Qdio - Final Report

Hugo Cliffordson[1], Alrik Kjellberg[1], Melker Veltman[1], & Oskar Wallgren[1]

Group 25

[1] *TDA 367*

Chalmers University of Technology

2018

Contents

1 Introduction

Ever since the rise of on-demand music streaming services, the world has gained the largest increase in music consumption and discovery in the history of music[1]. To fully understand the effect music has on peoples lives note that, as of 2017, the average American is spending no less than 4.5 hours a day listening to music [2]. With *Spotify* being the largest actor in the world for paying users [3] we saw an opportunity to further users flexibility with *Spotify* even more.

The functionality that *Spotify* is not providing is the availability to let users communicate with the device playing the music without having to manipulate this device physically. Our application *Qdio*, is the solution to this problem. The app enables a person playing music from *Spotify* to open a room that other users can connect to. Users connected to the room will then be able to search for music and add it to the music queue on the device playing the music. The use of this application will benefit everyone listening to music using *Spotify* as their on-demand music streaming service. The prosperity shines in the context of situations. Imagine offices, study groups, families or parties where there is a person playing music through their device or to an external speaker system. Users in these situations will then be able to contribute to the music playing using the application *Qdio*.

In short *Qdio* is an extension to the *Spotify* app with the purpose of enabling users to easily share their music to the queue of the person playing music through their own devices.

1.1 Definitions, acronyms, and abbreviations

Activity Class representing a Scene or screen in Android

AsyncTask Android-based helper class to easily manage threads

Fragment Class representing a part of an activity in Android

RESTful API Programming interface supporting a Representational State Transfer

2 User Stories

2.1 Epic 1

As a host, I want to create a room that shares a music queue so that others can add songs to my queue from other devices

1. Open a Room
   Story ID: 1
   As a host, I want to be able to open a room so that my friends and I can

have a common queue list for music.
Acceptance criteria:

(a) A button in welcome screen with text "create a room" in the center of the screen

(b) A loading animation that shows until room is successfully created.

(c) Pressing it takes user to main interface with search and song view.

2. Pause the Music
Story ID: 2
As a host, I want to be able pause the music so that my device gets quiet
Acceptance criteria:

(a) Below the song progress bar there is a two striped pause button that stops the music

(b) When pressing pause the button changes appearance to a play-icon

3. Fast forward Song
Story ID: 3
As a host, I want to be able to seek in a song so I skip or return to a part of the song
Acceptance criteria:

(a) Below the song information there is a seek bar with a dot representing the duration and position of the song

(b) Pressing this dot and dragging it somewhere else on the bar changes where in song it's playing

4. Skip Song
Story ID: 4
As a host, I want to be able to skip a song when I don't like the current playing.

(a) Show a button next to the play icon that has a skip icon

(b) Pressing this puts current song to history and puts first song in queue to current

2.2 Epic 2

As a guest, I want to be able to connect to a room so that I can add music to a shared music queue

1. Find Available Rooms
Story ID: 5
As a guest, I want to find a list of all available rooms so I can choose which to connect to
Acceptance criteria:

(a) A welcome screen shows a button "connect to a room"

(b) Pressing this button open a view with a list of all available rooms

2. Connect to a Room
   Story ID 6:
   As a guest, I want to be able to connect to a room so that I can contribute to a shared music queue.
   Acceptance criteria:

   (a) A list that shows all available rooms with their names

   (b) Pressing an item in the list connects the user to the room and takes the user to the main interface

2.3 Shared epic

As a user, I want to be able to search for songs and see playback status of a room

1. Search for a Song
   Story ID 7
   As a user, I want to be able to search for a song in the Spotify library so that I can find the song I want
   Acceptance criteria:

   (a) A search button in the bottom right corner opens up a keyboard and shows a search field in the top

   (b) As a user is searching a list with songs appear and reloads when adding/removing letters

   (c) Pressing search icon on keyboard hides the keyboard and shows list of search result

2. Add Song to Queue
   Story ID: 8
   As a user, I want to be able to add a song to my shared music queue so that I share my music with others
   Acceptance criteria:

   (a) From a list of songs one should be able to press an item and add it to the queue

