

Lab

MusicExplorer: Exploring the Space of Songs on your PC

Anitha Gonukula
Philipp Kuederli
Samuel Pasquier

Olga Goussevskaia
Michael Kuhn
Responsible assistants

Prof. Roger Wattenhofer
Distributed Computing Group
ETH Zurich

February 2008

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Short Summary

We developed a web application to explore a space of music in a user-friendly way. Our work is based on an 10 dimensional embedding. In this space nodes represent songs. Near songs are assumed to be similar. The main focus of this lab was the visualization of this space. An interactive map was chosen as visualization form. Two modes are proposed. One mode shows songs present in a very large database of song's metadata. The other mode shows songs present on the user machine. Some potential users could use this application to (re)-discover their personal music. Many promising features like automatic playlist generation, search of song were implemented in a user-friendly manner.

Contents

1	Introduction	1
1.1	Context	1
1.2	Task Description	1
2	Approach	3
2.1	User Music	3
2.2	Visualization	4
3	Scenarios	5
3.1	Local Mode	5
3.2	Remote Mode	6
3.3	Playlist Generation	6
4	Implementation	9
4.1	Architecture	9
4.2	Client-Server Task Repartition	9
4.2.1	Music-Collection Recognition	10
4.2.2	Playlist Generation	10
4.2.3	Map-Visualization	10
4.2.4	Song-Search	11
4.3	Visualization	11
4.3.1	Search for Song	11
4.3.2	Map of Songs	11
4.4	User Interface	12
4.5	Persistent Session	14
4.6	Metadata Matching	14
4.6.1	Realisation	14
4.7	DirectoryTree	15
4.7.1	Pros	16
4.7.2	Cons	16
5	Technologies	17
5.1	Hessian Binary Web Service Protocol	17

5.2	Persistent Session - JAXB	18
5.3	Multi-format Support - Jaudiotagger	18
6	Deployment	19
6.1	Signing JAR	19
6.2	Procedure	19
7	Future Work	21
7.1	Persistent Session Issues	21
7.2	Adapting Song's Popularity	21
7.3	Exploring the Different Dimensions	22
7.4	MusicBrainz Tag Cleaning	23
7.5	Real Remote Music	23
8	Experience Report	25
8.1	Experience Report Anitha	25
8.1.1	Contribution	25
8.1.2	Team Work	25
8.1.3	Tools	25
8.1.4	Difficulties	26
8.1.5	General impression	26
8.2	Experience Report Philipp	26
8.2.1	Task	26
8.2.2	Team	27
8.2.3	Problems	27
8.2.4	Conclusion	27
8.3	Experience Report Samuel	28
8.3.1	Task	28
8.3.2	General opinion	28
	Bibliography	29

Chapter 1

Introduction

1.1 Context

The easy distribution of music over the Internet allows the users to collect a huge number of songs. This demands for a good way to manage the collection and also to expand the collection.

This management of music collections is normally done manually by the users using the file system of the computer and creating some kind of classification, mostly by style and artist. When a user wants to create a playlist and listen to this music, he/she either selects the songs by this classification or just randomly. In the former case the user will most likely choose the best-known songs. Therefore many songs of its whole collection will be listened to only very rarely. It would be better if the playlist generation could be automatized and the user only gives some very coarse inputs, what the music he wants to listen to should be like.

To expand the collection, a user either already knows what songs his collection is missing or he searches for new ones by browsing in the Internet. In both cases a user mainly expands his music with the best-known songs or with songs from the same artists or with very similar to those he/she already knows. Better proposals could be given to the user at the time he/she is browsing his/her music collection. The application could indicate which songs are the most suitable, the most similar songs, to the one he is just exploring. Based on the work of Lorenzi [1] we assume that both scenarios, managing and extending the music collection, could be supported by an application, if we use a similarity-measure for the songs. This similarity-measure is discussed by Lorenzi [1]. As described in 2.1 our work concentrated on the management of the music collections.

1.2 Task Description

Lorenzi [1] implemented a prototype which proofed the concept of automated playlist generation. The focus of this prototype however was not the usability. The task for this work is to reuse and extend the implementation of Lorenzi [1] to implement a fully functional application which can be used by a common user.

Chapter 2

Approach

The two major strategic decision we took are described in this chapter.

2.1 User Music

A point, which we found relevant, was the ability of some potential users to gain an insight in their collection of music. Today the collections of music are so large that it becomes critical to keep a global overview of them.

MP3 music was for sure a very fast growing area this last years. The combination of P2P systems and MP3 Players has lead to a tremendous success of digital audio files. But the management of this large amount of data had some difficulty to follow this success. Therefore we assumed that many people own a rather large disordered collection of music. Maybe some part of these collections may even have been forgotten with the time. Therefore it was important for us not to provide a service which adds even more music to these collections. Rather, the goal was to implement a service which helps the users to realize which music they already own. As described later this service was provided in the form of a graphical representation of the music.

We identified a direct consequence to realize this goal. The process of gathering the information about which music was present on the computer had to be largely automated. The assumption here is that the user has few (or no) knowledge about where his/her music is located on his/her computer. The few indications the user should provide must be very intuitive. Because this procedure might be annoying to repeat for each utilisation the notion of session was introduced. The user should be able to store the state he left the application and recover it later. (See 4.5)

Another challenge concerning the automated obtention of information was the poor quality of the metadata provided with the audio files. The distribution of digital music via P2P systems has lead to partial or incorrect tagging. Sometimes the metadata are completely missing, providing no clue about the artist or title. Musicbrainz [8] would apparently provide solutions to this problem 7.4. Unfortunately because we lack time we didn't further investigate this possibility. An attempt in the direction of relaxed matching of the metadata was realized 4.6. The idea was to match the artist name and the title not strictly with entry from the database but to allow some misspelling from these strings. Unfortunately the performance was not good enough too allow the deployment

of this technique.

