency domain, we could use Fourier Transforms. Our requirement, though, was a frequency-time domain. Therefore it was imperative to first fully understand the procedure of Fourier Transforms, and then find a way to tweak this algorithm to serve our purpose.

6.4.1 Fourier Series

Before we understand what 'Fourier Series' actually means, we have to accept one fact: Every wave we can possibly draw is a sum of an infinite number of sine waves and cosine waves with differning amplitudes, and harmonics. Meaning if you take a graph sheet, and scribble on it, your scribble can be represented by a sum of sine waves, and cosine waves. These sine waves have different amplitudes, and different frequenies (i.e. the constituents may be sin(x), sin (2x), sin (3000x) or whatever).

Here's an example:

The two waves shown here are sin(x) and sin(2x) (with equal amplitudes):

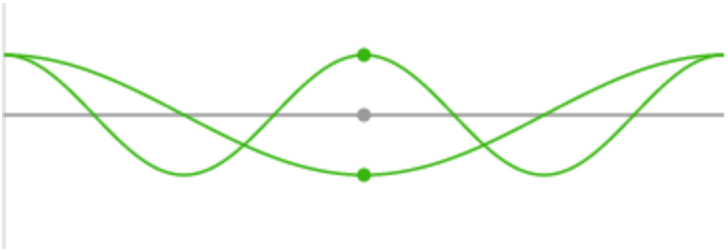

Fig 6.1 Wave Plotting

And the resultant wave is:

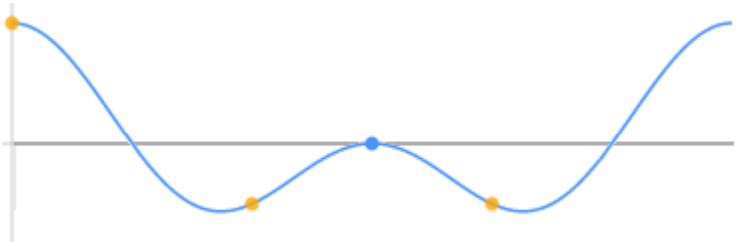

Fig 6.2 Wave Superimposition

But this isn't all, to extend this to a 2-dimensional graph, instead of imagining these scribbles to be broken up as sine, and cosine waves, we can imagine these as broken up into circles. Like this:

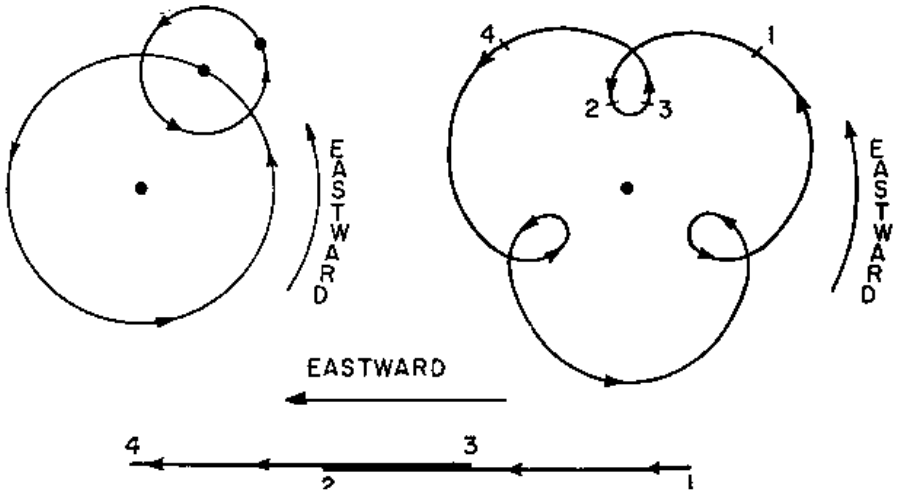

Fig 6.3 Real vs Complex Plane Sine Wave

So we can consider every scribble to be the result of a series of circles. And the resultant is dependent on :

- How fast each circle is spinning (frequency)
- What point each circle started spinning at (phase)
- The radius of each circle (amplitude)

But a point on a circle of radius 1, can be represented by e^{ix}!

And this, in turn, can be represesnted as a wave, cos(x) + isin(x).

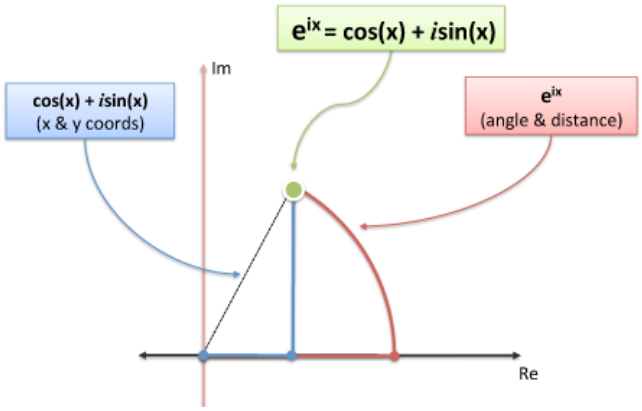

Fig 6.4 Jagged Wave

So Fourier Series of a wave, is the constituent real cosine waves, and imaginary sine waves.

6.4.2 Fourier Transform

This is just a mathematical operation, where you supply a wave, and it's broken into it's constituent waves. Practically speaking, we can measure a wave by sampling, so we have to apply a discrete Fourier Transform, over N samples

$$X_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n e^{i2\pi k \frac{n}{N}}$$

To find the energy at a particular frequency, spin your signal around a circle at that frequency, and average a bunch of points along that path.

You take the amplitude of your wave at the sample n (x_n), and multiply it by the contribution of a wave at frequency k. This is done by considering the constituent circle (a circle is e^{2*pi}, the circular wave spinning at frequency k is e^{2*pi*k}), and measuring it at the current point in time (given by n/N, where n is the current sample, and N is the total number of samples). This gives you the energy of the wave. But this is a complex number, where the amplitude is the real part, and the phase is the imaginary part.

Since there are N samples, the maximum number of frequencies we can have is N, for the best quality. With N frequencies, and N samples, the time complexity is N^2. However, using Fast Fourier Transforms, we can achieve N logN complexity.

In our application, we have 4096 samples, with a microprocessor speed of 3 million instructions per second. So putthing that in perspective, here are the algorithm speeds:

Fourier Transforms: 4096^2 = 16,777,216 = 10 minutes

Fast Fourier Transforms: 12 * 4096 = 49,152 = 20 seconds

6.4.3 Fast Fourier Transforms

It would be easier to view the fourier transform as a square matrix that you're using as an operator on your data fields.

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Here, a_0 through a_{n-1} is a vector containing the amplitudes at time intervals 0 to n-1. w, is another way of saying N^{th} root of unity. So $w^N = 1$. . The row selects the frequency (k), and the coloumn selects the time samples (n). Since the w^N is 1, there is no

need to divide the power by N. So applying a Fourier Transform is the same as multiplying the time domain vector to this matrix.

Considering the row number to be j, and the coloumn number to be k, we can say every value in the matrix is represented by w^{jk}. Dividing the coloumns into even numbered, and odd nmbered, and the samples into half, we can generalize the matrix as follows.

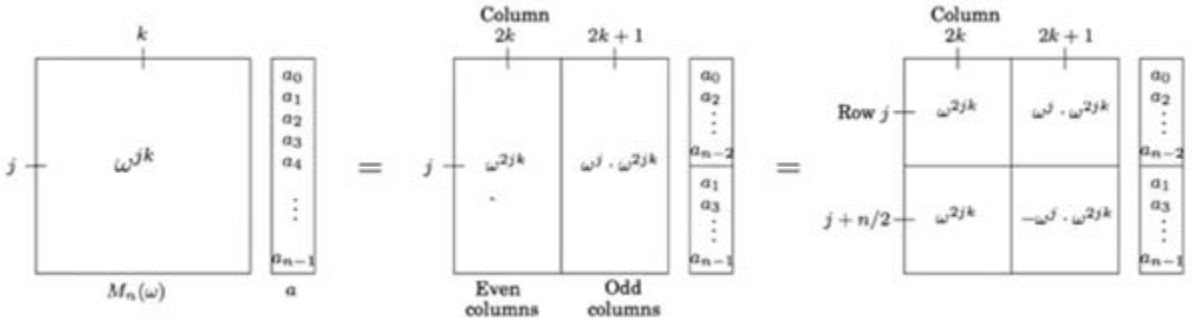

Fig 6.5 Fast Fourier Transform

We can see how a lot of the values get repeated. So to speed up the transform, we can recursively use FFT for half the bits separately, and the other half separately. Finally, we can use the respective multipliers for the odd coloumns, and obtain the same result.

Solving the recurrence relation, we get a time complexity of NlogN.

6.4.4 Conversion

We now know that we can use Fast Fourier Transforms to convert our time domain to frequency domain and it's also apparent that we lost the time domain in this process. Therefore it isn't possible to retain the time domain of the wave that we give into a Fourier Transform. As a little workaround to this, we decided instead to cut the whole song into multiple chunks, or multiple small waves and we would feed each of these small waves into a Fourier Transform thereby retaining its chunk number even though we lose the time in each chunk. Hence, we still have some notion of time, and in each interval of that time, we have a frequency domain. To demonstrate this, we took a random song, cut it up into chunks, and passed it into a Fast Fourier Transform and got a

table of values. We plotted the chunk number on the x axis, the different range of frequencies on the y axis, and colour coded the magnitude on the z axis.