   (b) The added song appears in the queue list

3. Search for an Artist
   Story ID: 9
   As a user, I want to be able to search for an artist so I can find a song made by an artist
   Acceptance criteria:

(a) Typing a name of an artist in search field should show songs from that artist ordered by relevance

4. See the Queue
   Story ID: 10
   As a user, I want to be able to see the current queue list so I can keep track of upcoming songs
   Acceptance criteria:

   (a) The bottom 2/3 of the screen shows a list of all songs in the queue

   (b) A list item should contain song name and artist

   (c) If there are many songs in the queue the user should be able to scroll down the list to see hidden songs in the queue

5. Playback Status
   Story ID: 11
   As a user, I want to see the playback status of a song so that I can get information about the song and its progress
   Acceptance criteria:

   (a) The top 1/3 if the interface shows an album cover picture to the left, next to name of song, artist, and album

   (b) Below the album cover picture and song information, there is a progress bar indicating song duration stretching from left to right

3 User Interface

Since Qdio is all about being easy to use, the user interface is simple and minimalist. The application contains five different scenes in total. Three out of these five will be used if the user wants to open a room in which people can connect to. Four out of these five will be used if the user wants to connect to an existing room.

3.1 Welcome Interface

Upon opening the application, the user will interact with the welcome interface. See figure 1. This scene is built up by two large and distinct buttons. The first of these reads 'Create Room' and reacts the way the user thinks. It will create a room and navigate the user to the host interface. The second button says 'Connect To Room'. Upon clicking, it will navigate the user to the browse interface.

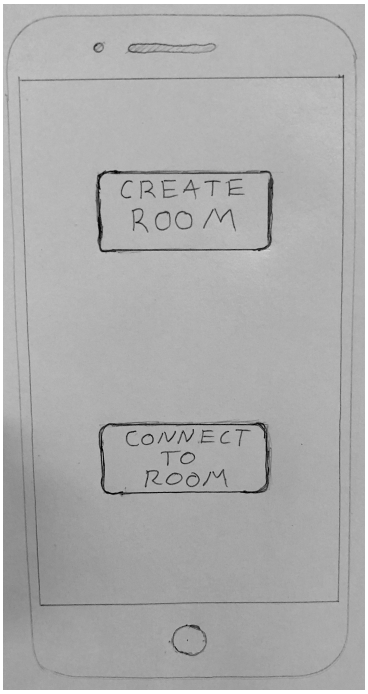

Figure 1: Welcome Screen

3.2 Room Browse Interface

After clicking 'Connect To Room' in the welcome interface the user will be lead this scene. This part of the application only shows a list of names of the available rooms that the user is able to connect to. See figure 2. Upon selecting a room, the guest main interface is shown.

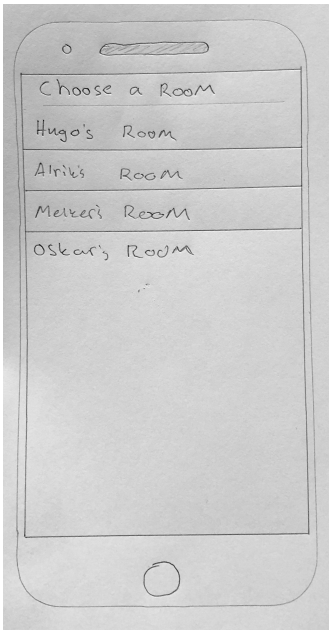

Figure 2: Room Browse Interface

3.3 Guest Main Interface

The guest main interface, see figure 3, is the main scene for users connected to a room. This interface shows the playing status and the active queue of music. In the top of this scene, the user can see the song name, artist, album and an album cover of the song that is currently playing. Beneath the song information, the user will be able to see all the songs in the queue and in which order they will play. The button with the magnifying glass icon will navigate the user to the music search interface.