2.2 Visualization

The most fundamental question of this lab was maybe: how should the similarity present in the space of music [1] be used in a clever way? The ability to tell which music has which properties like mood, energy, etc. first seemed us to be very appealing. But it came out that such feature were just not realistic, at least without additional information, to figure out. We assumed that if the space of music was organised. It was organised in very difficult way to describe with simple words. This organisation might even be completely subjective, which means that different persons might interpret the similarity in different manner. Because the space of music was build with some *average human subjectivity*, it might be difficult or maybe impossible to extract properties from it. But on the other hand giving directly back this *average subjectivity* to the users should make sense for the largest part of them. Visualization is a very straightforward way to provide a direct insight into this space. The importance of the visualization in this project was driven from this argument.

Visualising the space was obviously a complex task. The given embedding from [1] exhibits 10 dimensions. But our understanding of space is largely limited to three dimensions. Any representation of a space in more than three dimensions is not a trivial exercise. How to represent a 10 dimensional embedding with only 2-3 dimensions? This question is still not completely solved (see 7.3). The two first dimension of the embedding were chosen to represent the embedding.

It was decided that a good choice for the visualization was a map. Inspired from the *Google Map* [13] it would have to support the following feature: display some node for each song, a possibility to easily navigate on this map and the ability to support zoom. The third dimension (the depth or height dimension in analogy to the *Google Map*) was set to be the popularity of the different songs. So the start view should be the most popular songs and then based on this information one can decide to zoom in. If the related songs are close together, zooming in the map can be compared as looking at less popular, similar songs. More details about the realisation of the visualization can be found in 4.3.2.

Chapter 3

Scenarios

Our implementation supports two scenarios for the user. The first scenario allows the user to explore his/her own music collection on his local computer. He can further generate a playlist and listen to it. The second scenarios gives the user the alternative to explore the music collection of the server which consists of almost 500,000 songs. The disadvantage is that, while the user can create playlists, he/she it cannot be listened to them because we are not allowed to publish any songs.

A combination of these two scenarios, a hybrid solution using local and remote information, could allow the user to explore his own local music collection but in parallel also show the songs from the server. This would give him proposals for songs which are missing in the collection but which could be suitable. A future work could take place in this area.

The distinction for these two scenarios is already done at the starting screen of the applet. There the user can choose between local or remote mode.

3.1 Local Mode

In the local mode, the user can look at his own music collection. To explore the music collection or to generate a playlist the client must know the coordinates of the songs which are locally on the computer. So the user must first select which songs are intended to be available. For this the user can select several directories from the local file system where he expects the music files to be stored. When he has chosen a set of directories he can start the visualization. This makes the application start scanning the chosen directories for music files. The metadata of the located music files is checked for the title and the artist information. After scanning a list of title-artist tuples representing the corresponding songs just located is sent to the server which tries to recognize these songs. Each song which is recognized gains its coordinates. These coordinates are then sent back to the client. With this information the client is now able to start the visualization automatically and to show all the songs of the local music collection which could gain their coordinates from the server. The user now can explore his own music on the map or search for his songs by entering the artist name or the song title or both. On the map he can declare the playlist in a coarse way and he also has the possibility to manually remove or add single songs to the playlist. At the end the user can listen to the playlist directly if the underlying operating system allows starting a music

player automatically, alternatively he/she can save the playlist on his/her file system to listen to it manually afterwards.

3.2 Remote Mode

As opposed to the local mode the user directly encounters the visualization without scanning any files because all the songs lie on the server and the corresponding coordinates are already available. Similar to the local mode the user can explore the music from the server on the map or search for any songs by entering the artist name or the song title or both. On the map he can declare the playlist in a coarse way and he also has the possibility to manually remove or add single songs to the playlist. Since the songs are not available as music files and we are also not allowed to publish these music files the playlists cannot be used further.

3.3 Playlist Generation

Playlist generation is a concrete application of the embedding of the world of music. The basics concerning Playlist Generator are described in [1]. The algorithms described there are still valid in the actual implementation. However the 2-dimensional map raised new questions. Should the playlist be calculated on the basis of the 2-dimensional visualization? Will the user be confused if the playlist is displayed in 2 dimensions but computed in 10 dimensions? Furthermore it was not obvious how the user should give input for the generation of the playlist.

The actual applet proposes the following facilities. On the map he can declare the playlist in a coarse way. Either a position or a song can be selected. He also has the possibility to manually remove or add single songs to the playlist. At the end, if the user is in local mode, he can listen to the playlist directly if the underlying operating system allows starting a music player automatically, or he can save the playlist on his file system to listen to it manually afterwards. The playlist is in the format *m3u* [14]. Option fields for the playlist generation are different artist, different song, the length (time-constraints/track-constraints [1]).