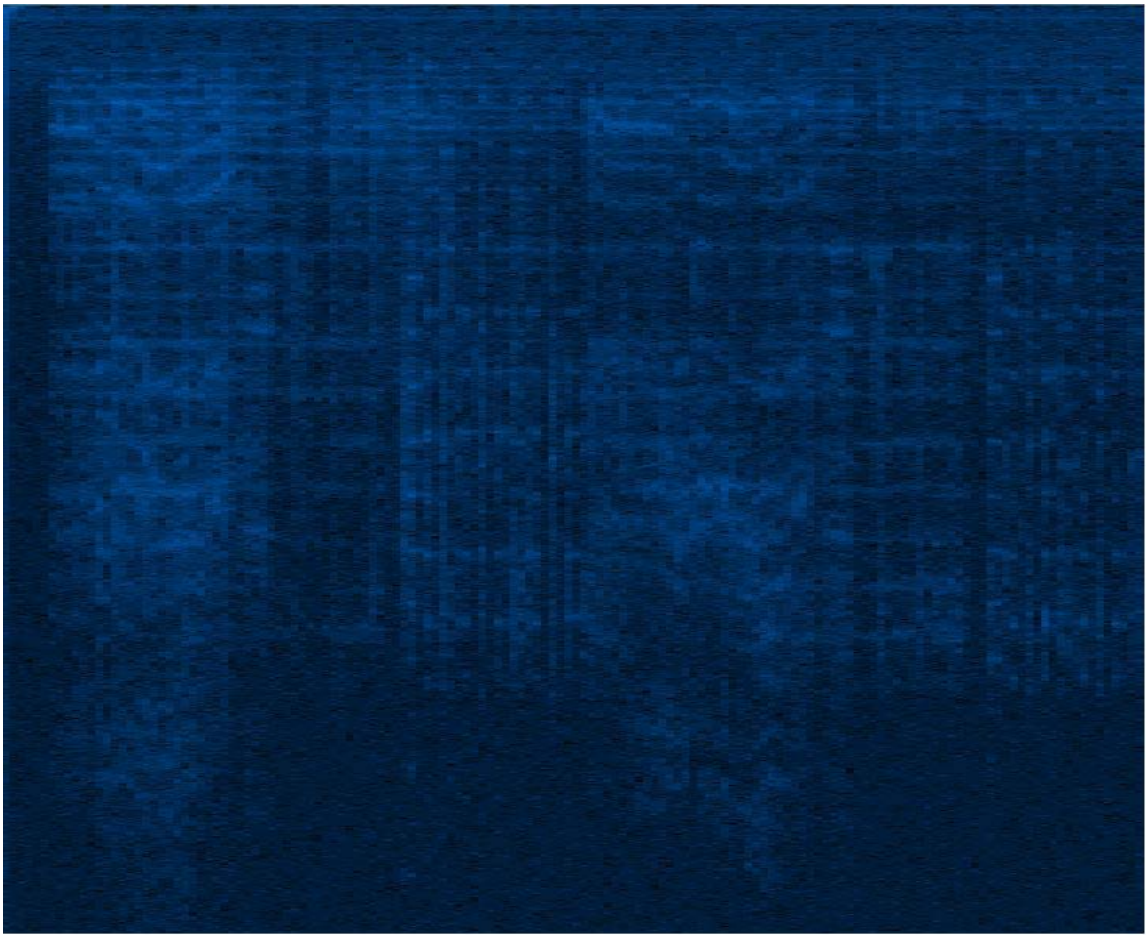

Fig 6.6 Spectogram 1 – Before Optimizing

The brighter the blue, the higher the magnitude.

With this, we had accomplished two things.

- We could take a song, and convert it into a time-frequency domain.
- We could take a user mic input, and do the same.

As to how we actually managed to take the mic input itself, we coded the following steps:

- We got a mixer from the available mixers
- We got a data line which pertained to that particular mixer.

- We got an audio input stream out of that data line and put it into a byte array by specifying the required format.

6.4.5 Optimizing

At the end of the last sub-topic, we had a three dimensional graph, which took up around 150MB of space for a 5MB song. Clearly, this wouldn't do, for two obvious reasons. One, the space complexity would be too much, and more importantly, two, this was too much data for a comparision not only in regard to its time complexity, but also the fact that the more naiive comparisions you make, the more chances of the comparision yielding false. On the whole, it was evident that we had to extract the most significant points or the most significant frequencies in each time slice instead of comparing all frequencies by brute force.

After much trial and error, we chose a frequency slice between 60 - 330 Hz, and we chose the size of each time slice as 4096 bytes. Between this region of 60 to 330, we cut up 5 periods of almost equal length, and chose the frequencies with the highest magnitudes in each period. This brought down both the time and space complexity by a huge amount.

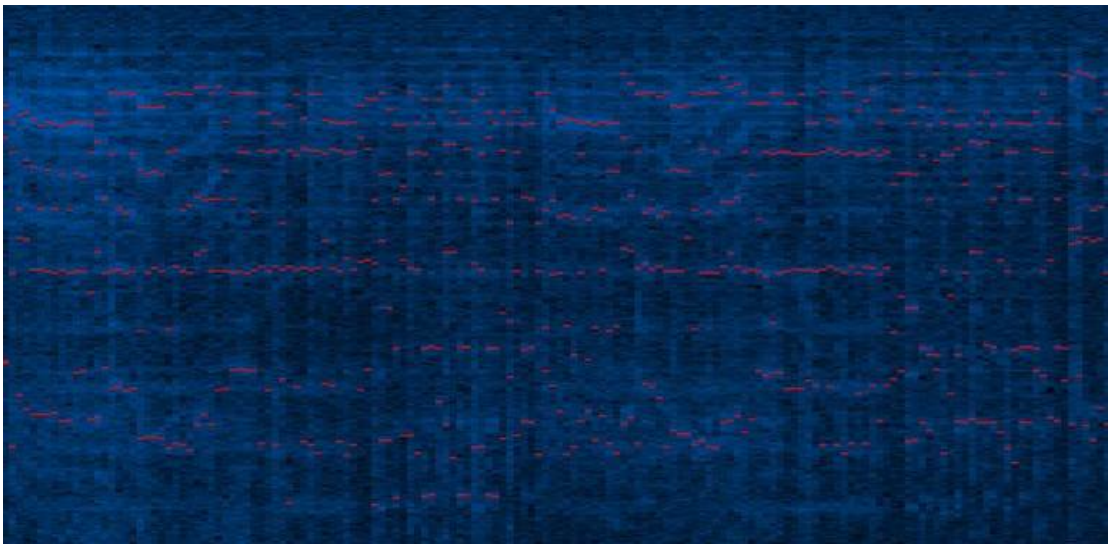

Fig 6.7 Spectogram 2 – After Optimizing

The bright red dots represent the most significant frequencies in each time slice

6.5 Music Synthesizer Part II (Hashing and Matching)

These four chosen frequency slices in our context represent a unique note. We had to know or store what collection of notes appear in each song so that when a mic input also happens to have that set of notes, we could see that the song with the highest matches with these notes would be the one the user was looking for. Now if we were to put inside the Song object all the notes that the song contained, in each search, we would end up looking at all songs. Instead, if we were to make a table of all possible notes, and all the songs in which each of these notes appear, then it would be just one straight lookup per note instead of an entire library search. (n to 1).

For this, we took the four frequency numbers and reduced them by a noise factor of two and coupled them together to make a hash. And this hash we used as a key for our frequency table. As a value for each key, we had an object called Occurrence info which contained the id of the song and the time slice in which the note appeared. With this much amount of pre-processing done, the actual match would just be to make notes out of a mic input and check the table to see what songs the notes appeared in.

This completes all the modules that have been implemented. Now we shall proceed to expand upon the actual code for the explanation we have given.