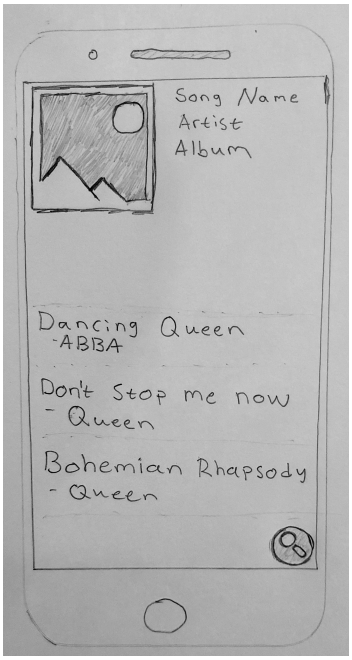

Figure 3: Guest Main Interface

3.4 Host Main Interface

The host main interface, see figure 4, is similar to the guest main interface. The difference between this scene and the guest main interface is that the host interface has controls for playing, pausing, seeking and changing song.

Figure 4: Host Main Interface

3.5 Music search Interface

After clicking the search button available in both host and guest main interface, the user will arrive at this scene. See figure 5. The interface contains a search bar and a keyboard in which users can search for songs, artists and albums. It also contains a list of the user's input results. When a song is pressed, it will be added to the queue and the user is brought back to the main interface.

Figure 5: Search Screen

4 Domain Model

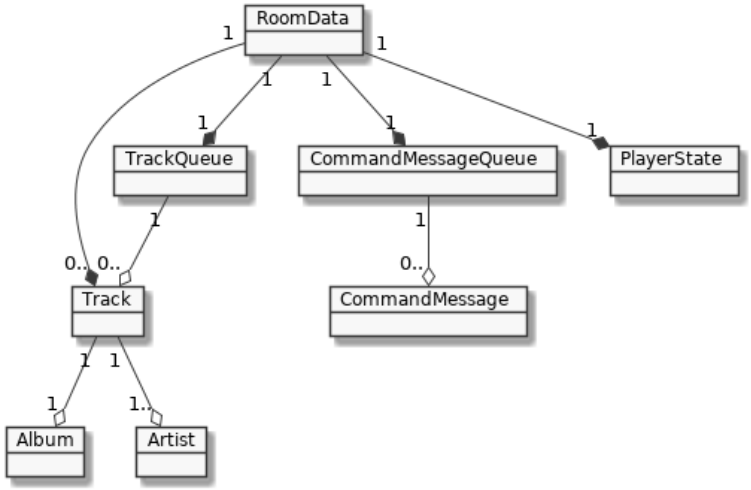

Figure 6: Domain model of Qdio

4.1 Classes

The entry point of the application is the class *RoomData* with the responsibility to delegate functionality to other classes.

The class *PlayerState* reveals what state the player is in, if it is playing, paused or stopped.

CommandMessageQueue works as a queue where all incoming messages from other devices are added and processed from. By keeping this logic separated from the communication logic, it can be separated from the main domain logic.

CommandMessage is the object that contains a message and the action to be executed. Depending on which action is set, the message is interpreted differently.

TrackQueue contains the logic for the current song queue list, which is implemented as a *First-in-First-out* queue. With the logic being separated from an ordinary Java *Queue* it would be an easy implementation to add vote functionality and a priority on songs of the same genre as the rest of the queue.

Track is the model for a song. This contains one or more *Artists* and one *Album*.

5 System architecture

The *Qdio* application features two separate use cases in the form of a room host and a room guest. However the logic is mostly kept the same for both the host and the guest.
The application involves at least one Android device, but there is no upper limit for the amount of devices connected to one room. The communication between devices is managed by *Google Nearby Connections*. See Dependencies.

Qdio consists of several separated modules divided by business logic and responsibility in line with the single responsibility principle. An example of this is the *information* module, which is the only place the system communicates with the *Spotify* information API. All other modules relying on information from *Spotify* gets it through this module.
Other modules including communication, playback, model, room and view, can be read about under corresponding subsections.
All the following diagrams have been generated using *PlantUML*[4] source code and renderer. All the source code for the diagrams can be found in the git repository[5].

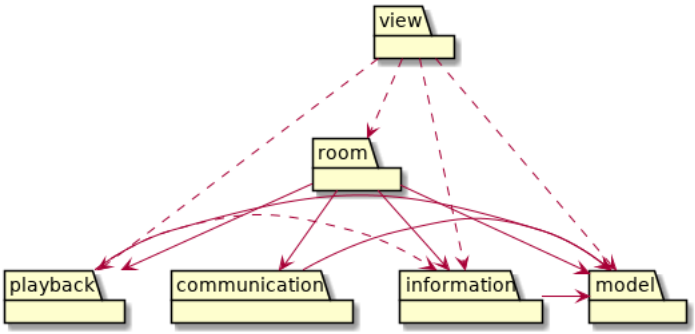

Figure 7: General package dependencies

Figure 7 describes the general dependencies between the modules of the system. Notice that there are relatively few dependencies between the strictly separated modules such as playback, communication and information. There is also nothing that depends on the view and the model is completely free from dependencies.