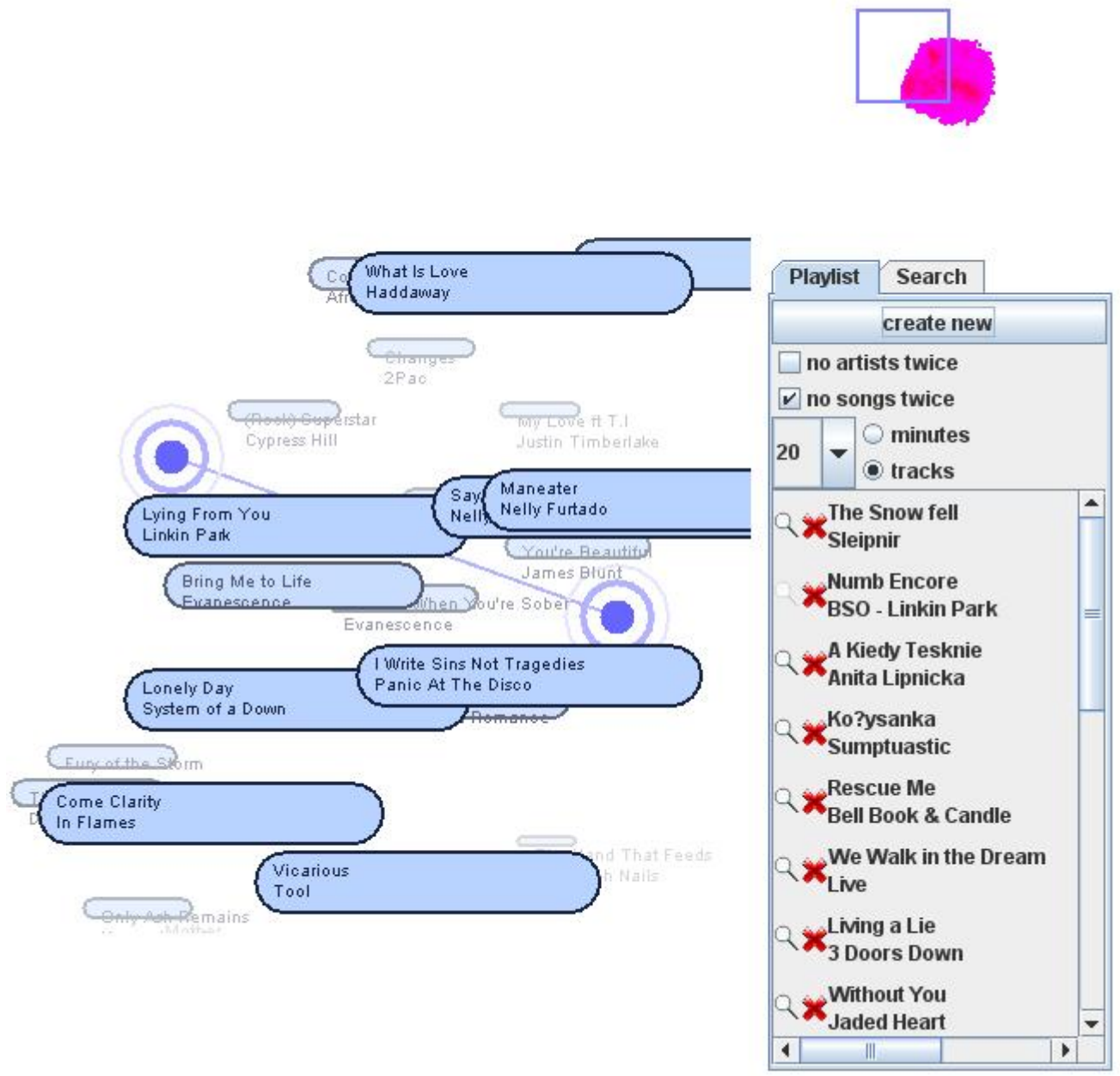


Figure 3.1: Screenshot Playlist

Chapter 4

Implementation

4.1 Architecture

The distribution of the resulting application was a major issue. The need to search for song files on the client was a demanding constraint. A fat client was required to perform this task. We considered two forms of client: applet or usual program. An applet was chosen because it did not require software distribution, avoiding the problem of backward compatibility and complex upgrading of the system. Please note that an applet can be turned without big difficulties into a usual program.

The resulting architecture of our system is a 3-tier architecture. (see 4.1)

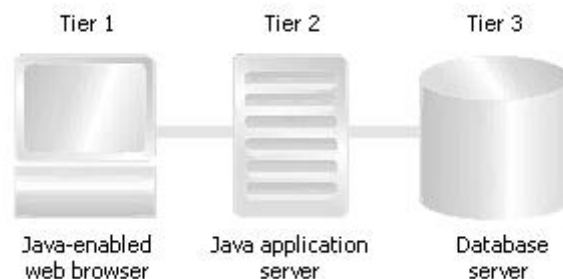


Figure 4.1: 3-Tier Architecture

4.2 Client-Server Task Repartition

Our client-server architecture has java applets as clients and a stateless web-service as a server; Stateless in the meaning that no per client sessions are stored on the server. The client-side is responsible for scanning the local filesystem for music files and for the visualization of the music collection. The server-side web-service provides four main functionalities and some trivial auxiliary functionalities. The main functionalities are responsible for music-collection recognition, playlist-generation, map-visualization and song-search.

4.2.1 Music-Collection Recognition

This functionality is only fallen back on when we are in local mode.

When the user wants to explore his own music which is stored locally on his computer the application scans the file-system for music files. Some of the music files contain artist and title information of the song in the metadata. For each song a new Song object is created and the metadata information is filled into the object. All these song objects are then put into a Songlibrary object. This object is then sent to the server which tries to recognize the songs based on the title and artist information. The recognized songs are filled with the corresponding coordinates. After this the server sends back a Songlibrary object.

A not yet satisfyingly solved problem is the recognition of the songs. (see 4.6 and 7.4).

4.2.2 Playlist Generation

Playlist Generation works in local mode and in remote mode. Even if the technique for the construction of the playlist, based on the algorithms described in [1], is the same in the local mode is as in the remote mode, there is some differences in the implementation.

In local mode, a KDTree is build after the recognition of the music collection 4.2.1. This KDTree contains direct references to the song item and is the basic component of a local embedding. With the help of the local KDTree the appropriate songs are chosen for the playlist. The start and end points of the playlist are either arbitrary positions in space, positions of concrete songs or any combination of both. The playlist can be generated either with a given number of songs or with given a total duration. All actions happen locally and no remote communication is needed.

In remote mode, communication between the client and the server is required. A KDTree is build on the server when the server starts. Each entry of the KDTree contains only information about the position of the songs and its ID. When a client wants to create a playlist, it sends the appropriate parameters (either ID of the song or specific position) to the server. Then the server creates a playlist with the KDTree. The server has to fill in the details of the songs with information he gets from the database. In the end, a Songlibrary object is sent back to the client. Note that in the remote mode the playlist can not be created for agiven a total duration, because no duration information is available for remote songs.

4.2.3 Map-Visualization

This functionality is only felt back on when we are in remote mode. In local mode the data structure which would be on the server in remote mode is calculated on the client so the server is not needed. For a potential hybrid solution with remote and local mode this functionality would be needed but perhaps the data structure on the server would need some adjustment.

The main problem of the visualization is the huge number of songs to display. We use a quadtree datastructure to perform an efficient visualization. This data structure is created on the client (the applet). The quadtree does some kind of space partitioning and each node of the quadtree corresponds to a rectangular cell on the map. The client then needs only to retrieve the songs for the visible cells. So it sends a query to the server which contains the paths in the quadtree to the

visible cells. The server then must return the songs of the cells. Obviously to do this the server must have available the same datastructure as it is built on the client. Fortunately the positions of the songs on the map are of global character meaning that the positions are the same for each client. We could say for example that for each client we would calculate the map of songs based on the client's own music collection. Then for a given song we would have different coordinates for different clients. But since this is not the case we are able to precompute the whole quadtree once on the server. Then each client can fetch the songs for a certain node respectively cell of the quadtree.

