Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

JUKEFOX core extraction

Semester Thesis

Sebastian Wendland, Samuel Zihlmann

wendlans@ee.ethz.ch, samuezih@ee.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Samuel Welten, Tobias Langner
Prof. Dr. Roger Wattenhofer

March 19, 2012

Acknowledgements

We would like to thank our advisors, Samuel Welten and Tobias Langner, for the great support during the semester. Furthermore, we want to thank Prof. Dr. Roger Wattenhofer for giving us the opportunity to realise this thesis at the DISCO department.

Abstract

The main goal of this semester thesis was to extract the core functions of the JUKEFOX music player. This included the decision of what the core functions are and how to separate them from the rest of the source code. After successful extraction and the port to native Java, JUKEFOX can be used on platforms other than Android.
Other parts of this project included to rewrite some of the code structure in a more readable form and to support the new cross-platform abilities.
This paper gives an overview of the new JUKEFOX code structure as well as some insights into the current state of JUKEFOX and finally some theoretical considerations that influenced the architectural decisions.

Keywords: JUKEFOX, Android, Java, MVC

Contents

Chapter 1

Introduction

1.1 Motivation

On the project page JUKEFOX is described as follows:

> JUKEFOX is an Android mobile application designed to change your music experience. At the first glance, it looks like an ordinary music player. A closer inspection, however, reveals some striking differences. JUKEFOX is based on the revolutionary concept of a music similarity map that facilitates some entirely new ways to interact with your music collection.[1]

Over 100'000 downloads and the high user rating demonstrate that JUKEFOX convinces many customers of the Android Market. One of JUKEFOX's major features is its "smart shuffle" mode, which tries to determine the perfect next song to play.
As a user of the outstanding features of JUKEFOX you will at some point be faced with the question whether you can use this player on a different platform, such as a personal computer.
Previously, JUKEFOX was a pure Android application and therefore depended heavily on the Android API, which makes porting of the source code difficult. This project's goal was to extract all the core functions of this music player. These functions had to be restructured and finally adapted to ensure that they continue to work both on Android as well as other platforms that support native Java.
Furthermore, this music player has been developed by many different people and teams, so over time the structure of the project became more and more unorganized. To counteract this trend, not only the core features were extracted and ported to native Java, but they were also rearranged in a new and more structured way.

[1]Project page http://www.jukefox.org

1.2 Objectives

The main objective of this project is to extract the main functions of JUKEFOX code and cut it free from the underlying Android system to get a cross-platform application. This has to be done without changing its functionality.
The other objective is to clean up the code structure to a more readable form.

1.3 Task description

The easiest way to describe the assignment of this project is to quote the official informal description from our advisors:

> Imagine how wonderful it would be if your music player automatically found out which music you want to listen to right now and just played such songs? JUKEFOX, a versatile music player developed in our group, is capable of this wonderful feature - it is called smart shuffle. Moreover, it can display all your music in a 2-dimensional map according to the similarity of your songs and has many other novel features (see `www.jukefox.org` for more information).
> However, our JUKEFOX (so far) only occurs in the sealed habitat of the Android platform for mobile phones. We would like to release JUKEFOX from its prison to let it roam the wilderness of other platforms.
> JUKEFOX has been developed in the Java programming language, specifically targeted for the use on Android mobile phones. Unfortunately, the underlying API is not abstracted in a way that allows to use it on a regular home computer. The main goal of this project is to extract JUKEFOX's powerful music similarity API to obtain a stand-alone version of the music database, which can easily be used in other players or in plug-ins for other players. A second goal is to use the extracted application to implement a plug-in for your most favourite music player.[2]

[2]Motivation and Informal Description for *Software Engineering: Help Our Jukefox To Conquer New Habitats!*, `http://disco.ethz.ch/theses/hs11/jukefoxAPI_Assignment.pdf`

Chapter 2
Background

2.1 Android

Android applications are written in the Java programming language, but this does not mean that they run on any Java Virtual Machine. Google's Virtual Machine Dalvik is a lightweight version of the Java VM and is designed to use less memory and to be executed in parallel to give every application its own Virtual Machine. Besides basic core libraries of Java, Android provides a set of additional libraries including special functionality for mobile devices.

2.1.1 API

Via the application framework the developers have access to a set of C/C++ libraries, for example a SQLite database or OpenGL.
An Android application has four different components: *Activities*, *Services*, *Content providers* and *Broadcast receivers*. Each of them provides a point through which the system can enter the application. *Activities* are front-end components with a user interface in contrast to *Services*, which run in the background to perform long-running operations such as data-fetching over the network or playing music. *Content Providers* are the components for sharing application data with other applications and are for example used in the music library framework. Finally, *Broadcast Receivers* are responsible for reacting to different broadcast announcements like "low battery" messages. All those components are implemented as a subclass of Java base classes provided with the Android API. The communication between those components is handled through messages called *Intents*. Different specialized data types are used, for example `ContentValues` are handover objects for different interfaces and only a `ContentResolver` can process them.[1] [2]

[1] http://developer.android.com/guide/topics/fundamentals.html
[2] http://en.androidwiki.com/wiki/