The view logic of the application is written using the *Model-View-ViewModel* structure, featuring extensive usage of observables in the form of *LiveData*, see View model & LiveData.

The technical flow of creating a room starts with the owner advertising the device using *Google Nearby Connections*. The application instantiates a new

headless Android fragment which sets up the connection to the *Spotify playback API*. After that is done the Room class is instantiated and takes control over the data flow in the application. The room owner is now taken to the main screen and guests are now able to connect to the room.

The technical flow for a user connecting to a room begins with the application starting the discovery process. The found rooms gets put in a list which is observed from the view. When one of the found rooms is selected, the connection process starts and when it is done the user is taken to the main screen. The user is now able to see the queue of the room, search for a song and add a song to the queue.

5.1 External dependencies & Libraries

Several dependencies have been used throughout the project since the implementation of these parts would have been outside of the application scope.

5.1.1 ViewModel & LiveData

The *Android Architecture Components Library*[6] is used in the application to achieve easier usage of observables in the view logic.

5.1.2 Google Nearby Connections

Google nearby connections[7] is used for the communication between devices. The library uses a peer-to-peer connection method abstracted away from the technologies that are used. By using this library the application does not have to use a server component to achieve its purpose.

5.1.3 Spotify App Remote

To be able to play music on the Android device through the *Spotify* application, the *Spotify app remote SDK* [8] library is used. The library creates an interface with the local *Spotify* application on the device and abstracts several parts of the logic required otherwise.

5.1.4 Spotify web API

Spotify also exposes information about their tracks, albums and artists through a RESTful API. Qdio uses a wrapper library around the API called Spotify Web API Android[9], since writing a wrapper specific for the application was out of the project scope.

5.1.5 Testing Libraries

To be able to unit test the application, *JUnit4*[10] has been used in conjunction with *Mockito*[11] and *PowerMock*[12]. The purpose of *Mockito* in a unit testing

environment is to be able to mock methods of classes outside the scope of the unit being tested. *PowerMock* has been used to be able to mock static methods. To test methods with sample data, the library *Faker*[13] was used to create a test data utility.

5.2 Android application

To be able to verify the the quality of the application, *STAN*[14], *PMD*[15] and *FindBugs*[16] has been used. *STAN* analyzed the source code of the application and reported no circular dependencies and verified our general package dependency diagram. *PMD* reported completely without errors using an already defined ruleset[17]. *FindBugs* only reported errors about the text encoding on strings not being specified, this was interpreted more as a warning than an error.

5.2.1 Information

The information module is responsible for the fetching of information such as track, artist and playback data from *Spotify*.

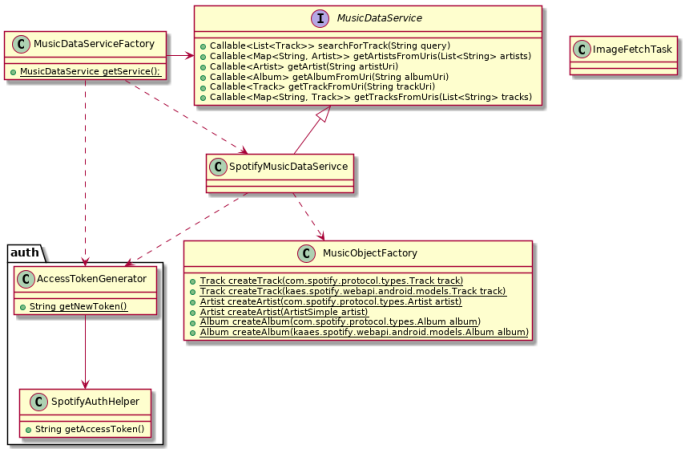

Figure 8: Information package dependencies

When music data is fetched, it is done through the Spotify web API. The API has been wrapped in the *SpotifyMusicDataService* class to suit the application's needs since the use of *AsyncTasks* is required. This is due to the calling of web requests not being permitted on the main thread. The module also contains an image fetch task which is used to acquire the album covers as bitmap. Similar to the music data service, the image fetch task also uses *AsyncTask* due to the use of webrequests.