4.2.4 Song-Search

Song Search is relevant in local mode and in remote mode. The idea is that the user can query songs based on title and/or their artist name.

In local mode the song are searched over a local list, created after the music-collection recognition step 4.2.1.

In remote mode, the parameters are send to the server which then ask the database. Based on the answer of the database, the create a Songlibrary object and send it back to the client.

4.3 Visualization

The visualization allows the user to explore his/her songs and to create a playlist. The exploring is done either by navigation on the map of the songs or by directly searching for a song.

4.3.1 Search for Song

Searching for a song is implemented straightforward in the current version. The user can enter either a title of a song or an artist name or both. The database is searched for these strings and as a result the user receives a list of possible songs. The user can add a song to the playlist or zoom to it on the map.

4.3.2 Map of Songs

The map of the songs is displayed in a two-dimensional manner. Assume that each song has a two-dimensional position on the map. The problem arises that not all songs can be displayed on the screen at once since there are too many songs. This can be compared to a map of the world where you only display the names of the cities, states, countries and continents. There you have some kind of hierarchy. At the root you have the continents, then the countries, the states and at the city level you order the importance by the size of the city. So if you want to explore the world, most likely you will start at the coarsest level where the continents are displayed. And then more and more you go into detail until you find a small city on the map. This gives the feeling of zooming into the map. Notice that at the city zoom level you only see a very small extract of the whole map.

The same approach we use for the map of the songs. Unfortunately the songs do not have a hierarchy. There is no obvious ordering. So we artificially have to introduce an order. In our

implementation we use the popularity of the songs as the order which means that at the coarsest zoom level you only see the most popular songs. If we assume that all songs inside a region have some similarity it is best to display the most popular of them as a representative for all these songs.

We introduce this kind of zooming since too many songs on the screen would result in a mess and it is simply not possible to draw thousands of songs on one screen. Another reason is the performance. In Java you should not draw more than several hundred objects. So we must ensure that we have an upper limit of songs displayed on the screen.

Assume we have a viewport which represents the rectangular area of the map which is currently visible on the screen. Moving the viewport on the map simply means that you see another region of the map. Resizing the viewport results in the feeling of zooming in and out where a large viewport represents a small zooming level.

We declare to display never more than a certain number of songs in a given viewport. If there are more songs inside the viewport we only display the most popular songs. The less popular songs will become visible if the viewport gets smaller (zooming into the map) since some of the more popular songs will fall outside the viewport. Unfortunately a bad effect arises when instead of zooming the user moves the current viewport. Assume you have the maximum number of songs on the screen. Now the user moves the viewport in a way that some of the songs are still inside the viewport but also new songs come into. Since we must display the n most popular songs it is possible that several of the new songs have a higher popularity than the current songs. This would mean that one of the current songs suddenly disappears although it would still lie inside the viewport. This is not acceptable for the user experience and would look like a bug. One possible solution would be to slowly fade the song away. This however would require that we can predict when the new song enters the extract so that we start with the fading in time. Since we cannot predict the movement of the user this seems not to be trivial and we try another approach.

Our approach is based on a quadtree data structure which partitions the map into rectangular cells. The datastructure is precomputed on the server so we do not have any performance problems. The root node of the quadtree represents a cell which encloses all songs on the map. Then a child node partitions its parent's cell into four equal sized rectangular cells covering the whole parent cell. We feed the quadtree with the songs ordered by descending popularity. Each node checks if its according cell contains the song and if the maximum number of songs inside a viewport is not exceeded and if the density of the songs inside the cell is not too high. If these criterias are met we add the song to the cell otherwise we delegate the cell to the child nodes.

For a given viewport we now find all cells which are larger than the viewport but whose half size would be smaller than the viewport. For these cells we retrieve the according songs from the server to display them. To get a better feeling of zooming we let the less popular songs fade in from the background. For this we also load the songs of the visible cells' children and their grandchildren.

4.4 User Interface

The common architecture for a user interface is in most cases based on the MVC-principle. Unfortunately the MVC-principle is not a strong specification and often only small examples point out

the idea of the principle. This is why often in practice a modified version of MVC is used. The same holds for our application. The main idea behind MVC is the separation of the model and the view. The controller is nowadays often not distinct from the view and so it often falls away.

We mainly expect from our architecture that we have as few special cases as possible which do not fit into the architecture while still maintaining simple guidelines.

In our architecture we distinguish between data, models and views.

Data Data objects are mostly entity objects or objects with very few logic. Often they only provide some getter and setter methods.

Model Models work with the data objects meaning that models modify data objects. Each model object provides the registering of listeners. These listeners are notified of changes in the model. Take care that the data objects themselves do not support listeners. A model modifies the data objects and then notifies its listeners of its changes. The data objects are not able to fire any events. Model objects contain the business logic in our application. They provide methods which represent the functionality of the application.

View View objects worry about the presentation of the data contained in the models. So each view is initialized with the models of which it must display the data. The models of course must provide methods to access the data objects which are managed by the model. A view does not allow to register listeners. Each view exactly shows the data of its underlying models. So our views do not notify any listeners when an event happens in the view as MVC suggests. Our views directly call methods of the model objects. Mostly the methods provided by a model represent some basic functionality of the application.

Using these three object categories we are able to create a quite well structured application. A fourth category, the components, is introduced for clearness but it brings no new value into the architecture.

Component A component prepares the models and assigns them to the views or in most cases to one view. The components actually only group some models and some views together.

It is of great importance to see that the views are only refreshed if a model makes any changes to the data and afterwards fires a change event. Moreover it is possible that two different views are registered to two different models which in turn use the same data objects. Now it could happen that one model changes the data objects on order of its according view. Its own view is then updated but the other view still displays the old data. Our architecture allows this inconsistency although it would be better when both views are working on the same model. To get around these problems in a clear way we introduce a new kind of objects, the synchronizers.