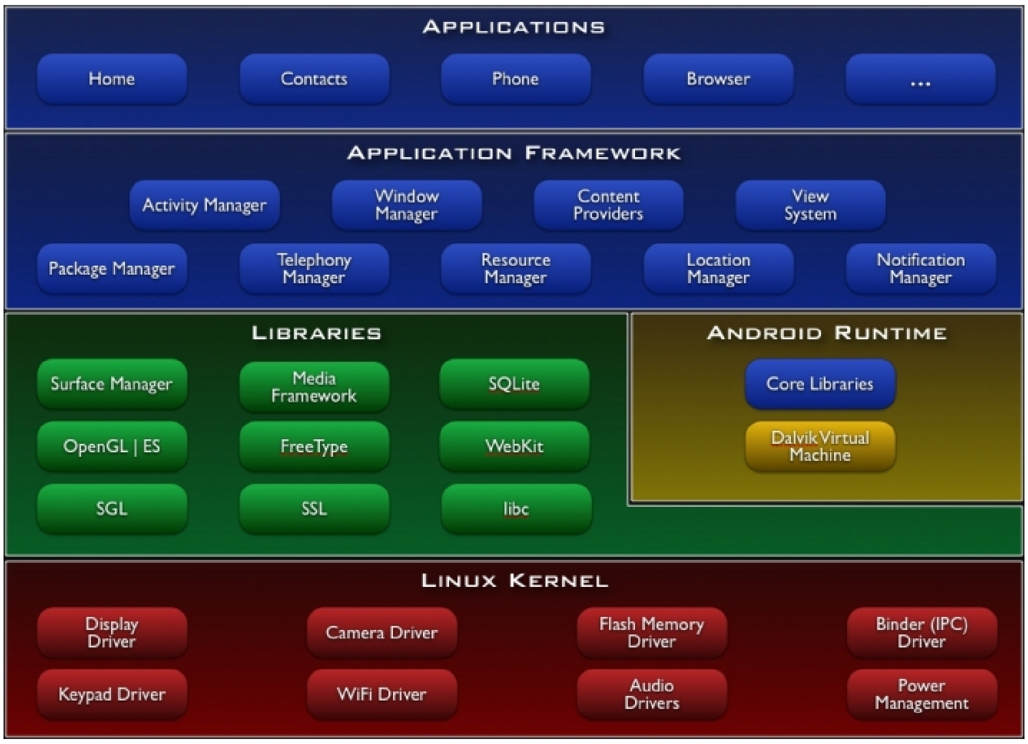


Figure 2.1: Android architecture

2.1.2 Porting from Android to native Java

To port an Android application to native Java we need to get rid of the Android API dependencies. Focusing on the core function of an application and ignoring the user interface, most API dependencies can be found on the data access level, for example an interface to a database, a media library or a network. The Android API provides a lot of helper classes for those data interfaces.
The dependency on those classes can be eliminated by using helper classes based on the standard Java libraries. Another way to make the code Android independent are self implemented interfaces, so called wrappers, who forwards those functions to the corresponding helper class. These helper classes are the specific implementation of those interfaces for their related platform.
All used data types have to be provided by the Java libraries and may need to be translated back to Android specified data types in certain interfaces.

2.2 MVC

The idea behind a pattern oriented software architecture is to get a standard design approach. There exist many patterns to achieve different structural goals.

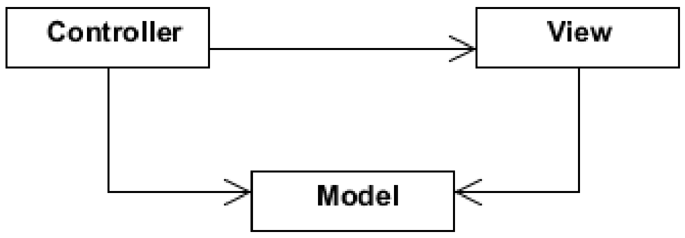


Figure 2.2: MVC concept

The **Model View Controller** (MVC) design pattern has the goal of decoupling the user interface, the underlying business logic and the model domain. This pattern has, as shown in figure 2.2, three basic sections which are separated from each other.[3] [4]

- ***Model*** This is a collection of classes which provide a data structure of some specified type. Further, one can also implement some *Model* internal business logic, for example a conversion of different data types.
- ***View*** This section is used to display the output of the application, but is not responsible for the logical calculations. A function call from the *View* will be forwarded to the corresponding controller. The *View* has access to the *Model* and will usually be notified, if there are changes of the data in the *Model*.
- ***Controller*** These classes manage and control all important processes, interpret user interface inputs and inform the *Model* and the *View* to change their states based events.

Beside the MVC pattern there exist some similar patterns for example the **Model View ViewModel** (MVVM). The difference to MVC is that the *View* in this pattern is completely separated from the model domain and vice versa, as the *View* only knows the *ViewModel* and the *ViewModel* again only knows the *Model*. This architectural pattern is used to eliminate all the logical code of the *View* because the displayed information is directly selected from the *View-Model* without the need for modification. This improves the cooperation between designers and programmers compared to other architectures.[5]

[3] `http://msdn.microsoft.com/en-us/library/ff649643.aspx`
[4] `http://en.wikipedia.org/wiki/Model-view-controller`
[5] `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`

Chapter 3

Implementation

3.1 Main goals

Before the implementation of the previously specified designs, the most important rules on which all future implementation decisions will be based had to be defined. These guidelines are the result of considering what the user needs, in which context something is used and the current state of the project.

1. **All core functions should run independently from the operating system**

   The primary and most important goal is to secure the independence of all core functions from the underlying operating system as much as possible. This can be done by splitting all problematic parts into two or more independent implementations which will run under one or more platforms or rewriting existing code so it can be executed on all platforms in the same way. The rule here is simple: It is advantageous to have only one function defined for all platforms instead of different ones because it allows easier maintenance.