6.6 Source Code

```
package MusicBox;

import MusicLibrary.*;

import MusicPlayer.MusicPlayer;

import MusicSynthesizer.PatternMatcher;

import MusicSynthesizer.SongConvertor;

import java.io.*;

import java.util.*;

import java.util.logging.Level;

import java.util.logging.Logger;



/**

 *

 * @author Monkey D Alok

 */

public class MusicBox implements Serializable{


        MusicLibrary theLibrary;

        MusicPlayer thePlayer = new MusicPlayer();

        HashMap<Long,ArrayList<OccurrenceInfo>> frequencyHashTable;
```

```
Thread extract;
SongMixer mixJob;
UserInput theMic;
int songIdCounter;
Song currentSong = null;
 ArrayList<Song> currentList,shuffleList;
int songPosition;


RepeatState repeat = RepeatState.REPEATLIST;
boolean shuffle = false;
boolean exit_state;



VolumeControl volumeChanger = new VolumeControl();


public MusicBox(){
        //holds the list of currently playing songs
        currentList = new ArrayList<>();


        //holds the list of currently playing songs in shuffled state
        shuffleList = new ArrayList<>();
```

```
            //starts the songMixer method
            startMixer();


            load();
        }


        public void load(){


            ObjectInputStream load = null;
             try {
                    FileInputStream fileStream = null;
                    try {
                            fileStream = new FileInputStream( new
File("SaveData.ser"));
                            load = new ObjectInputStream(fileStream);
                            try {
                                    theLibrary = (MusicLibrary)load.readObject();
                                    frequencyHashTable =
(HashMap<Long,ArrayList<OccurrenceInfo>>)load.readObject();
                                    songIdCounter = (int)load.readObject();
                                    //System.out.println(songIdCounter);
```

```
                        } catch (ClassNotFoundException ex) {

Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
                        }
                    } catch (FileNotFoundException ex) {
                        theLibrary = new MusicLibrary();
                        frequencyHashTable = new HashMap<>();
                        songIdCounter = 0;
                    }

                } catch (IOException ex) {

Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
                }
            }

            public void save(){
                //try {
                    //if(extract.isAlive()){
                        //extract.join();
```

```
                //}
            //} catch (InterruptedException ex) {

        //Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
            //}
            ObjectOutputStream save = null;
            try {
                FileOutputStream fileStream = null;
                try {
                    fileStream = new FileOutputStream( new File("SaveData.ser"));
                } catch (FileNotFoundException ex) {

        Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
                }
                save = new ObjectOutputStream(fileStream);
                save.writeObject(theLibrary);
                save.writeObject(frequencyHashTable);
                save.writeObject(songIdCounter);
                exit_state = true;
            } catch (IOException ex) {
```

```
        Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
            } finally {
                try {
                    save.close();
                } catch (IOException ex) {
        Logger.getLogger(MusicBox.class.getName()).log(Level.SEVERE, null, ex);
                }
            }
        }

        public boolean playSong(){
            if(currentSong == null){
                return true;
            }
            if(thePlayer.getPath() == null ||
!thePlayer.getPath().equals(currentSong.getPath())){
                thePlayer.setPath(currentSong.getPath());
                thePlayer.play();
                return true;
            }
```

```
            else{
                thePlayer.resume();
                return false;
            }
        }

        public void pauseSong(){
            thePlayer.pause();
        }

        public void changeSong(Song s){
            currentSong = s;
        }

        public void stopSong(){
            if(!thePlayer.isSongDone()){
                thePlayer.stop();
            }
            thePlayer.init();
        }
```

```
	public void addSong(String containerName, File songPath, boolean isPlayList){
		songIdCounter++;
		System.out.println("Adding Song: " + songIdCounter);
		theLibrary.addSong(songIdCounter , containerName, songPath, isPlayList);

		Runnable extractHashes = new Extractor(songPath , songIdCounter);
		extract = new Thread(extractHashes);
		extract.start();

	}

	public int getCurrentSongLength(){
		if(currentSong == null){
			return -1;
		}
		return currentSong.getLength();
	}

	public String getCurrentSongTitle(){
		return currentSong.toString();
```

```
    }

    public void findSong(String containerName, String title, boolean isPlayList){
        changeSong(theLibrary.findSong(containerName, title, isPlayList));
    }

    public String[][] getTableContents(String containerName, boolean isPlayList){
        ArrayList<HashMap<Tags,String>> tableContents =
theLibrary.getContainerInfo(containerName, isPlayList);
        if(tableContents == null){
            return null;
        }

        return convert(tableContents);
    }

    public String[] getTableHeader(){
        ArrayList<String> header = new ArrayList<>();
        String[] headerArray;
        Object[] objArray;
        for(Tags tag : Tags.values()){
```

```
                header.add(tag.toString());
            }
            objArray = header.toArray();
            headerArray = new String[objArray.length];
            for(int i=0; i< objArray.length;i++){
                headerArray[i] = objArray[i].toString();
            }
            return headerArray;
    }

    public String[] getAlbumList(){
            Set<String> list;
            list = theLibrary.getAlbumNames();
            Object[] listObjArray = list.toArray();
            String[] listArray = new String[listObjArray.length];
            for(int i = 0; i < listObjArray.length; i++){
                listArray[i] = listObjArray[i].toString();
            }
            return listArray;
    }
```

```
public String[] getPlayListList(){
        Set<String> list;
         list = theLibrary.getPlayListNames();
        Object[] listObjArray = list.toArray();
        String[] listArray = new String[listObjArray.length];
        for(int i = 0; i < listObjArray.length; i++){
                listArray[i] = listObjArray[i].toString();
        }
        return listArray;
}

public void changeVolume(float slider){
        float volume = slider/20;
        volumeChanger.changeSpeaker(volume);
}

/**
 *
 * @param slider
 */
public synchronized  void scrollTo(int slider){
```

```
                int length = getCurrentSongLength();
                int total = thePlayer.getTotal();
                float posFloat = ((float)slider/length)*(float)total;
                int pos = (int)posFloat;
                if(Math.abs(slider - songPosition) < 2 ){
                        return;
                }
                else{
                        songPosition = slider;
                }

                thePlayer.pause();
                thePlayer.play(pos);

        }

        public String[][] search (String text , String option){
                ArrayList<HashMap<Tags,String>> listOfSongs;

                if(option.toLowerCase().equals(SearchOption.ALBUM.toString())){
```

```
                listOfSongs = theLibrary.albumSearch(text);
            }
            else
if(option.toLowerCase().equals(SearchOption.PLAYLIST.toString())){
                listOfSongs = theLibrary.playListSearch(text);
            }
            else{
                listOfSongs = theLibrary.songSearch(text);
            }

             if (listOfSongs == null) {
                return null;
            }

            return convert(listOfSongs);



        }
        public String[][] convert(ArrayList<HashMap<Tags,String>> tableContents){
            ArrayList<String> tableRow ;
            ArrayList<String[]> table = new ArrayList<>();
```

```
            for(HashMap<Tags,String> row : tableContents){
                tableRow = new ArrayList<>();
                for(Tags tag : Tags.values()){
                    tableRow.add(row.get(tag));
                }
                String[] tableRowArray = tableRow.toArray(new String[0]);
                table.add(tableRowArray);
            }
            String[][] tableArray = table.toArray(new String[0][0]);
            return tableArray;

    }

    public void updateNextSong() {
            Song toPlay = mixJob.getNextSong();
            stopSong();
            changeSong(toPlay);
    }

    public void updatePreviousSong() {
            Song toPlay = mixJob.getPrevSong();
```

```
            stopSong();
            changeSong(toPlay);
        }

        public void setRepeatState(RepeatState state) {
            repeat = state;
        }

        public void setShuffleState(boolean state) {
            shuffle = state;
        }

        /**
     *
     */
        public final void startMixer() {
            mixJob = new SongMixer();
            Thread mixThread = new Thread(mixJob);
            mixThread.start();
        }
```

```
public void letTheHummingBegin() {
        theMic = new UserInput();
}

public void captureMicInput() {
        theMic.startACapture();
}

public void stopCapture() {
        theMic.StopCurrentCapture();
}

public void cleanUpMic() {
        theMic.discardAudioData();
}

public void playCapturedSong() {
        theMic.playAudioData();
}

public void stopPlaying() {
```

```
            theMic.stopPlay();
        }

        /**
         *
         * @return
         */
        public String[][] searchTheSong() {
            byte[] audioData = theMic.getAudioData();
            ArrayList<Long> micHashes;
            micHashes = SongConvertor.convertThis(audioData);
            int hitId = PatternMatcher.matchThis(micHashes , frequencyHashTable);
            System.out.println(hitId);
            ArrayList<HashMap<Tags,String>> tableContents = theLibrary.getSongListById(hitId);
            return convert(tableContents);

        }

        public String[] getPlayLists(String currentSong) {
            ArrayList<String> results;
```

```
            results = new ArrayList<>();


            Set<String> playListNames = theLibrary.getPlayListNames();
            for(String playList : playListNames) {
                  ArrayList<Song> songs = theLibrary.getPlayListTable(playList);
                  for(Song song : songs) {
                        if(song.toString().equals(currentSong)) {
                              results.add(playList);
                              break;
                        }
                  }
            }


            String[] allPlayLists = null;
            try {
                  allPlayLists = results.toArray(new String[0]);
            } catch(NullPointerException ex) {
                  return null;
            }
            return allPlayLists;
      }
```

```
public String getAlbum(String selectedSong) {
        throw new UnsupportedOperationException("Not yet implemented");
}

public boolean isMicDone() {
        return theMic.isSongDone();
}

public void endOfHumming() {
        theMic = null;;
}

private class Extractor
        implements Runnable
{

        File songPath;
        int id;
        public Extractor(File songPath , int id) {
                this.songPath = songPath;
```

```
            this.id = id;
        }

        @Override
        public void run() {
            ArrayList<Long> songHashes;
            songHashes = SongConvertor.convertThis(songPath);

            for(int i = 0; i < songHashes.size(); i++){
                OccurrenceInfo selectedDataPoint = new OccurrenceInfo(i
, songIdCounter);
                ArrayList<OccurrenceInfo> listOfHits;
                if(frequencyHashTable.containsKey(songHashes.get(i))){
                    listOfHits                                    =
frequencyHashTable.get(songHashes.get(i));
                    listOfHits.add(selectedDataPoint);
                }
                else{
                    listOfHits = new ArrayList<>();
                    listOfHits.add(selectedDataPoint);
                }
                frequencyHashTable.put(songHashes.get(i) , listOfHits);
```

```
                }
                System.out.println("done");
        }
}


public class SongMixer implements Runnable {
        Song nextSong;
        @Override
        public void run() {

                while(exit_state == false && thePlayer.isSongDone() == true) {
                        try {
                                Thread.sleep(1000);
                        } catch(InterruptedException ex) {
                        }
                }

                //the thread runs as long as the program runs
                while(exit_state == false) {
```

```
//nobody will notice if the change in song is
//        just one second late
try {
        Thread.sleep(1000);
} catch(InterruptedException ex) {
}

//loops until the song has finished playing
if(!thePlayer.isSongDone()) {
        continue;
}

//then updates the song list
updateList();

//loops back if there is nothing to play
//not sure about later, but this is useful
//at least at the start, I think.

if(currentList.isEmpty()) {
        continue;
```

```
}

//if it isn't supposed to repeat, it'll keep looping
//it's up to the user to play a song.

if (repeat == RepeatState.NOREPEAT) {
        changeSong(null);
        continue;
}

//this is used to repeat a song over and over
//it can only get this far after a song is complete,
//then it just changes the song to itself, and starts
//all over again.

if (repeat == RepeatState.REPEATSONG) {
        changeSong(currentSong);
        while(thePlayer.isSongDone()) {
                try {
                        Thread.sleep(1000);
                } catch(InterruptedException ex) {
```

```
                    ex.printStackTrace();
                }
            }
            continue;
        }

        //whether it's a repeat-list, or play-list-once
        //is up to the function

        nextSong = getNextSong();

        /*if (nextSong == null) {

            //loop back, there'll be nothing more to
            //play until the user chooses to
            changeSong(null);
            continue;
        }*/

        changeSong(nextSong);
        while(thePlayer.isSongDone()) {
```

```
                        try {
                                Thread.sleep(1000);
                        } catch(InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }
}

public Song getNextSong() {
        ArrayList<Song> theList;

         //this selects whether the list is shuffled or not
        //this is mainly for abstraction purposes
        updateList();

        if(shuffle) {
                theList = shuffleList;
        }
        else {
                theList = currentList;
```

```
                }

                int index;
                index = theList.indexOf(currentSong);
                index = (index + 1) % theList.size();

                //if the condition was to repeat the list once, and if we
                //had reached the end of the list in the last song, the
                //new index would be 0, which is when we should stop playing

                if ((repeat == RepeatState.REPEATLISTONCE) && (index == 0))
{
                        return null;
                }

                //otherwise, there is no difference between the repeat-list
                //and repeat-list-once. We just return the new song in the list
                return theList.get(index);
        }

        public Song getPrevSong() {
```

```
ArrayList<Song> theList;


 //this selects whether the list is shuffled or not
//this is mainly for abstraction purposes


updateList();
if(shuffle) {
        theList = shuffleList;
}
else {
        theList = currentList;
}


int index;
index = theList.indexOf(currentSong);


if(index == 0) {
        index = theList.size() - 1;
}


else {
```

```
                index--;
            }

            return theList.get(index);
        }

        public void updateList() {

            ArrayList<Song> temp;
            temp = theLibrary.getSongList();


            //If nothin is being played, the currentList must be empty
            //The same applies to the shuffleList

            if (temp.isEmpty() || temp == null) {
                currentList.clear();
                shuffleList.clear();
                return;
            }
```

```
            //If the lists are the same, there is nothing to update
            //This also adds the benefit that you can click on a song
            //in the same list and not change the state of the list

            if (temp.equals(currentList)) {
                return;
            }

            //otherwise, the list needs to be updated
            //the lists are cleared;
            currentList.clear();

            shuffleList.clear();

            //populated with the new list;
            currentList.addAll(temp);
            shuffleList.addAll(temp);

            //and finally, the shuffleList is shuffled
            Collections.shuffle(shuffleList);
        }
```

```
        }

}
package MusicSynthesizer;

/**
 *
 * @author Monkey D Alok
 */
public class Complex {
        public final double re;   // the real part
        private final double im;   // the imaginary part

        // create a new object with the given real and imaginary parts
        public Complex(double real, double imag) {
                re = real;
                im = imag;
        }
```

```
// return a string representation of the invoking Complex object
public String toString() {
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im <  0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
}

// return abs/modulus/magnitude and angle/phase/argument
public double abs() {
        return Math.hypot(re, im);
} // Math.sqrt(re*re + im*im)

public double phase() {
        return Math.atan2(im, re);
}// between -pi and pi

// return a new Complex object whose value is (this + b)
public Complex plus(Complex b) {
        Complex a = this;             // invoking object
```

```
        double real = a.re + b.re;
        double imag = a.im + b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this - b)
    public Complex minus(Complex b) {
        Complex a = this;
        double real = a.re - b.re;
        double imag = a.im - b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this * b)
    public Complex times(Complex b) {
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }
```

```
// scalar multiplication
// return a new object whose value is (this * alpha)
public Complex times(double alpha) {
        return new Complex(alpha * re, alpha * im);
}

// return a new Complex object whose value is the conjugate of this
public Complex conjugate() {
        return new Complex(re, -im);
}

// return a new Complex object whose value is the reciprocal of this
public Complex reciprocal() {