Synchronizer We do not want a model to register itself to another model as a listener since this would extend the model with functionalities which would not belong to the main functionalities which would result in less clearness. So if one model must be informed about the changes of another model we introduce a synchronizer which delegates the changes to the other model.

Mostly the synchronizers objects are created in the components since the component knows all models which are needed for a certain view.

4.5 Persistent Session

The search of music can be a heavy time consuming task. For people accessing regularly MusicExplorer it might be inconvenient to spend each time some minutes for searching songs. After some investigations it turns out that the applet mechanism doesn't provide a straightforward way to implement caching. However because the applet already requires user privilege, it can also write files directly to the disk. The idea is that the user can choose if he wants to store some session about the song found. He also chooses the location where he wants to store these informations. The next time he wants to retrieve the session he will have to manually indicate the location.

The tool used for this service is described in 5.2

4.6 Metadata Matching

We extended our application so that it recognizes the song name and artist name though it is misspelled or mistyped by one or more characters, which is a drawback in the existing application.

4.6.1 Realisation

This feature is implemented using a special function in MySQL called *SOUNDEX* [15] and the Java Class *RefinedSoundex* [16]. *SOUNDEX* takes a string as an input and returns a soundex string matching the input string. Two strings that sound almost the same should have identical soundex strings. A standard soundex string is four characters long, but the *SOUNDEX* () function returns an arbitrarily long string. All non-alphabetic characters in the input string are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.

Ex: If we apply the *SOUNDEX* algorithm to the string (Artist name: Bob Sinclair), it returns the string "bob secular", since in the database, there exist no artist with the name "bob Sinclair".

Java class *RefinedSoundex*: Encodes a string into a Refined Soundex value. A refined soundex code is optimized for spell checking words. There is a system method named *difference* () in *RefinedSoundex*, which takes two strings as parameters and compares them and returns the difference as an integer value.

Ex: `soundex.difference(bob sinclar, bob sinclair)` returns 10 as difference.

Whereas, `soundex.difference(bob sinclar, bob sienclair)` returns 9.

4.6.1.1 Steps for implementation

It involves various steps to implement an approximate name matching.

1. Artistname without *SOUNDEX*: First it tries to get the `artist id` using original artist name without Soundex. If it is found, it tries to fetch the `song id` and `occ` using artist id and

original song name. If the songID and occ are not found, it will try with SOUNDEX in the SQL statement.

2. Artistname with SOUNDEX: If the artistname is not recognized in step 1, a second query using the SOUNDEX function in the SQL statement. If the SQL statement fetches only one artistname and artist id, it tries to look for songid and occ using artist id and original song name. But if the sql statement fetches more than one similar artist id and artist name, the data must be filtered using the RefinedSoundex java class. This compares the similarity between the original artist name and the artist names, which are fetched using the query with soundex statement and returns an int value as difference between them. From the int value, the percentage of similarity will be calculated.
3. If the percentage is greater than 95 for the artist name, it looks for the songid and song name using artist id and original song name.

4.6.1.2 Pros

1. Using this algorithm more songs are recognized compared with earlier
2. From the tested songs, it is hundred percent sure that it doesn't fetch the wrong artist and song name

4.6.1.3 Cons

1. This algorithm reduced performance as it involves lot of instructions.
2. It takes lot of time for the results since in step 2, SQL statement fetches more than one artist name.
3. SOUNDEX and RefinedSoundex, currently implemented, are intended to work well with strings that are in the English language only. Strings in other languages may not produce reliable results. These functions are not guaranteed to provide consistent results with strings that use multi-byte character sets, including utf-8.

Because of the performance issue this approach was not deployed in the final release.

4.7 DirectoryTree

The selection of the music's emplacement should be as intuitive as possible. A DirectoryTree seemed us to be a good solution. This section describes the implementation of a DirectoryTree.

- A DirectoryTree is nothing but the extension of JTree with the possibility to select or deselect the nodes of JTree.
- Each node of the JTree contains a CheckBox.
- When a node's checkbox is selected, all its descendants should also get checked and vice versa.

- When all descendants of a node are not checked but some of them are checked (i.e, partial selection) then the node should be checked with a gray tick.
- When all children of a node are unselected, the node should be unselected.

Directory Tree displays all the folders of selected path in the tree structure, when the GUI comes up and the file system is read. This enables the user to look at the folders in the tree view and select or deselect the folders even after file system selection.

In the current Java API, there exists no such a tree. Because of that we had to implement every class on our own. It has got totally four important java classes.

1. DirectoryTree, which is extended from JTree.
2. DirectoryTreeNodeSelectionListener, is an extension of CheckBox and implements MouseListener is used to listen to mouse actions.
3. DirectoryMutableTreeNode is extended from DefaultMutableTreeNode
4. TreeNodeWithCheckboxRenderer extends DefaultTreeCellRenderer

4.7.1 Pros

1. It looks user friendly
2. It is convenient for the user to select music

4.7.2 Cons

1. It consumed a lot of time to implement
2. It has got some unknown bugs

Chapter 5

Technologies

5.1 Hessian Binary Web Service Protocol

Based on the 3-Tier architecture, the applet and the server needed a way to communicate. This communication had to be efficient (possibly large amount of data to transmit), flexible (for extensibility and reusability of project) and light (one part sits on the client side). First we tried to solve it with file upload and serialization of object in XML. But this solution was fastidious and not flexible at all. After some discussion with expert in the field it turns out that *Hessian Binary Web Service Protocol* [2] might be an excellent solution. The Hessian's implementation is available in at least 8 programming languages (including Flash and PHP). The Hessian protocol is relatively small and therefore well-suited for clients like applets. Our experience with this web service was extremely positive. We view the usability of this web service protocol as optimal. The separation between client and server becomes almost transparent. No problem concerning performance was encountered.