2. **Reduce performance penalty on Android systems**

   The second goal is to minimize the speed losses in the Android version due to the new implemented abstract structures. In the current state the program runs mainly on smart phones which have very limited resources. Any negative performance changes on these systems has to be avoided. On the other hand, the targeted new platforms are mainly desktop computers and laptops which have a lot more available resources. So any negative effects caused by altering of the source code have not the same big impact as on a smart phone.

3. **Restructuring the source code**

   Another goal of this project is to clean up the source code to avoid large and unreadable code structures. To enhance the readability it is important

to add comments to all central classes and interfaces. This has no effect on the final product but since this project is not yet completed, it will help future development.
Additionally, the entire project is reorganized according to a suitable architecture pattern to simplify its structure.

3.2 Core functions of jukefox

The logic of the play modes is the main point that makes all the difference between JUKEFOX and other music players. This feature requires a basic music management system with additional information like MUSICEXPLORER[1] coordinates and an infrastructure supporting play statistics.
The current music management contains a framework for a music collection and an import algorithm to update the songs with additional information from the Internet and store them in the database. The interface to the database and other storage mediums is also provided through the core of JUKEFOX.
In summary the core functions are all classes that are essential for the music player including all play modes, the music library management and the database. Not included are all classes which contains elements of the visual output namely all classes from the *View*.

3.3 Extraction of the core functions

The new design should include a better overview and handling of the source code. Additionally, it should satisfy the new opportunities and needs of alternative platforms.
A better overview can be achieved with the above mentioned design patterns, whereby parts of the code are divided into new sections.
Furthermore, it is important to note that this project does not need to be rebuilt from scratch. Instead, the already existing source code is adapted to the new requirements. Therefore, the existing code was analysed.

3.3.1 Before extraction

The analysis showed that in the current state, the project can be roughly divided into two different blocks: The *View*, which includes all visual designs and is responsible for the actual output of the program. This part is optimized only for Android devices and therefore heavily depends on Android specific code. The other block contains most of the logical processes and data structures, namely the *Model* and the *Controllers*. The *View* depends also on the *Model* and *Controller*

[1] http://www.musicexplorer.org

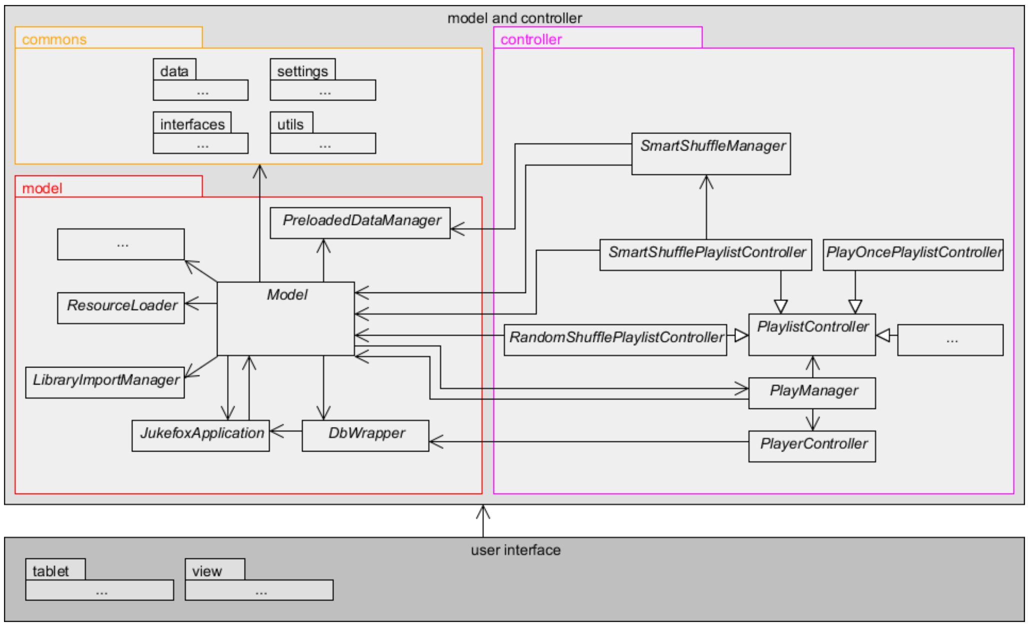


Figure 3.1: Source code structure before extraction

parts but neither the *Model* nor the *Controllers* depends on functions of the *View*. This strict separation allows a more or less operation system independent implementation of all data structures and most of the controllers.
For this project only the *Model* and the *Controllers* are relevant, because to design a new visual concept for the music player on another platform is beyond the scope of this project.

Model class

This central class offers access to all data objects used by the controllers and playmodes, such as songs or playlists. Another task is to control major processes, such as importing new songs or loading resources. The disadvantages resulting from such a class are its size and the illegible structure.
Although the class primarily provides access to Android-independent data, it is heavily dependent on Android's internal functions. This dependence is a result from the manager classes which are attached to it as well as the class `JukefoxApplication` which is the standard start up class for this Android application and includes some important global variables used in other functions.