The implementation of the *MusicDataService* interface is package-private and

only reached through the Factory, in line with the dependency inversion principle and open-closed principle.

When the application uses the *MusicDataService*, the factory ensures that the *SpotifyMusicDataService* is instantiated and has a valid access token. If it does not have a valid token, it is generated from the *AccessTokenGenerator* class. The *AccessTokenGenerator* delegates the functionality of sending the web requests to get the tokens to the *SpotifyAuthHelper* class.

5.2.2 Communication

The communication module handles all communication with external devices when a device has either connected to a room or opened a new room.

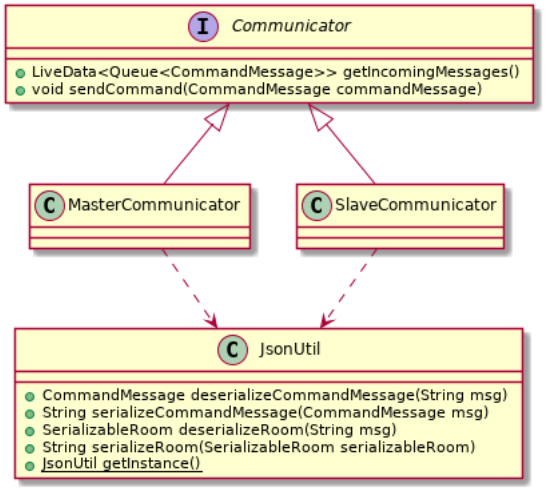

Figure 9: Communication package dependencies

It is through the *Communicator* interface the separate *Room* implementations sends and receives their commands. The *Room* implementations are completely independent from the *Communicator* implementation being used, however the *MasterCommunicator* is used for the *MasterRoom* and the *SlaveCommunicator* is used for the *SlaveRoom*.
The communicator package does not know what it is sending or receiving. This is in line with the separation of concerns principle and induces low coupling.
The *JsonUtil* class is used to serialize or deserialize data in JSON format.

5.2.3 Playback

The playback module takes care of the music playback and the *Spotify* app remote authentication.

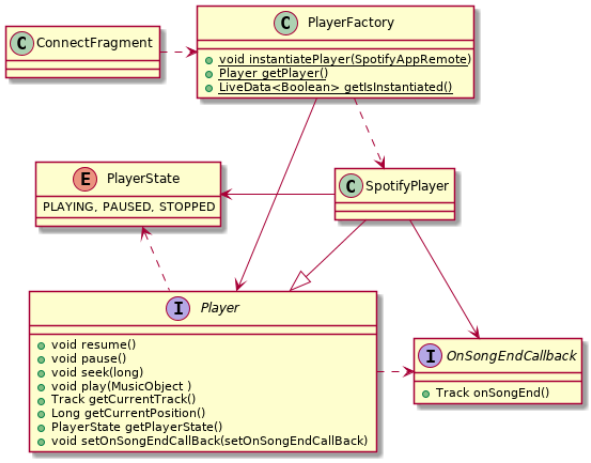

Figure 10: Playback package dependencies

When the application is told to open a new room, the *ConnectFragment* class is instantiated and begins the authorization with the *Spotify App Remote SDK*. The ConnectFragment is a *Headless Fragment* meaning that it gets attached to an Android activity but has no view at all.
The module features a Player interface which is implemented in *SpotifyPlayer* class. The implementation of the interface uses the same principles as the implementation of *MusicDataService* resulting in information and implementation hiding while maintaining open-closed principle.

OnSongEndCallback is used to attach a callback to a player. The callback is executed when a song has ended.

The *PlayerFactory* features static methods to instantiate, get and check if the Player has been instantiated. *PlayerFactory* is also the entry point of this module.

5.2.4 Model

The model package is where the applications domain parts are located. It consists of critical objects that define the state of the application. This package has no dependencies as described in the System Architecture section and shown in the general UML package diagram.