        double scale = re*re + im*im;
        return new Complex(re / scale, -im / scale);
}

// return the real or imaginary part
public double re() {
        return re;
}
```

```
    public double im() {
        return im;
    }

    // return a / b
    public Complex divides(Complex b) {
        Complex a = this;
        return a.times(b.reciprocal());
    }

    // return a new Complex object whose value is the complex exponential of this
    public Complex exp() {
        return new Complex(Math.exp(re) * Math.cos(im), Math.exp(re) *
Math.sin(im));
    }

    // return a new Complex object whose value is the complex sine of this
    public Complex sin() {
        return new Complex(Math.sin(re) * Math.cosh(im), Math.cos(re) *
Math.sinh(im));
    }
```

```
    // return a new Complex object whose value is the complex cosine of this
    public Complex cos() {
        return new Complex(Math.cos(re) * Math.cosh(im), -Math.sin(re) * Math.sinh(im));
    }

    // return a new Complex object whose value is the complex tangent of this
    public Complex tan() {
        return sin().divides(cos());
    }

     // a static version of plus
    public static Complex plus(Complex a, Complex b) {
        double real = a.re + b.re;
        double imag = a.im + b.im;
        Complex sum = new Complex(real, imag);
        return sum;
    }

}
```

```
package MusicSynthesizer;


/**
 *
 * @author Monkey D Alok
 */
public class Transformer
{
        // compute the Transformer of x[], assuming its length is a power of 2
        /**
         *
         * @param x
         * @return a complex array of frequency domain
         */
        public static Complex[] fastFourierTransform(Complex[] x) {
                int N = x.length;


                // base case
                if (N == 1){
                        return new Complex[] { x[0] };
                }
```

```
// radix 2 Cooley-Tukey Transformer
if (N % 2 != 0) {
        throw new RuntimeException("N is not a power of 2");
}

// fastFourierTransform of even terms
Complex[] even = new Complex[N/2];
for (int k = 0; k < N/2; k++) {
        even[k] = x[2*k];
}
Complex[] q = fastFourierTransform(even);

// fastFourierTransform of odd terms
Complex[] odd  = even;  // reuse the array
for (int k = 0; k < N/2; k++) {
        odd[k] = x[2*k + 1];
}
Complex[] r = fastFourierTransform(odd);

// combine
```

```
                Complex[] y = new Complex[N];
                for (int k = 0; k < N/2; k++) {
                        double kth = -2 * k * Math.PI / N;
                        Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
                        y[k]       = q[k].plus(wk.times(r[k]));
                        y[k + N/2] = q[k].minus(wk.times(r[k]));
                }
                return y;
        }

}
package MusicSynthesizer;

import java.io.*;
import java.util.ArrayList;
import javax.sound.sampled.*;

/**
 *
 * @author The Speed Phantom
 */
```

```
public class SongConvertor
{
    static final int NOISE = 2;
    static final int CHUNK = 4096;

    public static ArrayList<Long> convertThis(File fileName) {
        try {
            File songPath;
            songPath = fileName;
            try (AudioInputStream inputStream = AudioSystem.getAudioInputStream(songPath))
            {
                AudioInputStream convertedInputStream = null;
                AudioFormat baseFormat = inputStream.getFormat();
                AudioFormat decodedFormat;
                decodedFormat = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
                        baseFormat.getSampleRate(),
                        16,
                        baseFormat.getChannels(),
                        baseFormat.getChannels() * 2,
                        baseFormat.getSampleRate(),
```

```
                                                    true);

                        convertedInputStream =
AudioSystem.getAudioInputStream(decodedFormat, inputStream);
                        convertedInputStream = convertStream(1 , 8 , true ,
convertedInputStream);
                        return rawBytes(convertedInputStream);
                    }
                }
                catch (UnsupportedAudioFileException | IOException |
LineUnavailableException e){
                    System.out.println("Sad");
                }
                return null;
            }

        private static AudioInputStream convertStream(int channels , int sampleSize ,
boolean isBigEndian , AudioInputStream sourceStream){
                AudioFormat sourceFormat = sourceStream.getFormat();
                AudioFormat targetFormat;
                targetFormat = new AudioFormat(sourceFormat.getEncoding(),
                        sourceFormat.getSampleRate(),
```

```
                    sampleSize,
                    channels,
                    calculateFrameSize(channels, sampleSize),
                    sourceFormat.getFrameRate(),
                    isBigEndian);


            return AudioSystem.getAudioInputStream(targetFormat , sourceStream);
        }

        private static int calculateFrameSize(int nChannels, int nSampleSizeInBits){
            return ((nSampleSizeInBits + 7) / 8) * nChannels;
        }

        private static ArrayList<Long> rawBytes(AudioInputStream inputStream) throws
IOException , LineUnavailableException{

            ByteArrayOutputStream byteArrayBuffer = new
ByteArrayOutputStream();

            int nRead;
```

```
            byte[] intermediateBuffer = new byte[16384];

            while ((nRead = inputStream.read(intermediateBuffer, 0,
intermediateBuffer.length)) != -1) {
                byteArrayBuffer.write(intermediateBuffer, 0, nRead);
            }

            byteArrayBuffer.flush();

            byte[] byteArray = byteArrayBuffer.toByteArray();

            inputStream.close();

            return convertThis(byteArray);

        }

        public static ArrayList<Long> convertThis(byte[] byteArray) {
            int size = byteArray.length;
            System.gc();
```

```
//int size = 44100 * 20;
int numberOfChunks = size / CHUNK;
Complex[] convertedArray = new Complex[CHUNK];

double[] highestMagnitudes = {0.0 , 0.0 , 0.0 , 0.0};
int [] frequenciesOfHighestMagnitudes = new int[4];
int[][] bestFrequencies = new int[numberOfChunks][];

for(int chunkNumber = 0; chunkNumber < numberOfChunks; chunkNumber++){
    Complex[] audioChunk = new Complex[CHUNK];
    for(int byteInChunk = 0; byteInChunk < CHUNK; byteInChunk++){
        audioChunk[byteInChunk] = new Complex(byteArray[(chunkNumber*CHUNK) + byteInChunk] , 0);
    }
    convertedArray = Transformer.fastFourierTransform(audioChunk);
    for(int byteInChunk = 60; byteInChunk < 330; byteInChunk++){
        double candidate = Math.log(convertedArray[byteInChunk].abs());
        int range = getRange(byteInChunk);
        if(highestMagnitudes[range] < candidate){
```

```
                                highestMagnitudes[range] = candidate;
                                frequenciesOfHighestMagnitudes[range] = byteInChunk;
                            }
                        }
                        bestFrequencies[chunkNumber] = frequenciesOfHighestMagnitudes;
                        highestMagnitudes = new double[5];
                        frequenciesOfHighestMagnitudes = new int[4];
                    }

                    ArrayList<Long> hashes = new ArrayList<>();



                    for(int[] sample : bestFrequencies){
                        hashes.add(makeHash(sample));
                    }

                    //for(Long hash : hashes){
                        //System.out.println(hash);
                    //}
```

```
        return hashes;
    }

    private static int getRange(int index){
        int[] ranges = {100 , 160 , 240 , 330};
        int i = 0;
        while(ranges[i] < index){
            i++;
        }
        return i;
    }

    private static long makeHash(int[] samples){

        long hash = (((samples[3] - (samples[3] % NOISE)) * 1000000000l)
            + ((samples[2] - (samples[2] % NOISE)) * 1000000l)
            + ((samples[1] - (samples[1] % NOISE)) * 1000l)
            + ((samples[0] - (samples[0] % NOISE)) * 1l));
        //System.out.println(hash);
        return hash;
```

```
        }


}

package MusicSynthesizer;


import MusicBox.OccurrenceInfo;

import java.util.*;

import java.util.Map.Entry;


/**

 *

 * @author The Speed Phantom

 */

public class PatternMatcher

{


        public static int matchThis(ArrayList<Long> micHashes, HashMap<Long, ArrayList<OccurrenceInfo>> frequencyHashTable) {

                HashMap<Integer , Integer> hitCount = new HashMap<>();

                Set<Long> keys;

                //keys = frequencyHashTable.keySet();
```

```
//for(Long key : keys){
        //System.out.println(key);
//}
for(Long songHash : micHashes){
        if(frequencyHashTable.containsKey(songHash)){
                ArrayList<OccurrenceInfo> candidates = frequencyHashTable.get(songHash);
                for(OccurrenceInfo candidate : candidates){
                        int songId = candidate.getId();
                       // System.out.println(songId);
                        int count = 0;
                        if(hitCount.containsKey(songId)){
                                count = hitCount.get(songId);

                        }
                       count++;
                       hitCount.put(songId, count);
                       hitCount.put(songId, count);
                }
        }
}
```

```
            return keyOfHighestValue(hitCount);
        }

        public static <K, V extends Comparable<V>> K keyOfHighestValue(Map<K, V>
map) {
            K bestKey = null;
            V bestValue = null;
            for (Entry<K, V> entry : map.entrySet()) {
                if (bestValue == null || entry.getValue().compareTo(bestValue) >
0) {
                    bestKey = entry.getKey();
                    bestValue = entry.getValue();
                }
            }
            return bestKey;
        }

}
package MusicLibrary;
import MusicSynthesizer.*;
import java.io.*;
```

```
import java.util.*;


/**
 *
 * @author Monkey D Alok
 */
public class MusicLibrary implements Serializable
{
        ArrayList<Song> songList;
        HashMap<String,ArrayList<Song>> albumTable = new HashMap<>();
        HashMap<String,ArrayList<Song>> playListTable = new HashMap<>();

        public ArrayList<Song> getPlayListTable(String s) {
                return playListTable.get(s);
        }


        public void addSong(int songId , String containerName, File path, boolean isPlayList){

                SongMaker theMaker = new SongMaker(songId);
```

```
Song theSong = theMaker.makeASong(path);
if(albumTable.get(theSong.getAlbumName()) == null){
        songList = new ArrayList<>();
        songList.add(theSong);
        albumTable.put(theSong.getAlbumName(), songList);
        if(containerName.toLowerCase().equals("library")){
                return;
        }
}
else{
        songList = albumTable.get(theSong.getAlbumName());
        songList.add(theSong);
}
if(isPlayList){

        if(playListTable.get(containerName) == null){
                songList = new ArrayList<>();
                songList.add(theSong);
                playListTable.put(containerName, songList);
        }
        else{
```

```
                songList = playListTable.get(containerName);
                songList.add(theSong);

            }

        }


    }

    public Song findSong(String containerName, String title, boolean isPlayList){

        if(isPlayList){
            songList = playListTable.get(containerName);
            for(Song aSong: songList ){
                if(aSong.toString().equals(title)){
                    return aSong;
                }
            }
            return null;
        }
```

```
            else{
                songList = albumTable.get(containerName);
                for(Song aSong: songList ){
                    if(aSong.toString().equals(title)){
                        return aSong;
                    }
                }
                return null;
            }

        }

    public ArrayList<HashMap<Tags,String>> getContainerInfo(String containerName, boolean isPlayList){


            if(isPlayList){
                songList = playListTable.get(containerName);
            }
            else{
                songList = albumTable.get(containerName);
```

```
            }

            return extractInfo(songList);



    }

    public Set<String> getAlbumNames(){
            return albumTable.keySet();
    }

     /**
    *
    * @return
    */
    public Set<String> getPlayListNames(){
            return playListTable.keySet();
    }

    public ArrayList<HashMap<Tags,String>> albumSearch(String text){
            System.out.print(text);
```

```
// stores results
ArrayList<Song> results = new ArrayList<>();

// holds all album names
Set<String> albumNames = getAlbumNames();

// iterates over all the album names
for (String s : albumNames) {

    //places lowercase content in temp variable
    String temp = s.toLowerCase();

    // if the given search query is contained in the album name:
    if ( temp.contains(text.toLowerCase())){

        // all the songs in the album are added to results
        results.addAll(albumTable.get(s));
    }
}
for(Song s : results){
    System.out.println(s.toString());
```

```
            }

            return extractInfo(results);

    }


    public ArrayList<HashMap<Tags,String>> playListSearch(String text){


       //stores results

       ArrayList<Song> results = new ArrayList<>();


       // holds all playlist names

       Set<String> playListNames = getPlayListNames();


            // iterates over all the playlist names

       for (String s : playListNames) {


            //places lowercase content in temp variable

            String temp = s.toLowerCase();


            // if the given search query is contained in the playlist name:

            if ( temp.contains(text.toLowerCase())){
```

```
                // all the songs in the playlist are added to results
                results.addAll(playListTable.get(s));
                    }
            }
            for(Song s : results){
                    System.out.println(s.toString());
            }
            return extractInfo(results);



}

public ArrayList<HashMap<Tags,String>> songSearch(String text){

   // stores results
   ArrayList<Song> results = new ArrayList<>();

   // holds all album names
   Set<String> albumNames = getAlbumNames();

        // iterates over all album names
```

```
            for (String s : albumNames) {


                // all songs in the album are stored in a songlist
                ArrayList<Song> temp = albumTable.get(s);


                // iterating over all the songs in the songlist
                for (Song trill : temp) {


                    // if a given search query is contained in the song-name:
                    if (((trill.toString()).toLowerCase()).contains(text.toLowerCase())) {


                        // it is added to the results
                        results.add(trill);
                            }
                    }
                }
                return extractInfo(results);
        }
        public ArrayList<HashMap<Tags,String>> extractInfo(ArrayList<Song>
songList){
                HashMap<Tags,String> info;
```

```
            ArrayList<HashMap<Tags,String>> infoList = new ArrayList<>();
            if(songList == null){
                return null;
            }
            for(Song s : songList){
                info = s.getTags();
                infoList.add(info);

            }
            return infoList;

    }

public ArrayList<Song> getSongList(){
    return songList;
}

    public ArrayList<HashMap<Tags, String>> getSongListById(int[] hitIds) {
            throw new UnsupportedOperationException("Not yet implemented");
    }
```

```
        public ArrayList<HashMap<Tags, String>> getSongListById(int hitIds) {
                System.out.println(hitIds);
                ArrayList<HashMap<Tags, String>> data = new ArrayList<>();
                Set<String> albumNames = getAlbumNames();
                for(String album : albumNames){
                        System.out.println(album);
                        ArrayList<Song> songs = albumTable.get(album);
                        for(Song song : songs){
                                System.out.println("Song Id : " + song.getId());
                                if(song.getId() == hitIds){
                                        System.out.println(song.toString());
                                         data.add(song.getTags());
                                }
                        }
                }
                return data;
        }
}
package MusicPlayer;


import java.io.*;
```

```
import javazoom.jl.player.Player;



/**
 *
 * @author Monkey D Alok
 */
public class MusicPlayer implements Runnable , Serializable
{
    private Player player;
    private FileInputStream FileStream;
    private BufferedInputStream BufferStream;
    private boolean canResume;
    private File path;
    private int total;
    private int stopped;
    private boolean valid;

    public MusicPlayer(){
        path = null;
        init();
```

```
    }

    public void init(){
        player = null;
        FileStream = null;
        valid = false;
        BufferStream = null;
        total = 0;
        stopped = 0;
        canResume = false;
    }

    public boolean canResume(){
        return canResume;
    }

    public boolean isSongDone(){
            if(player == null ){
                return true;
            }
            return false;
```

```
    }

    public void setPath(File path){
        this.path = path;
    }

    public int getTotal(){
            return total;
    }

    public File getPath(){
                return path;
    }
    public boolean play(){
        return play(-1);
    }

    public boolean play(int pos){
        valid = true;
        canResume = false;
        if(path == null){
```

```
            return false;
        }
        try{
            FileStream = new FileInputStream(path);
                System.out.println("ha");
            total = FileStream.available();
            if(pos > -1) {
                FileStream.skip(pos);
            }
            BufferStream = new BufferedInputStream(FileStream);
            player = new Player(BufferStream);
            Thread song = new Thread(this);
            song.start();
        }
        catch(Exception e){
          //JOptionPane.showMessageDialog(null, "Error playing mp3 file");
            valid = false;
                pos = -1;
        }
        return valid;
    }
```

```
public void pause(){
    try{
    stopped = FileStream.available();
    player.close();
    FileStream = null;
    BufferStream = null;
    if(valid) canResume = true;
    }catch(Exception e){
    }
}

public void resume(){
    if(!canResume){
        return;
    }
    if(play(total-stopped)){
        canResume = false;
    }
}
```

```
	public void stop(){
		player.close();
		init();
	}



	public void run(){
		try{
			player.play();
		}
		catch(Exception e){
		//JOptionPane.showMessageDialog(null, "Error playing mp3 file");
		valid = false;
		}
	}


}
```