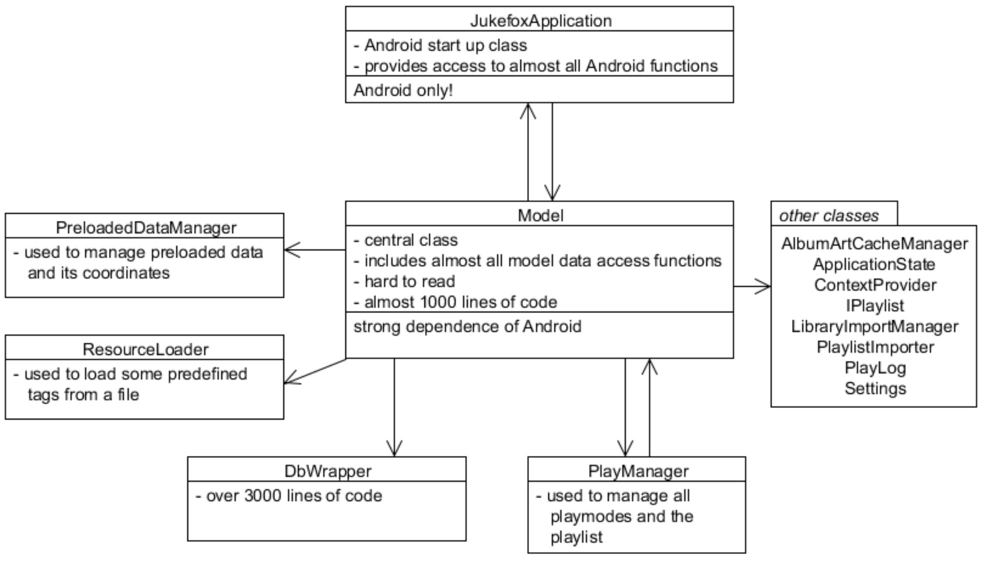

JukefoxApplication
- Android start up class
- provides access to almost all Android functions
Android only!
Model
- central class
- includes almost all model data access functions
- hard to read
- almost 1000 lines of code
strong dependence of Android
PreloadedDataManager
- used to manage preloaded data and its coordinates
ResourceLoader
- used to load some predefined tags from a file
DbWrapper
- over 3000 lines of code
PlayManager
- used to manage all playmodes and the playlist
other classes
AlbumArtCacheManager
ApplicationState
ContextProvider
IPlaylist
LibraryImportManager
PlaylistImporter
PlayLog
Settings

Figure 3.2: Central model class

Music data structure

The data structure of all the music player objects is already well organized as shown in figure 3.3. However in the current state there are many different classes which access and modify the data, mainly the previous mentioned model class. To reduce the resulting disorder it is necessary to define a structured way to access these music data classes.

Database

The access to the SQLite database provided by Android is given by an API class, which specifies a customized function for every kind of query. In addition, there are numerous help classes to manage the database, create and delete tables. The known Java Database Connectivity[2] (JDBC) standards are not considered, probably because the optimization for mobile devices and some needed controlling features are missing.
Besides some auxiliary classes, the database access in JUKEFOX is handled by one very large class called `DbWrapper`. It consists of over three thousand lines of code and is a collection of prepared SQL statements. The class is disorganized and therefore poorly manageable, many functions are similar to others or even duplicated. The Android dependency is only given through the execute functions of the corresponding API class for database access.