Here is the interface of the Web Service.

```
public Songlibrary getFromLibrary(Songlibrary songlibrary);
public RangeReturn getFromRange(double[] UpperRange, double[] LowerRange, int size,
double depth);
public Songlibrary getSongArtistTitle(String title, String artist);
public Songlibrary getPlaylist(int StartSongId, int EndSongId, double [] StartPosition,
double [] EndPosition, int length, boolean AllowDups, boolean DifferentArtists);
public Songlibrary getSongofArtist (String artist);
public Songlibrary getSongwithTitle(String title);
public List<ClientEntry> getClientEntries(List<int []> path);
public double[] [] getMapPixels();
public ViewRectangle zoomToSong(int songId, double x, double y);
public ViewRectangle getEnvelopingViewRectangle();
```

5.2 Persistent Session - JAXB

JAXB 2.0 [3] was used to perform the storage of user session 4.5. It provides an automatic method to serialize Java Objects in XML and to construct Java Object from XML files. Using an Xml Schema we directly generated Java classes using XJC.

5.3 Multi-format Support - Jaudiotagger

In the previous web application only the mp3 music format was supported. The new applets is also able to access metadata from *m4a* [6] files. Itunes commonly produces music files with this extension. The library Jaudiotagger [7] was used to decode m4a metadata. This library could even support the Vorbis and Flac extension but this isn't exploited for the moment

Chapter 6

Deployment

6.1 Signing JAR

Usual applets are executed within a sandbox. A sandbox is a set of rules that are used when creating an applet that prevents certain functions when the applet is sent as part of a Web page. The applet created in this lab requires some privileges, like access to the local hard disk or establishing a remote connection, which are out of the sandbox. After user's permission, a signed applet is exactly given these privileges. A short introduction about how to sign applet can be found in [5].

6.2 Procedure

Here is the procedure to follow for the deployment of the web application.

1. Create a jar file with the client code
2. Include the needed library with the created jar file (set `Class-Path` to the corresponding files)
3. Sign the jar file
4. Include the signed applet in the html material (how to insert the applet tag is described in [4])
5. Pack server code, signed applets and html content in a WAR file

The repartition between client and server code is the following.

Client Code	Server Code	Shared
Package *.applet	Package *.init	Package *.webservice
Package *.libconv	Package *.servlet	Package *.middleware
		Package *.musicworld.library
		Package *.utils
		Package *.embedding

Table 6.1: Client/Server Code repartition.

Note: For building the client jar file the resource files in the directory `res` have to be included.

Chapter 7

Future Work

In this chapter we describe the future possibilities we identified in the area of MusicExplorer.

7.1 Persistent Session Issues

It might be useful to further investigate how the states containing user's information could be saved. For the moment only the songs found and successfully recognized are stored in the xml file on the client side. (see Section 4.5) Maybe some users might want to store more information like the playlist they generated, their most popular songs, etc. Introducing user account could be one possibility to enhance the usability. Moreover one has to consider that the user might use different computers. Information, like the location of the music, are probably more computer dependant than user dependant. Maybe more information, like the actual generated playlist, could also be stored in the user session.

7.2 Adapting Song's Popularity

For the moment the popularities of the songs are statically determined. This popularity was extracted from the column `occ` in the `tblVertices` table. The highest occurrence is 9490 and the lowest one is 1. Figure 7.1 is a plot of the lowest part of the distribution (from popularity 0 to 60).

The behaviour of this function gives valuable information. Obviously this function has a very steep slope. Only few songs have popularity higher than 500 and extremely few songs have popularity higher than 3000. But more than 60000 out 430000 songs have popularity 1. Almost 11000 songs have popularity 10. This means that a lot of songs will be at the same depth and will appear only with a strong zoom. This behaviour is appropriate because as the range (the considered square on the map) gets smaller the level of density of songs should grow to maintain the same number of songs on the map. Further work could try to optimise this density. But a more problematic issue is maybe the fact that songs with high popularity are mostly from the same style of music (principally rock), hiding to the user the expected grouping behaviour of the songs on the map. To counter this problem, a possible solution might be to weight the popularity of songs given their styles. Popularity of songs with less popular music styles could be increased.

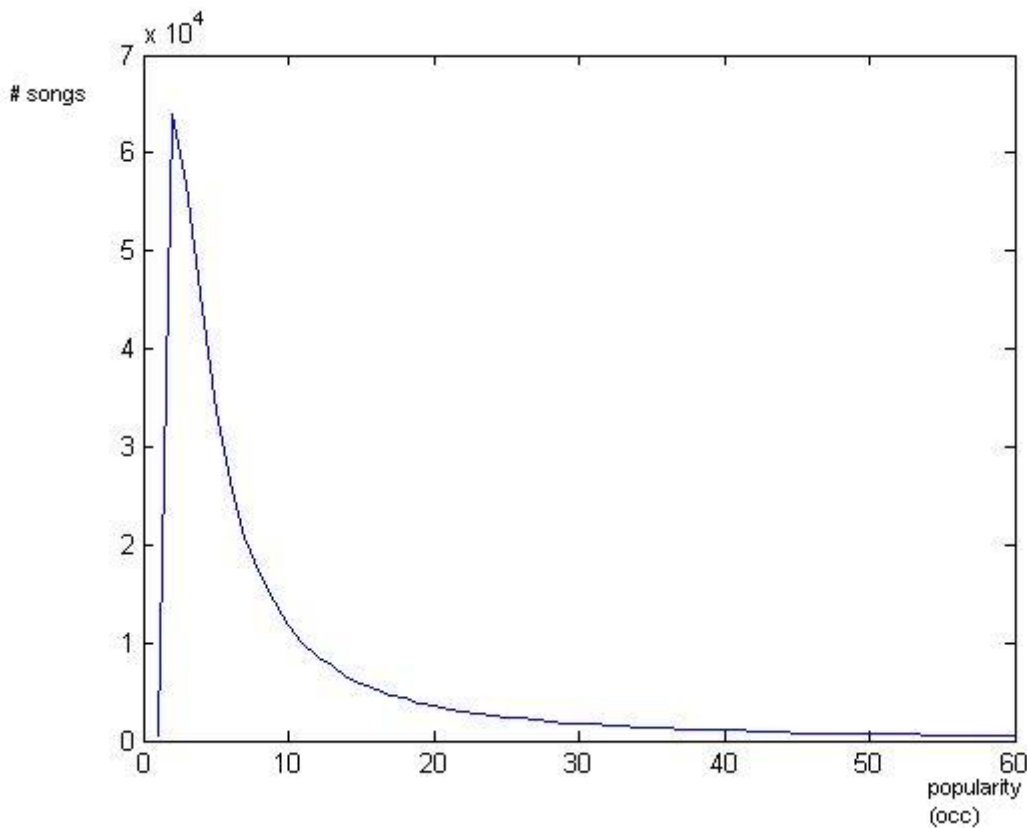


Figure 7.1: Distribution of the popularity

Notice that the scan of the local songs could produce, almost for free, rating about the popularity of songs. This could be used to flatten the density and to weight the popularity to the preferences of the user of MusicExplorer. Furthermore song popularities may change over time. So it would also be nice to adapt the popularity of songs dynamically given some static distribution.