Chapter 7

Analysis

ANALYSIS

After developing the application, we had to analyze the efficiency. To do this, we decided to take three parameters.

- Number of Hits
- Accuracy
- Resource Consumption

7.1 Number of hits

We decided to test this with 6 different types of genres of music:

- Rock
- Soft
- Psychedelic
- Pop
- Trance
- Hip-Hop

We tried the song-matching algorithms against 50 samples in each genre. Plotting a graph of the song sample against the number of hits returned by the application, we received:

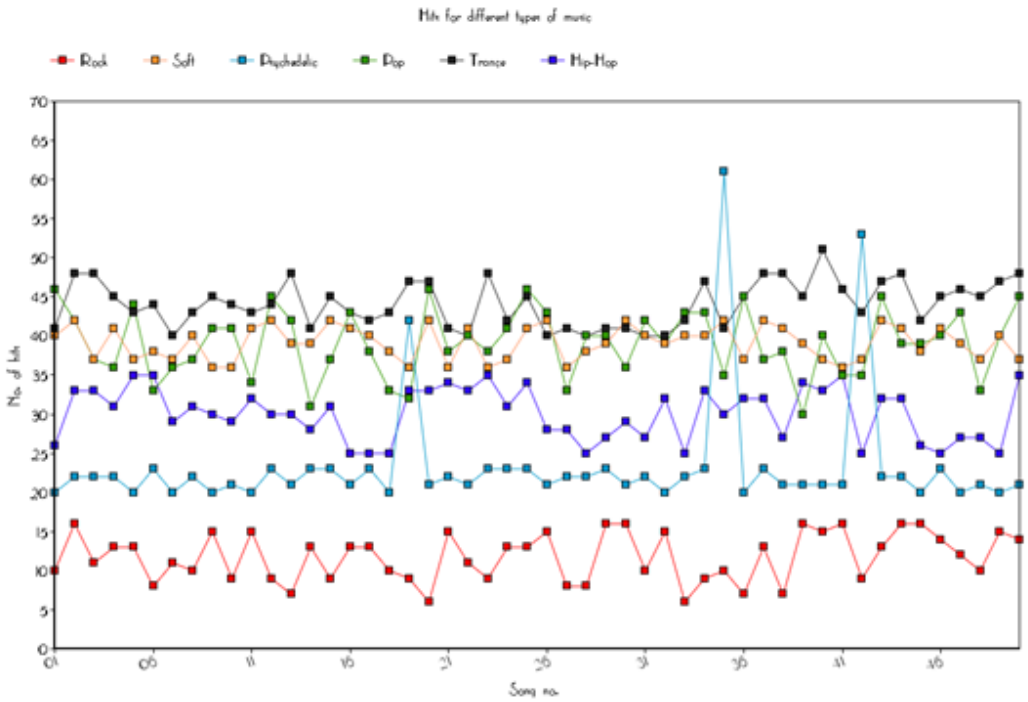

Fig 7.1 All Genres Analysis

7.1.1 Rock

In this genre, the magnitude of the instruments is comparable to that of the vocals. This, combined with background noise, filters out most of the songs giving back very few hits.

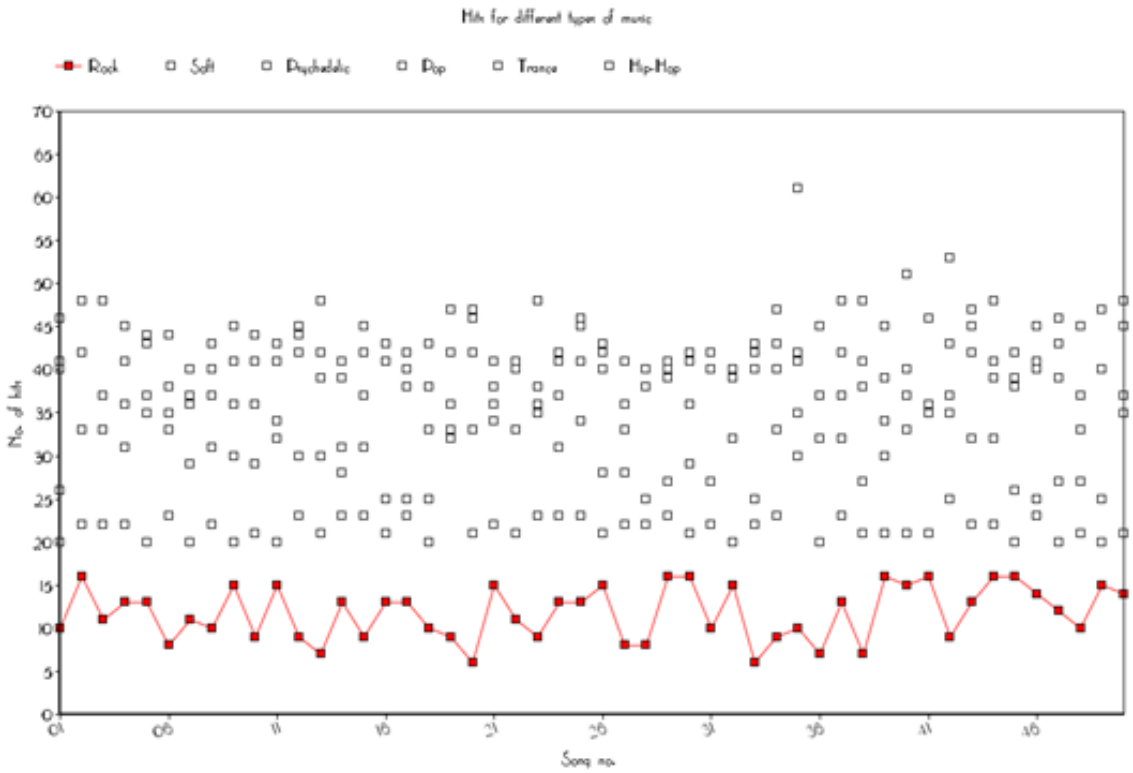

Fig 7.2 Rock

7.1.2 Soft

This genre is composed of music that is almost exclusively vocal. Without instrument interference, this genre is easy to compare to, and returns relatively more results.