[2] http://docs.oracle.com/javase/1.3/docs/guide/jdbc/

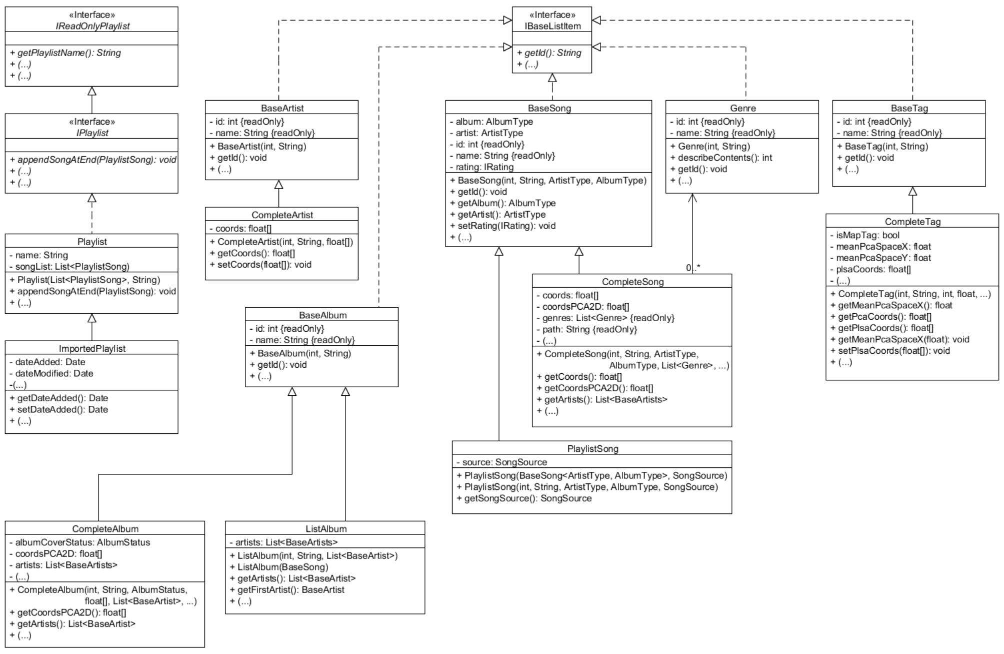

«Interface»
IReadOnlyPlaylist
+ getPlaylistName(): String
+ (...)
+ (...)
«Interface»
IPlaylist
+ appendSongAtEnd(PlaylistSong): void
+ (...)
+ (...)
Playlist
- name: String
- songList: List<PlaylistSong>
+ Playlist(List<PlaylistSong>, String)
+ appendSongAtEnd(PlaylistSong): void
+ (...)
ImportedPlaylist
- dateAdded: Date
- dateModified: Date
-(...)
+ getDateAdded(): Date
+ setDateAdded(): Date
+ (...)
BaseArtist
- id: int {readOnly}
- name: String {readOnly}
+ BaseArtist(int, String)
+ getId(): void
+ (...)
CompleteArtist
- coords: float[]
+ CompleteArtist(int, String, float[])
+ getCoords(): float[]
+ setCoords(float[]): void
BaseAlbum
- id: int {readOnly}
- name: String {readOnly}
+ BaseAlbum(int, String)
+ getId(): void
+ (...)
CompleteAlbum
- albumCoverStatus: AlbumStatus
- coordsPCA2D: float[]
- artists: List<BaseArtists>
- (...)
+ CompleteAlbum(int, String, AlbumStatus, float[], List<BaseArtist>, ...)
+ getCoordsPCA2D(): float[]
+ getArtists(): List<BaseArtist>
+ (...)
ListAlbum
- artists: List<BaseArtists>
+ ListAlbum(int, String, List<BaseArtist>)
+ ListAlbum(BaseSong)
+ getArtists(): List<BaseArtist>
+ getFirstArtist(): BaseArtist
+ (...)
«Interface»
IBaseListItem
+ getId(): String
+ (...)
BaseSong
- album: AlbumType
- artist: ArtistType
- id: int {readOnly}
- name: String {readOnly}
- rating: IRating
+ BaseSong(int, String, ArtistType, AlbumType)
+ getId(): void
+ getAlbum(): AlbumType
+ getArtist(): ArtistType
+ setRating(IRating): void
+ (...)
Genre
- id: int {readOnly}
- name: String {readOnly}
+ Genre(int, String)
+ describeContents(): int
+ getId(): void
+ (...)
BaseTag
- id: int {readOnly}
- name: String {readOnly}
+ BaseTag(int, String)
+ getId(): void
+ (...)
0..*
CompleteSong
- coords: float[]
- coordsPCA2D: float[]
- genres: List<Genre> {readOnly}
- path: String {readOnly}
- (...)
+ CompleteSong(int, String, ArtistType, AlbumType, List<Genre>, ...)
+ getCoords(): float[]
+ getCoordsPCA2D(): float[]
+ getArtists(): List<BaseArtists>
+ (...)
CompleteTag
- isMapTag: bool
- meanPcaSpaceX: float
- meanPcaSpaceY: float
- plsaCoords: float[]
- (...)
+ CompleteTag(int, String, int, float, ...)
+ getMeanPcaSpaceX(): float
+ getPcaCoords(): float[]
+ getPlsaCoords(): float[]
+ getMeanPcaSpaceX(float): void
+ setPlsaCoords(float[]): void
+ (...)
PlaylistSong
- source: SongSource
+ PlaylistSong(BaseSong<ArtistType, AlbumType>, SongSource)
+ PlaylistSong(int, String, ArtistType, AlbumType, SongSource)
+ getSongSource(): SongSource

Figure 3.3: Music data structure

3.3.2 Design and implementation of the new jukefox core

To ensure a clear separation between the different platforms, it was necessary to create three Java projects. On the one hand a main project with the cross-platform core functions of JUKEFOX. On the other hand the two platform specified projects which build on the main project. Functions with platform dependencies are written in an abstract form in the main project and then implemented in the specified project according to its underlying operating system.

Software architecture

When selecting a suitable new software architecture there are different relevant aspects, the most important would be the currently used architecture. Although it is quite possible to change to a completely different design, this would have the distinct disadvantage that the cost would be in a very poor relation to the yield. Another important aspect is to gain an advantage compared to the situation before, for example a better overview or a faster running program.
A modified version of the MVC software architecture pattern was therefore an obvious choice. The development of a new visual design for a different platform does not require any changes in the underlying logic, due to the separation of the *View* from the rest of the code.

The amount of work to change the current source code remains limited because it already has a very similar structure. Additionally, the separation of the data structure and the controllers makes it possible to get a cleaner *Model* structure which is independent from the operative part of the code and also from the used software platform.
Finally, the following three sections according to MVC and based on the 3rd main goal have emerged:

- The *Model*, which holds all data classes, their corresponding access providers and the model manager which own links to all providers. (see figure 3.4)
- The *View* which had to be adjusted only in some details, and still contains all elements used for the visual representation of the application.
- The *Controller*, which contains all the other classes including the data portals in the Java package `data`, the player controllers in the package `controller` and the manager classes in the package `manager`. Additionally there exists a package `commons` which includes all auxiliary functions which can be used by any class.

Naming

To avoid confusion, it is important to define a globally used naming scheme to support the software architecture.

- **Manager** A class which grants both read and write access to a specific resource or topic.
- **Provider** A class which grants access to a specific resource. In contrast to a manager class, this is a read-only function.
- **Data portal** A class used to access and modify external data sources.
- **Prefix:** “`Android`” A prefix for Android specific classes.
- **Prefix:** “`Abstract`” A prefix for classes which contain abstract functions.
- **Prefix:** “`I`” A prefix for interfaces.