7.3 Exploring the Different Dimensions

A study in [1] showed that the smoothness of the playlists increases while incrementing the number of dimensions. The actual embedding has 10 dimensions. Of course no straightforward representation of a 10 dimensional space exists. The map created can encode 3 dimensions: the x,y coordinates and the depth (i.e popularity /zoomlevel). It was assumed that this map could be developed without definitely setting the coordinates. The depth was set as the popularity. The x,y coordinates are currently the 2 first dimension of the embedding. Of course this might be easily changed.

The problematic is how to optimally transform a 10 dimensional space in a 2 dimensional one. We tried to find out which dimensions were containing the most valuable informations. It seemed that the 2 first dimensions of the embedding were preserving best the distances between the different songs. Unfortunately we lacked time to provide guarantee about this fact.

In general it would be useful to investigate how this many dimensions behave on the map. It could be interesting to research if some dimensions make visibly more sense than others. If so, does

this *impression of similarity* depend on the person who watches it? Maybe these answers are tightly related to our perception of music. We consider that this new tool could allow some innovative research in the field of the psychology of music. It would also be good to know if some special projection of the embedding in 2D could increase the quality of the map.

7.4 MusicBrainz Tag Cleaning

MusicBrainz [8] is a user-maintained community music metadatabase. Information contained in this database is, among others, the artist name and the release title. Some tagging application, such as *PicardTagger* [9], make use of these data. They allow the user to automatically identify and label music files. Here are the different possibilities identified as promising for MusicExplorer. MusicBrainz provides a webservice [10] which can be used to query musicbrainz data from any application which is able to parse XML. This could be a first step toward metadata cleaning. Unfortunately this simple solution might be infeasible for large amount of data because MusicBrainz state the following: "All users of the XML web service must ensure that each of their client applications never make more than ONE web service call per second."

An alternative would be to setup an own copy of the MusicBrainz server, and import a database dump [11]. Furthermore Musicbrainz provide a way to recognize a track even if no metadata are provided at all. (see [12] for more details)

We believe that this approach could solve many problems related to the correct tagging of metadata and could dramatically increase the percentage of songs recognized.

7.5 Real Remote Music

Even if we insisted on the importance to use songs already owned by the user, an additional service could try to make songs, not owned by the user, remotely available. Apparently the main issues of such a system would be legal restrictions, and possibly performance. A solution might be to allow the user to listen for free to music snippets. Complete songs might be charged. A peer-to-peer system could be another approach to this problem.

Chapter 8

Experience Report

8.1 Experience Report Anitha

8.1.1 Contribution

I contributed following tasks to the project, which are already explained briefly in the document.

- Deciding the architecture
- To integrate applet in the html pages and make it to be running in web browser by signing it.
- Designing GUI with swing components and background functionality.
- Developing algorithm for Metadata Similarity or matching.
- Implementing navigation between scanning and visualising.
- Deployment of server side files on tomcat.

8.1.2 Team Work

In the beginning we were four people in team, unfortunately from the middle of the semester only three people had to work on the project. Nevertheless, it was great and pleasant to work in such a team. I enjoyed working with them, though I haven't known them before beginning of the lab. Every one of us has his/her strengths, weakness and luckily from the strengths I could profit more while I wasn't disturbed by their weakness. Like every team we also had different ideas and opinions. Nonetheless everybody has contributed his/her tasks good to the project and at the end could make it to be working.

8.1.3 Tools

As a development environment we all used Eclipse, since everyone in our team knows eclipse well. As I had to work more with GUI and layouts, I also tried other environments like Netbeans and IntelliJ. In fact, I didn't have much success with them. For version control CVS is used, which I think is absolutely indispensable for every project with more than one developer. Moreover, due to its plug-in support in the IDE makes easier, convenient and problem less to work in the group.

8.1.4 Difficulties

We wasted more than two weeks deciding the architecture to precede the project, since we didn't get much experience in developing client side applications. At one point we were also not sure whether to use flash or java applets or some other technology to run application at client side.

In the beginning we didn't discuss in the project meetings about look and feel of the GUI. I designed some html pages and GUI for selection music and scanning with swing components for more than four weeks and showed in the meeting, which was ok, but at the end it was suggested to be improved a lot, where we didn't have lot of time. I think it would have been better, if I would have known in the beginning itself that how good it should look like.

I invested lot of time for JTree with Checkboxes to view the selected music path as tree structure to enable the user to select or deselect the folders and it was also complicated. Nevertheless, it looks too simple and it's not believable that it took so much time. I regret that I invested so much time and also learnt lesson, which is really useful for the future not to repeat such mistakes.

One more point is about Metadata Similarity (see 4.6), I developed algorithm for metadata matching. That means, application recognizes artist and song name though they are misspelled or mistyped by one or more characters. It worked very well, when I tried with different number of songs. I am happy that, It can also recognize more songs than earlier. But what I realised at the end is the performance problem. If the application without this algorithm takes five minutes to scan the music, with this algorithm takes more than ten minutes, since this algorithm involves lot of instructions. Improving performance could be the future work.

8.1.5 General impression

I decided to take this lab two weeks after beginning the semester. Luckily, since the project hasn't yet been started, I had a possibility to join the group.