7.1.3 Psychedelic

This genre uses natural instruments, and not much importance to vocal singing. Because of the importance to instruments, the number of hits is kept low, but we can observe some unexpected spikes in the number of hits, which can be explained by repetition of music patterns among the decade's worth of songs recorded.

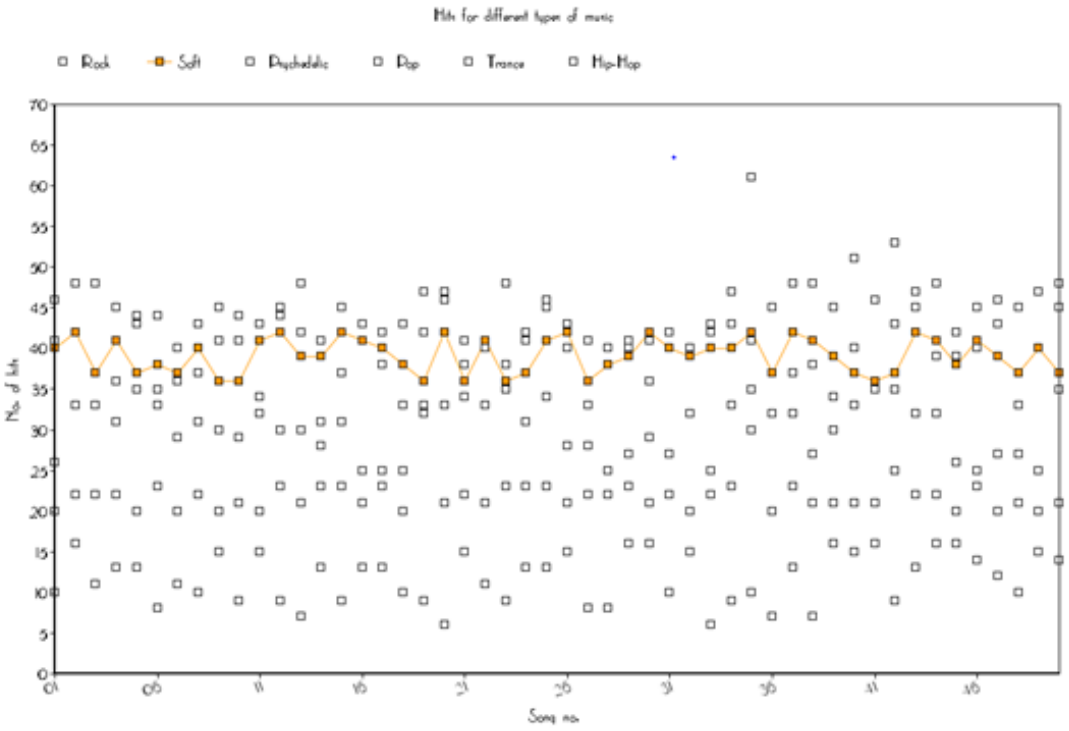

Fig 7.3 Soft

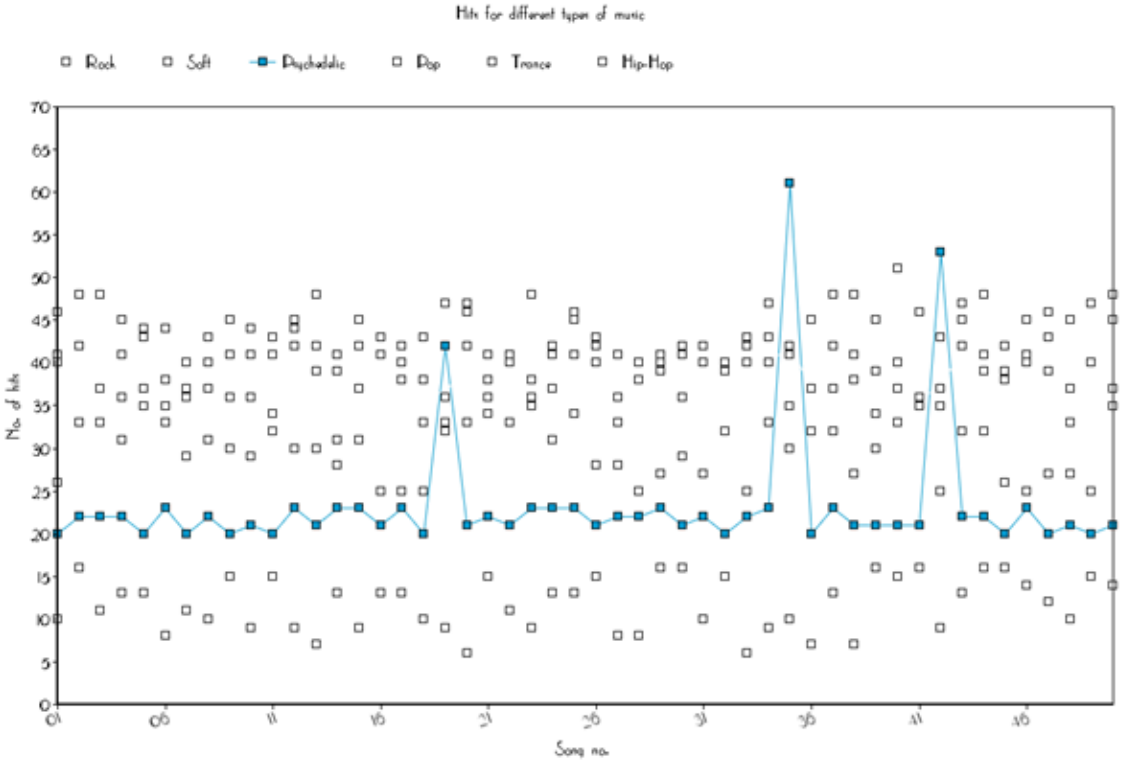

Fig 7.4 Psychedelic

7.1.4 Pop

This is the most popular, and simple of all the genres. Even if it is easily filtered by the instruments, the sheer number of Pop songs results in a high number of hits.

7.1.5 Trance

This is possibly the most varied of all the genres, stretching from music that is high and upbeat, to music that is low and melancholy. The instruments are used to incite emotion, but is more or less disjoint from the vocal of the song, nullifying the effect. This turns up the most results in all the genres.

7.1.6 Hip-Hop

This is also mainly vocal, with beats in the background, and emphasizes mainly on lyrics. The beats are repetitive, however, and that helps in narrowing down the song-match-list to a reasonable size.