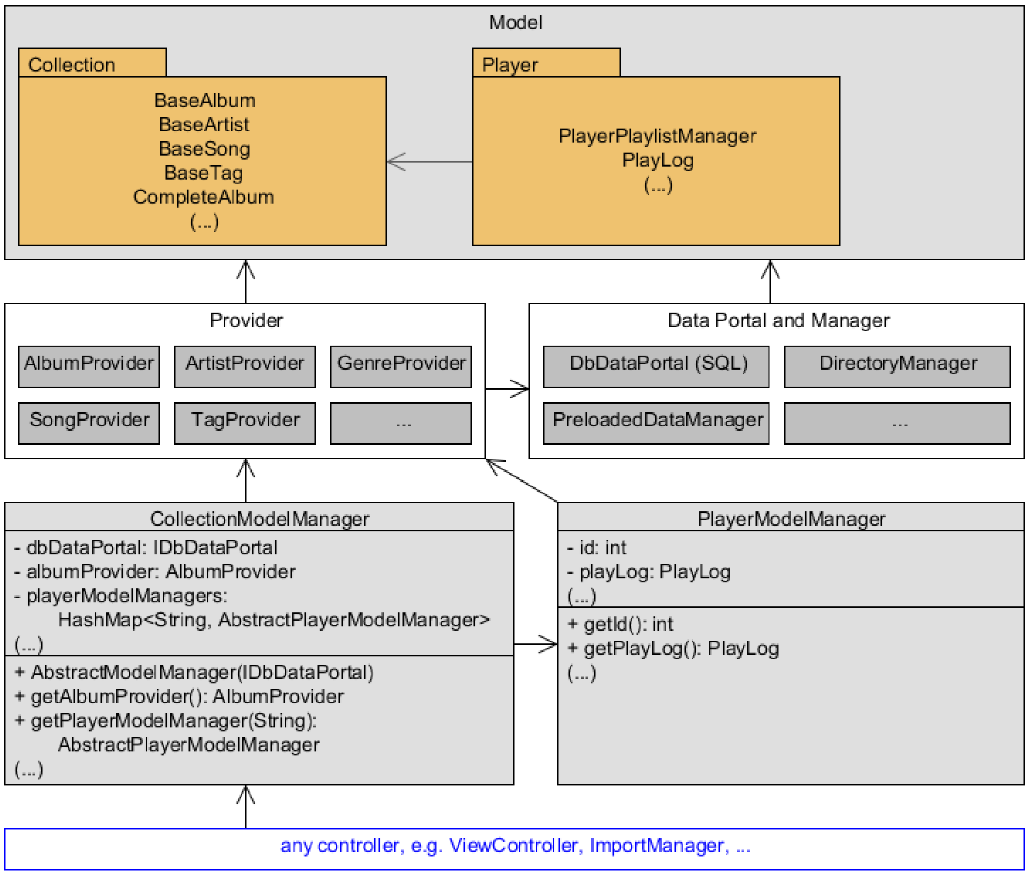


Figure 3.4: The model domain

Model domain

There are two main changes. First the newly introduced providers are used to control the access to all kinds of model objects and serve as an abstracted layer above the data portals. This was previously a task of the rather large model class.
Simultaneously, each provider is used for a particular type of resource, for example the `SongProvider` for all song objects. This improves the overview and enhances the handling of many resources. This improvement is therefore mainly based on the 3rd rule of the main goals.
The second big change is to divide the *Model* into two separate parts:

1. **Collection**

   This includes all static information about model objects, which are similar for all users and can be used regardless of the currently used player. An example would be a song object with all its MP3-tag information and additional parts like file path.

2. **Player**

   In contrast to the collection all objects placed here are in some way dependent of the current player. An example here would be statistics related to a song, for example, how many times it was played or skipped.

Database

The goal of restructuring the `DbWrapper` class was clear, having to write as little code as possible according to the 1st main goal. Nevertheless, it must be possible to freely choose what kind of database technology is used behind the `IDbDataPortal` interface. For the Java port it seems natural to chose the standard JDBC classes because they are supported by many different database manufacturers.
The main class of the SQL data portal is abstract and mainly includes the code of the former `DbWrapper`. All adapted database functions are abstract and similar to the corresponding Android functions. Based on the 2nd rule, the original structure is still unchanged and makes it easy to reintegrate the base class into the original Android project. Functions like `execSelect` or `insertOrThrow` are then implemented in the corresponding subclass of `SqlDbDataPortal` for a translation to the JDBC interface or to be forwarded to the Android API.
The return value of the database functions is another wrapper interface, called `ICursor`, which provides access to the corresponding Cursor in Android or `ResultSet` in JDBC.
As a result, it is now possible to replace the used SQLite database with any other SQL database with a corresponding JDBC driver. The design also allows to exchange the SQL implementation for another database technology.

3.3.3 Reintegration into Android

The reintegration of the modified source code into the Android project is possible without further adaptions. The basic functionality of JUKEFOX is not impaired by the reorganization of the software architecture or the usage of the newly implemented interfaces and wrapper classes. Furthermore, no performance penalty is noticeable on an Android device.