I liked the time being spent on the project, though project took more time than expected. I could able to learn technologies like java Applets, CSS, and new Java API's and deployment of java applets and server side files on tomcat. I had a possibility to study and understand the already existing master thesis *Similarity Measures in the World of Music* [1] deeply and I found the work and division of project interesting. I am also happy with the results we got at the end, even though we got performance problem for the feature Metadata Similarity , described in 4.6. I would also appreciate weekly team meetings with the supervisors. Thereby we could discuss the state and modifications of project and exchange the ideas. For that I would like to thank Olga and Michael for their patience, support and excellent supervision.

8.2 Experience Report Philipp

8.2.1 Task

The whole idea behind MusicExplorer sounds interesting when it is presented to you. The first question which arises is always how the similarity between the songs could be measured. The answer on this question is quite fuzzy and so at the beginning it was a bit frustrating that we were

not yet able to say if the result of our application will be a success. If I would do such a project in my freetime this would be a great challenge to try out but if I do it as a Lab work the uncertainty brings some doubts with it.

I was responsible for the visualization of the map. Unfortunately during the Lab I also worked on the other parts of the GUI so that at the end I could not decide on what I should concentrate.

8.2.2 Team

The team aspect of the Lab makes the whole work more interesting but also can become a quite critical part for a successful project. If we would work in a company we would be a project team and would work tight together. For us this was not possible since we had totally different timetables and also we had not the same workload per week in mind. Further although one could expect a student to speak English fluently I sometimes could not express myself as good as I hoped what made our discussions difficult.

Our big mistake was the splitting into two groups. It is a well known principle that the GUI is done independently of the Business Logic. But in our case we are simply not two different teams and the splitting just resulted in the loss of our knowledge exchange. At the end we were in a situation where we were not able to help each other since each of us only knew its own code.

8.2.3 Problems

I personally had most problems with the MVC principle. The theory behind this principle sounds quite easy. But often in practice one would like to have a MVC "cookbook" besides. At the end I found my own variation which was quite consistent in the whole application. But it took me a lot of time to find out how such an application is designed so that a clear structure is possible.

If you want to make the application robust and responsive you must make use of threading. Unfortunately the asynchronous state transitions make the application very unpredictable which leads to bugs.

The whole visualization of the map was quite time consuming. We spent a lot of time about discussing problems in the group. The solution at the end was simple but not very memory efficient. The revenue and expense did not match in our project. I think we were quite inefficient. Most likely the mutual motivation was missing in our team. This led to many pledges from my side concerning deadlines which I could not hold. It was sometimes frustrating that I was the only one which was able to extend the application because I was responsible for the GUI from the beginning. At the same time I was refusing any help because I thought that it would be easier to do it myself than to introduce another team member into the whole code from the GUI.

8.2.4 Conclusion

A group work is really difficult to organize and can decrease the efficiency of the team members. Myself I would have appreciated a team member which could have helped me with my work.

8.3 Experience Report Samuel

8.3.1 Task

In our first task repartition I was belonging to the group responsible for every others aspects than the map itself. Additionally I spend much time trying to facilitate the communication between the different actors. Among other things, I was in charge for the following works: Implementing support for the m4a formats, Communication Applet-Server, Persistent Session, Search of Song, Playlist Generation, Deployment of the Web Application, Design of the WebPage, Design of the report, Slides of the Presentation.

In general I was active in many different areas of our project and I could therefore keep a good overview of what we realized.

8.3.2 General opinion

I view this lab as a surprising and general positive experience. Surprising because the difficulties and the challenges were not present at the places I would have expect them to be. Fortunately, after some patience, work and collaboration solutions could be found.

The first point I would like to emphasize was the difficulty we had to define the final design and goal of this application. Many weeks had already gone before we really started to have a good idea of what the system could look like. In my opinion, the lack of definition led afterward to many problems. For example we started to develop many small applications which then had to be merged together or even aborted. Moreover this precious time of the first weeks was missing us at the end of the semester. This is maybe my only critic toward the two assistants which supervised us because I found them excellent concerning the rest. I would have like them to help us focusing on some final goal rather than proposing (too) many different ideas. This is maybe more important in a lab than is some semester/master because at the end we have to deliver a product and not only ideas. Hopefully in the end we were able to find out concrete solutions, even if they might be not perfect, which could help to better understand and use this space of music. The second aspect I found delicate was our collaboration. We started with a group of 4 people with heterogeneous background, knowledge and availability. This has caused, I think, a big administrative overhead and sometimes has lead to an overall diminished motivation. I often felt, it was often difficult to estimate people's capabilities, availabilities and motivation. The fact, that one person left the group during the semester, was also a destabilizing event. The coordination of this group was for me a challenging task and I could acquire some good experience in this domain.

Working with people with different point of view was a good aspect. It many times helped me to approach the problems in different manner.

I finish this lab with an impression of enriching experience.

Bibliography

- [1] Similarity Measures in the World of Music
Master Thesis, Michael Lorenzi
- [2] Hessian Binary Web Service Protocol
<http://hessian.caucho.com/>
- [3] JAXB Reference Implementation Project
<https://jaxb.dev.java.net/>
- [4] How to Include Applets in HTML
<http://java.sun.com/j2se/1.4.2/docs/guide/misc/applet.html>
- [5] How to Sign Applets Using RSA-Signed Certificates
http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/rsa_signing.html
- [6] .m4a Format
<http://en.wikipedia.org/wiki/M4a>
- [7] Jaudiotagger Audio Tagging library
<http://www.jthink.net/jaudiotagger/>
- [8] Musicbrainz Project
<http://musicbrainz.org/>
- [9] Picard, a MusicBrainz Tagger
<http://musicbrainz.org/doc/PicardTagger>
- [10] Musicbrainz Webservice
<http://musicbrainz.org/doc/XMLWebService>
- [11] Musicbrainz Server
<http://musicbrainz.org/doc/MusicBrainzServerSetup>
- [12] Musicbrainz PUID
<http://musicbrainz.org/doc/HowPUIDsWork>

[13] Google Map

<http://maps.google.ch/>

[14] M3U File Format

<http://en.wikipedia.org/wiki/M3u>

[15] Soundex SQL

<http://msdn2.microsoft.com/en-us/library/ms187384.aspx>

[16] Refinedsoundex Function

<http://commons.apache.org/codec/apidocs/org/apache/commons/codec/language/RefinedSou>