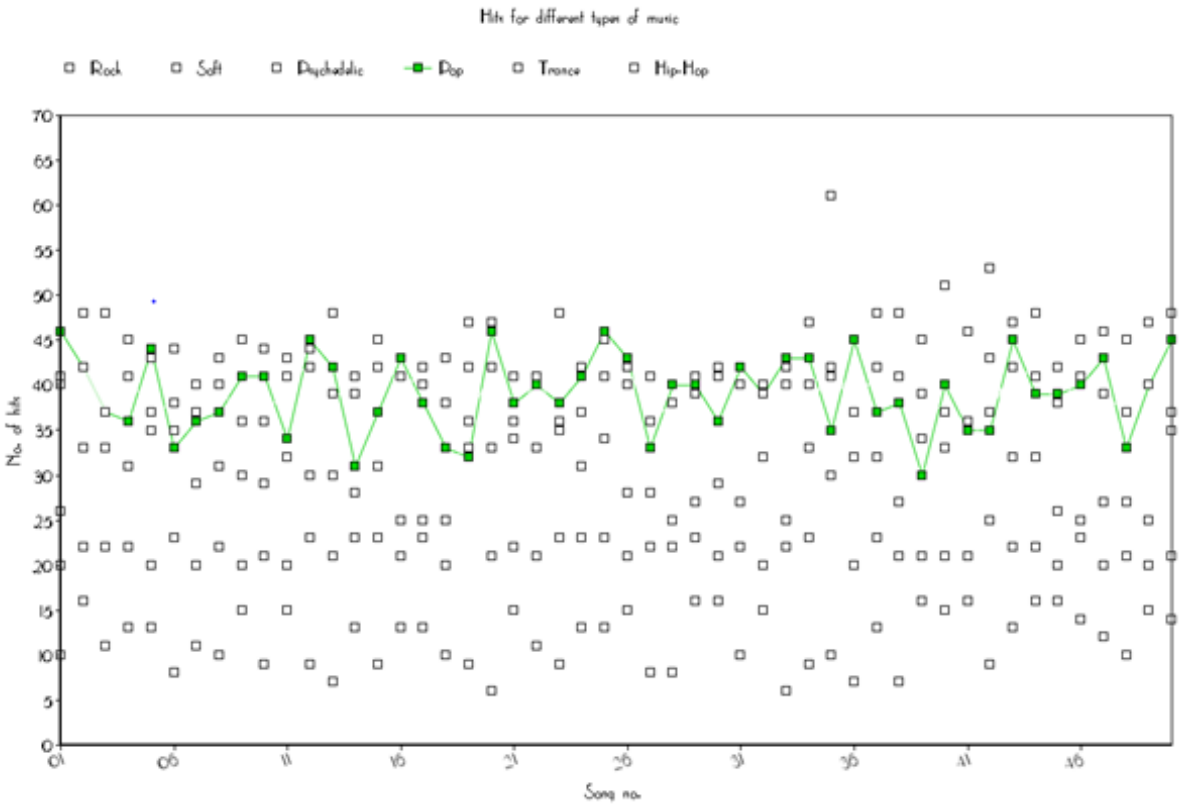

Fig 7.5 Pop

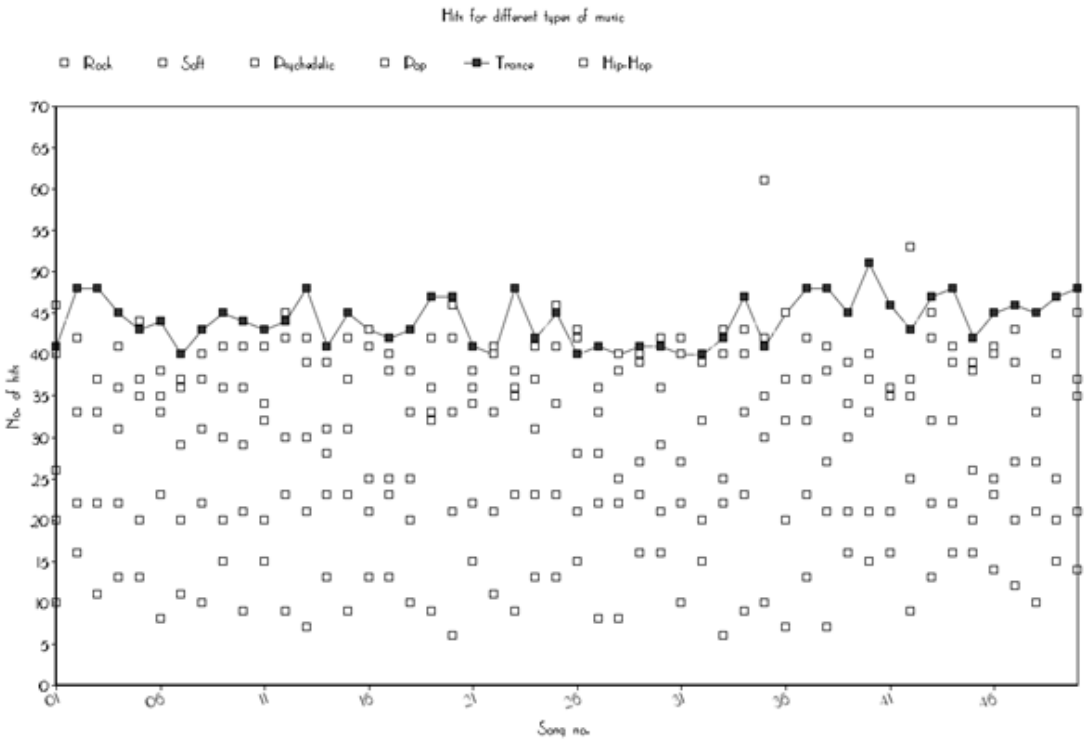

Fig 7.6 Trance

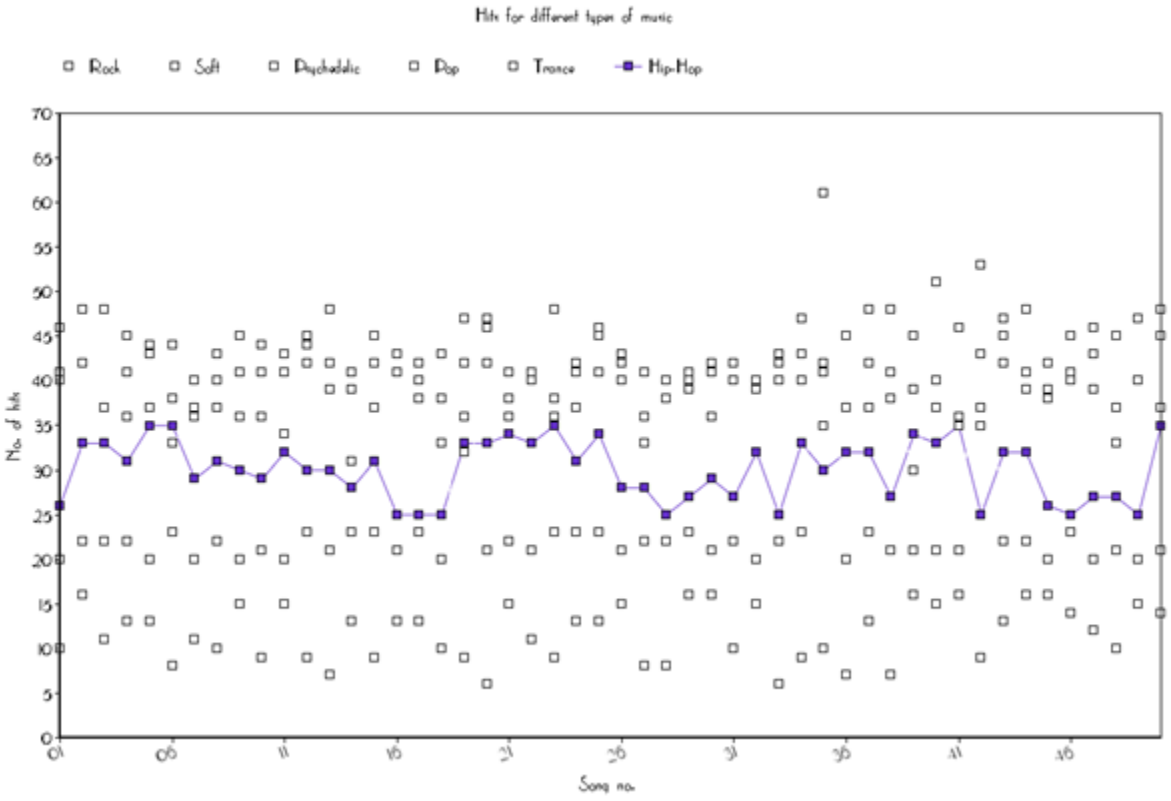

Fig 7.7 Hip - Hop

7.2 Accuracy

	Shazam	Tra Laa Laa
Rock	100%	100%
Soft	100%	100%
Psychedelic	100%	98%
Pop	100%	98%
Trance	98%	98%
Hip-Hop	100%	100%

Table 7.1 Accuracy

Note: Only 50 samples were tested for, so the percentages are in multiples of 2.

7.3 Resource consumption

	Tra Laa Laa	Shazam
Music files	MP3 files, provided by the user	MPEG-7 files (HD)
Space	500MB Hard Disk Space	500TB Server Space
Hardware	1 Laptop (4GB RAM, 512MB for IDE)	4 Mainframes
Time	4 students for 3 months	40 employees for 10 years
Accuracy	99%	99.7%

Table 7.2 Resource Consumption

Chapter 8

Testing

TESTING

Testing parameters were pretty straightforward. We first tested the music application and then the humming mode. The tests were targeted on the major functionality that we aimed to offer via the application.

8.1 Music Application test cases

Our Music Application mainly boasts of a scroller, volume slider, repeat and shuffle modes and text search functionality. We can add a file using drag and drop or add file dialog. These features have been tested as follows:

Testcase Description	To play the Music file
Expected Result	Should play the music via speakers
Actual Result	As expected
Status	OK

Testcase Description	Requesting the file metadata from music library
Expected Result	Should receive metadata from music library
Actual Result	As expected
Status	OK

Testcase Description	To check the scroller function
Expected Result	Should increment scroller position at proper intervals and should halt at the end when the song ends
Actual Result	As expected
Status	OK

Testcase Description	To check the shuffle function
Expected Result	Play a song deviating from the actual order
Actual Result	As expected
Status	OK

Testcase Description	To check repeat function
Expected Result	Play the current song repeatedly
Actual Result	As expected
Status	OK

Testcase Description	To check if user has entered the input via mic at humming mode
Expected Result	Spectrogram of the user's mic input
Actual Result	As expected
Status	OK

Testcase Description	To check add song functionality
Expected Result	Store the file and convert the files into its respective spectrograms
Actual Result	As expected
Status	OK

Testcase Description	Comparison of user's and stored spectrograms
Expected Result	An almost match of both the spectrograms
Actual Result	As expected
Status	OK

Testcase Description	To search a song by text in the music library
Expected Result	An exact result of the song with its metadata
Actual Result	As expected
Status	OK

8.2 humming mode test cases

The humming mode test cases are more specific to the topic of the project. We need to check if the conversion happens when a song is added to the app. For this we display the hashes and check if they correspond with the actual numbers. We check if the spectrograms of the mic input and song mp3 look similar. We check if a song search produces expected results. We check if the app still gives results under variable noise conditions and close song possibilities.

Testcase Description	Search function with one song in database and one input
Expected Result	Complete match with no of hits being more than 10

Actual Result	As expected
Status	OK

Testcase Description	Search function with 5 songs in the database and one user input
Expected Result	A list of songs with the best match on top of list and having over 10 hits for the best match and less than 10 for the rest of the songs in the list
Actual Result	As expected
Status	OK

Testcase Description	Search function with over 150 samples in database and one user input
Expected Result	A list of songs with best match on top of list and having more than 35 hits for the best match and a far lesser number for the rest of the songs in the list
Actual Result	As expected
Status	OK

Testcase Description	Search function when songs have similar chords and rhythm in the database
Expected Result	A list of songs with the best match on top of list and the difference in hits between the best match and the subsequent matches being more than 10
Actual Result	As expected
Status	OK

Testcase Description	Search function with noise frequencies ranging from 60-140Hz
Expected Result	A list of songs with the best match on top of list and the difference in hits between the best match and the subsequent matches being more than 10
Actual Result	As expected
Status	OK

Testcase Description	Search function with low quality mp3 files in the library
Expected Result	A list of songs with the best match on top of list
Actual Result	As expected
Status	OK