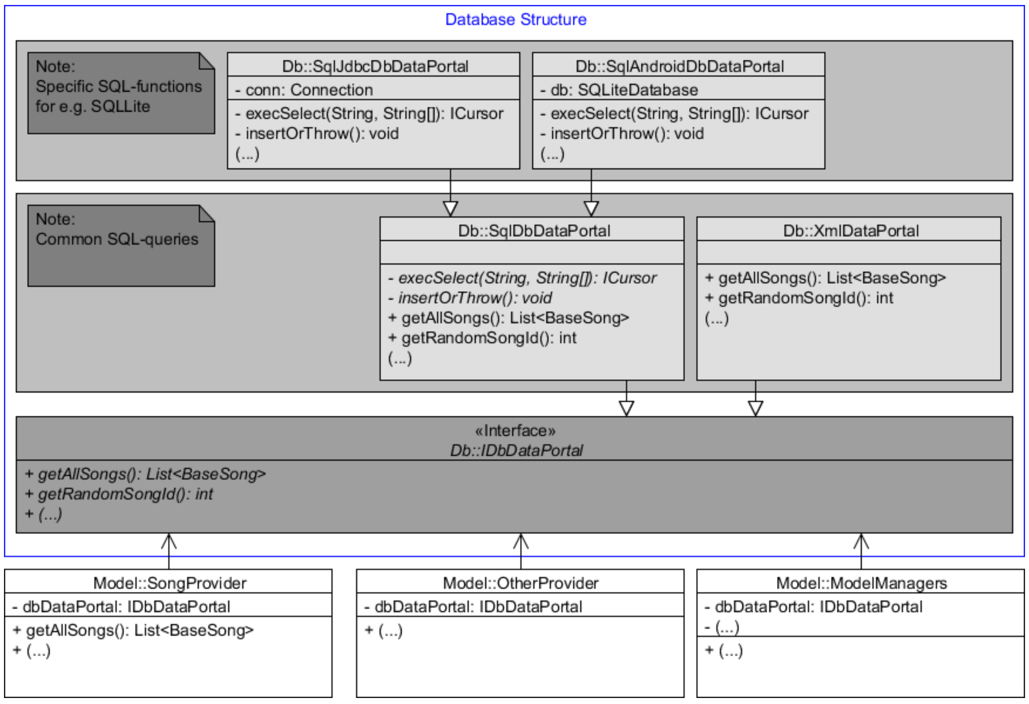


Figure 3.5: The database structure in overview

3.4 Interfaces / API

The playmodes of JUKEFOX are exchangeable classes, which are responsible for the calculation of the next song based on the current state. The old structure, in which the player controller is controlled by the playmodes, is not desired any more. The central control unit is now the `PlayerController`. The `PlaybackController` is now responsible for handling the player and for processing the user input. With the help of the playmodes, the `PlaylistManager` manages the playlist and always keeps the next song prepared. The reason of this reconstruction is the need for a hierarchical structure. The playmodes are now part of the cross-platform core and act as an advisor unit.

3.4.1 Playlist commands

The communication between playmodes and controllers is organized over playlist commands. These are a set of commands, which are performed in a predetermined order on a playlist to get the desired result. An example of a command set would be to attach a new song at the end of the list and play it afterwards. All necessary functions are covered by six commands: `ADD_SONG`, `REMOVE_SONG`, `PLAYER_ACTION`, `SET_POS_IN_LIST`, `SET_POS_IN_SONG` and `SET_PLAY_MODE`. They

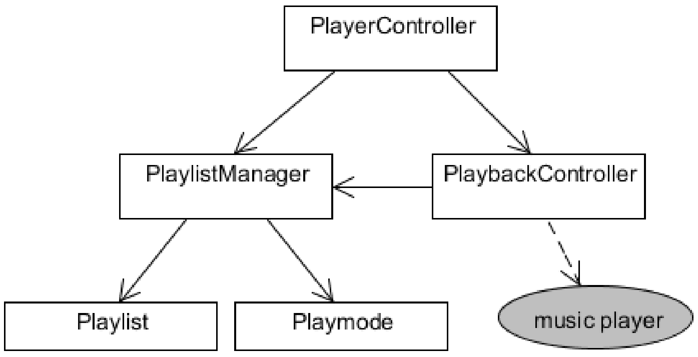


Figure 3.6: The new hierarchical structure of the player and the playmodes

are used to add or delete a song at a certain position on the playlist. Likewise it is possible to set the current position in the list or jump to a exact position within a song. With `PLAYER_ACTION` the command for the music player like `PLAY`, `PAUSE` or `STOP` are given. The complete set of commands is represented by a list and can be processed point by point from the receiver, in this case the `PlaylistController`.

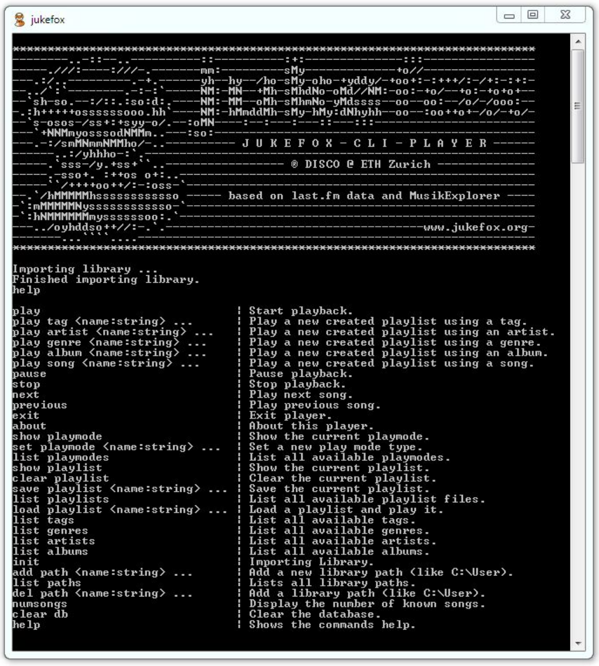
3.4.2 API

Structure

On a personal computer, there are often various media players installed and every user has his favorites. Most of them are free of charge and can be extended with so-called plugins. The functionality of features like smart-shuffle can be realized as a plugin without further ado, assumed that the core capabilities of jukefox are somehow available. Running this core functions as a service in the background and access them via a known API is one possibility. How such a plugin API will look like is still unclear. However, the fact that playmodes no longer controls the player and only suggests the next action, helps to create a useful interface based on the `PlaylistController` and the `PlaybackController`.