8.3 Snapshots

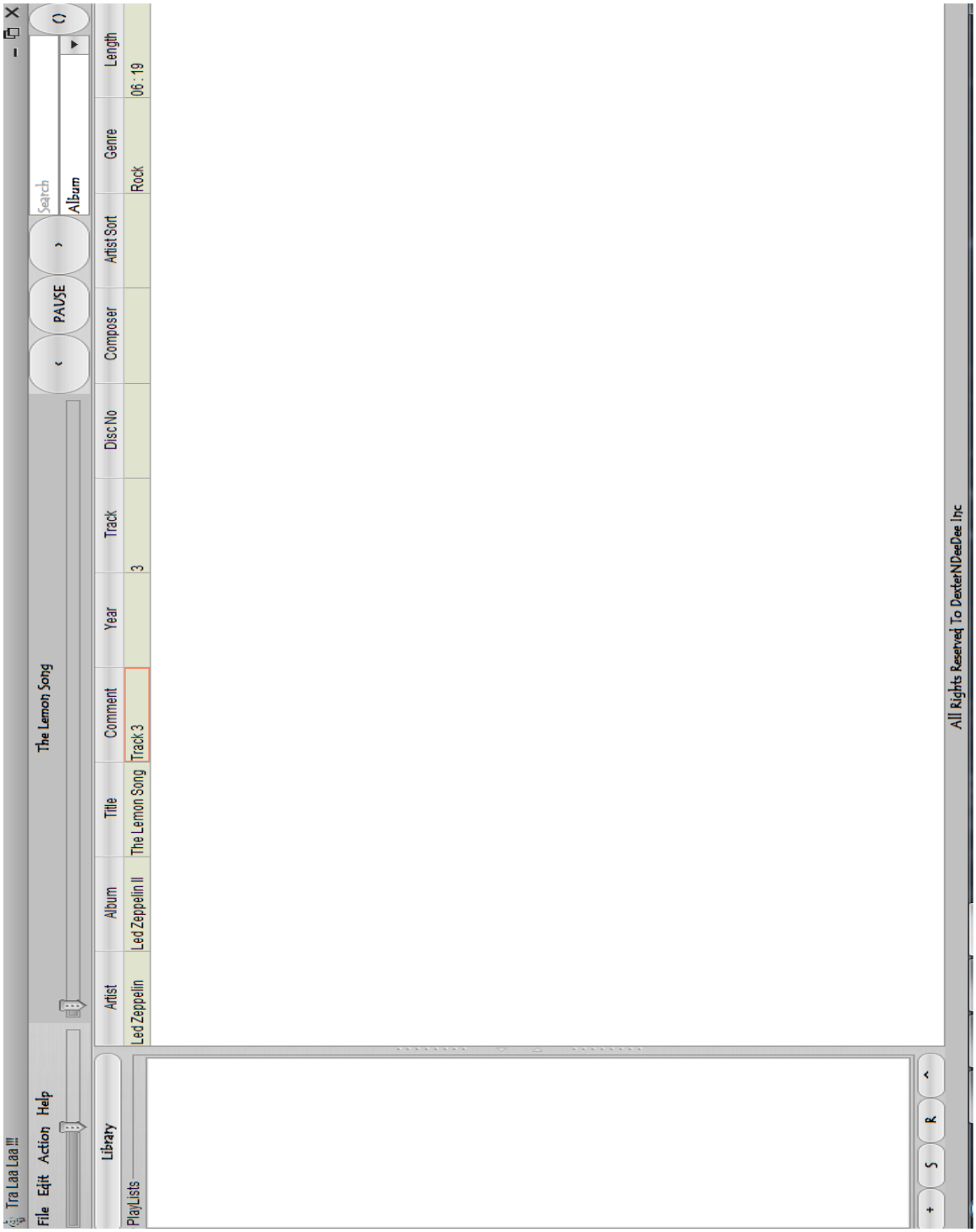

Fig 8.1 Album Table

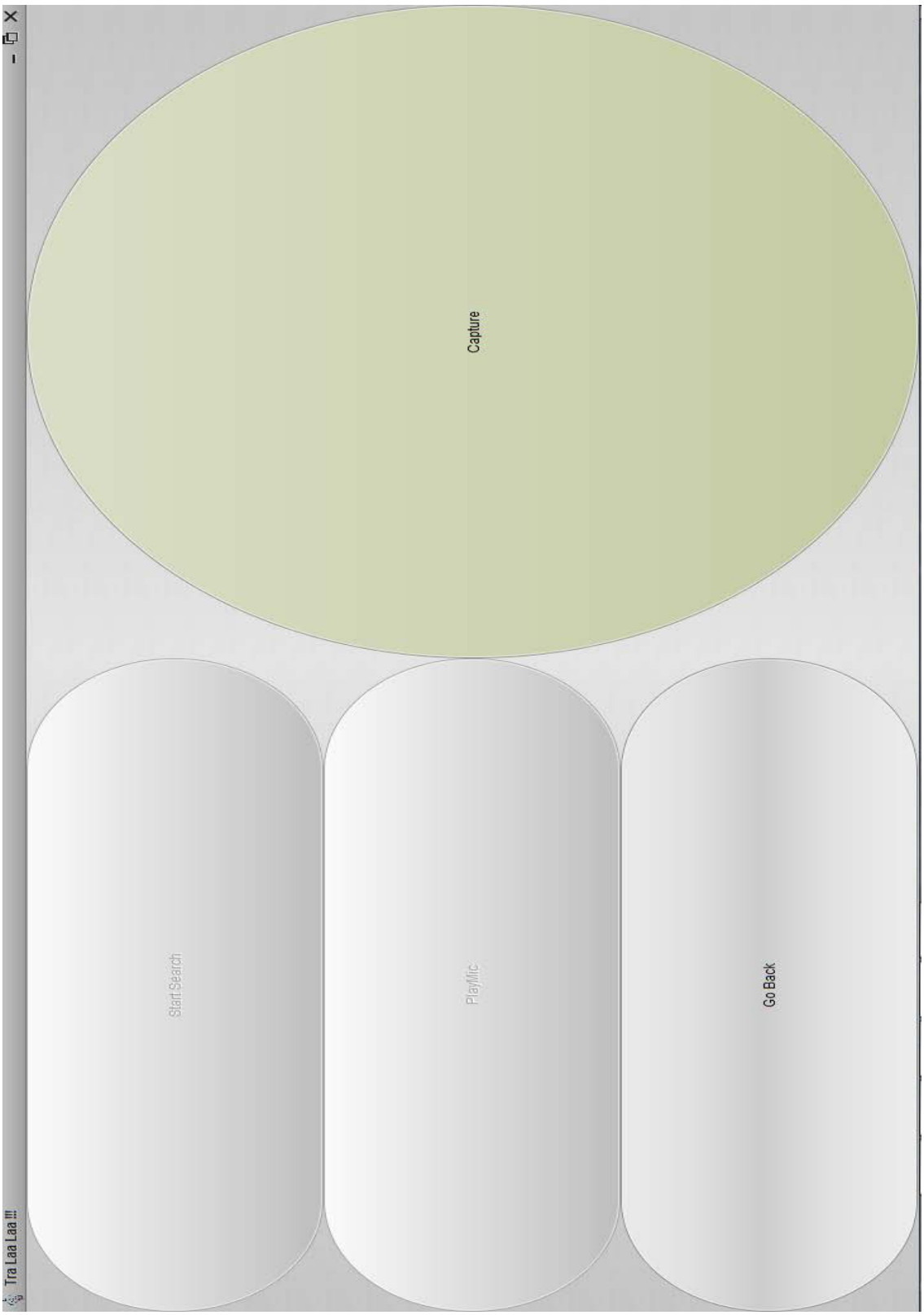

Fig 8.2 Humming Mode

Fig 8.3 After Humming

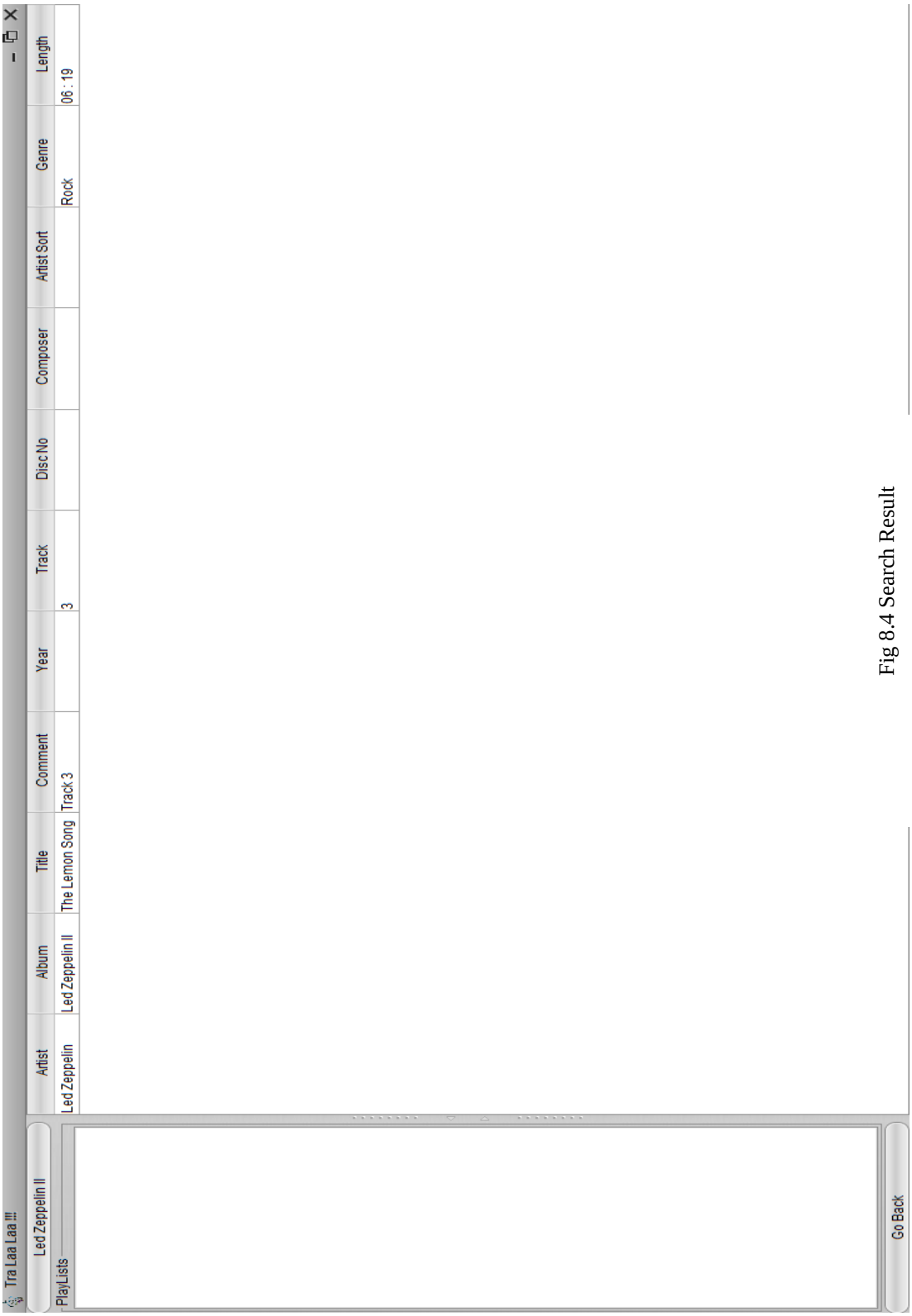

Fig 8.4 Search Result

Chapter 9
Conclusion and Future Work

CONCLUSION

- Songs in Shazam's database are all mpeg-7 files which means more quality and more accuracy than an mp3. Mp3 has just one stream with the lyrics and the instruments embedded.
- Shazam uses vocal tracks for matching humming. Their database has a separate copy of the vocal track for each song.
- Matching with vocal tracks is not a feasible task for us sincec we use the user's database.
- We have created an application to play and enjoy music , as well as identify it by its tune itself
- The domain is relatively new but there is a wide range of improvement
- With minimal hardware and software requirements, we have implemented an app par with any other app that is there on the market.
- In future , we aim to extend it to web and other environments.

FUTURE WORK

- The Application can be extended over the web and mobile platforms.
- The Application can be improved on the noise reduction scales.
- Humming mode can be developed to take in actual Humming

Chapter 10

Bibliography

BIBLIOGRAPHY

[1]An *Industrial-Strength Audio Search Algorithm* Avery Li-Chun Wang, avery@shazamteam.com ,Shazam Entertainment, Ltd. USA: 2925 Ross Road Palo Alto, CA 94303 United Kingdom: 375 Kensington High Street 4th Floor Block F London W14 8Q.

[2] *Robust Sound Modeling for Song Detection in Broadcast Audio* Pedro Cano, Eloi Batlle, Harald Mayer and Helmut Neuschmied *1 Grup de Tecnologia Musical, Pompeu Fabra University, Barcelona, 08003, Spain 2 Institute of Information Systems, Joanneum Research, Graz, A-8010, Austria Correspondence should be addressed to corresponding author Pedro Cano (pcano@iua.upf.es)*

[3] Roy van rijn , who has written an article on sound matching in his website www. redcode.nl

[4] http://www-personal.ksu.edu For information on Fourier series.

[5] http://www.betterexplained.com For Fourier transforms and analysis.

[6] http://www.edx.org For detailed lectures on fast fourier transforms.