CLI player

A command line interface (CLI) is one possible implementation of the API, which is simple but effective. The current CLI for the PC project is not intended to be an API but a simple test user interface. Many of the possibilities you have on the Android user interface are also available on the command-line player. For example choosing the folders of your music library, changing playmode, saving



```
**************************************************************************
--------..-::--..-----------::-----------:+:-----------------:::----------
-----.///:-----://-.--------mm:----------sMy---------------+o//-----------
---.:/..----------.-+.------yh--hy--/ho-sMy-oho-+yddy/-+oo+:-:+++/:-/+:-:+:-
--../`:`---------.-:-:`-----NM:-MN--+Mh-sMhdNo-oMd//NM:-oo:-+o/--+o:-+o+o+--
--`sh-so.--:/::.:so:d:.----NM:-MM--oMh-sMhmNo-yMdssss--oo--oo:--/o/-/ooo:--
-.:h+++++osssssssooo.hh`----NM:-hMmddMh-sMy-hMy:dNhyhh--oo--:oo++o+-/o/-+o/-
--`s-osos-/ss+:+syy-o/.--:oMN----:--:---:---::---:::----------------------
---`+NNMmyosssodNMMm..---:so:---------------------------------------------
---.-:/smMNmmNMMho/-..---------- J U K E F O X - C L I - P L A Y E R ------
------..:/yhhho-:`.-------------------------------------------------------
-----.`sss-/y.+ss+``..----------------- ® DISCO @ ETH Zurich -------------
-----.-sso+. :++os o+:..--------------------------------------------------
-----``/++++oo++/:-:oss-`-------------------------------------------------
--.`/hMMMMMhssssssssssso ----- based on last.fm data and MusikExplorer ----
-`:mMMMMMNysssssssssso-`--------------------------------------------------
-`:hNMMMMMMmysssssssoo:.`-------------------------------------------------
---../oyhddso++//:-.`.-------------------------------------www.jukefox.org-
-------...````  ....------------------------------------------------------
**************************************************************************

Importing library ...
Finished importing library.
help

play                              ¦ Start playback.
play tag <name:string> ...        ¦ Play a new created playlist using a tag.
play artist <name:string> ...     ¦ Play a new created playlist using an artist.
play genre <name:string> ...      ¦ Play a new created playlist using a genre.
play album <name:string> ...      ¦ Play a new created playlist using an album.
play song <name:string> ...       ¦ Play a new created playlist using a song.
pause                             ¦ Pause playback.
stop                              ¦ Stop playback.
next                              ¦ Play next song.
previous                          ¦ Play previous song.
exit                              ¦ Exit player.
about                             ¦ About this player.
show playmode                     ¦ Show the current playmode.
set playmode <name:string> ...    ¦ Set a new play mode type.
list playmodes                    ¦ List all available playmodes.
show playlist                     ¦ Show the current playlist.
clear playlist                    ¦ Clear the current playlist.
save playlist <name:string> ...   ¦ Save the current playlist.
list playlists                    ¦ List all available playlist files.
load playlist <name:string> ...   ¦ Load a playlist and play it.
list tags                         ¦ List all available tags.
list genres                       ¦ List all available genres.
list artists                      ¦ List all available artists.
list albums                       ¦ List all available albums.
init                              ¦ Importing Library.
add path <name:string> ...        ¦ Add a new library path (like C:\User).
list paths                        ¦ Lists all library paths.
del path <name:string> ...        ¦ Add a library path (like C:\User).
numsongs                          ¦ Display the number of known songs.
clear db                          ¦ Clear the database.
help                              ¦ Shows the commands help.
```

Figure 3.7: The new hierarchical structure of the player and the playmodes

and loading of a playlist and all standard music player commands like `PLAY`, `PAUSE` or `NEXT`. The CLI player is the beginning of JUKEFOX on a PC and helps to test the interaction between all important core functions.
The natural CLI library[3] as well as a pure-java audio player library[4] are the basis of JUKEFOX CLI player.

3.4.3 Testing

In a project of this size it is not only useful, but absolutely necessary to have test classes. Using `jUnit` was an obvious choice because it is already perfectly integrated in the used development environment. Different database functions and the newly implemented settings management are the main part of the testing. The CLI player turned out to be quite helpful to test related core functions such as for example the import process of new songs. The interplay between all the corresponding functions used in this process can not be covered efficiently with unit tests. The CLI player can also support the testing of new features in future development.

[3]http://naturalcli.sourceforge.net/
[4]http://code.google.com/p/java-audio-player/

Chapter 4

Conclusion

All three main goals were achieved as planned. Firstly, the functional CLI player on a home computer shows that all core functions run independently from the operating system. Secondly, the successful reintegration into Android without major impacts on the performance proves that the applied changes do not effect the functionality in a negative way.
Many of the newly implemented and most of the important original classes are now provided with comments and ordered in a newly structured way. This improves the overview as stated in the 3rd main goal.
However, many adaptions are still necessary to be able to comfortably use JUKEFOX on a desktop computer. The complex and dynamic data management and the multi-user capability on a computer requires some additional changes in the JUKEFOX core. The current database structure is designed for a single user which is the default for Android devices. Furthermore, the problem of temporarily unavailable data on external storage or even network storage is not yet resolved. To attract people to use JUKEFOX on a PC it is necessary to create a convenient user interface, for example a plugin for a popular music player.

Appendix A

Appendices

A.1 Used programs

- **eclipse** (Indigo Service Release 1) `http://www.eclipse.org`
  - **Android SDK** (16.0.0) `http://developer.android.com/sdk/index.html`
  - **subclipse** (1.6.18) `http://subclipse.tigris.org`
  - **UMLet** (11.3) `http://www.umlet.com`
  - **Fat Jar** (0.0.31) `http://fjep.sourceforge.net`
- **Enterprise Architect** (9.2) `http://www.sparxsystems.com`