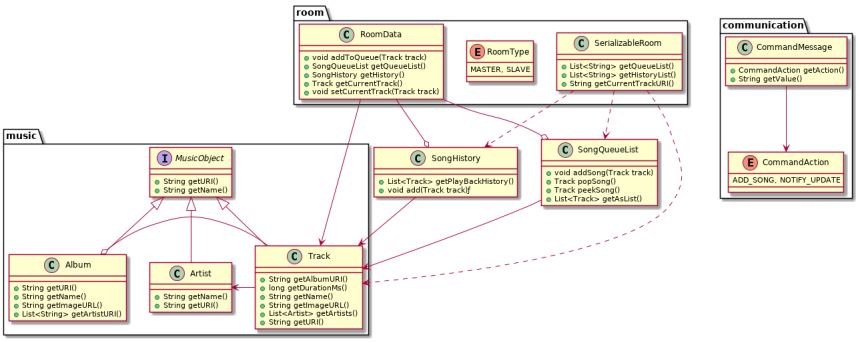

Figure 11: Model package dependencies

The application features its own implementation of a Track, Album and Artist object that differ slightly from the implementations found in both the web API and the app remote API. All these implement the *MusicObject* interface to simplify searching and queuing due to them all having a similar URI. There are also objects for the song queue and the history of recently played songs. Even though these classes currently lack large amounts of logic, they exist so the possibility of extension of functionality is more feasible. Furthermore featured in the model package is the *CommandMessage* object which is used in every message sent between devices. Whenever a message is sent, a *CommandMessage* is created with a *CommandAction*. This message contains either *ADD_SONG* or *NOTIFY_UPDATE* and its value is passed to the instantiated communicator. Some of the Room related logic is placed in the model package as well. The data used in the implementation of Room is stored by using a *RoomData* instance. The data may be simplified using the *SerializableRoom* class. The sole purpose of this class is to be converted to JSON and sent using a *CommandMessage*.

5.2.5 Room

The room module contains the implementation for the different types of rooms used depending on the type of user. The type of room defines restrictions and the manner in which the application communicates.

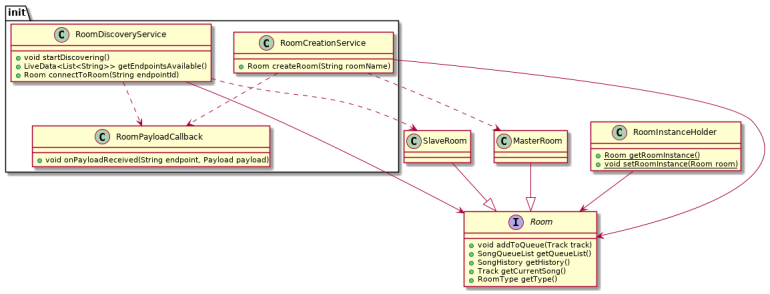

Figure 12: Room package dependencies

There are two different types of rooms where the used type depends on if the user is a host or a guest. When the application is entered as a host, the *MasterRoom* is instantiated and used throughout the application lifecycle. The *MasterRoom* fetches an instance of the Player and is therefore able to manipulate its current state by calling play when a song is added to an empty queue and setting the *onSongEndCallback*. The communicator in the *MasterRoom* is set to send all connected devices a *CommandMessage* containing the *NOTIFY_UPDATE CommandAction* and a *SerializedRoom* object with the current song, song queue, and song history in a JSON format.

If the application however is entered as a guest user, the *SlaveRoom* is instantiated and used. There is no dependency towards the Player and it is therefore not possible to manipulate. The *SlaveRoom* has its own representation of the current song, song queue and song history that updates when a *NOTIFY_UPDATE CommandMessage* is recieved. When calling the *addToQueue* method, a *CommandMessage* is sent using the communicator with the *ADD_SONG CommandAction* and the track URI to the host *MasterRoom*. Both types of room implements the *Room* interface which is used as a middle hand between the rooms and their communicators.
Since every instance of the application only contains one single room, there is a *RoomInstanceHolder* that make the current room easy to access. It contains a method that returns the current room instance or throws an exception if no room is yet instantiated.
The *RoomDiscoveryService* is the service that handles both discovering of open rooms for guests and connection to an open room if the guest decides to connect to it. To be able to observe on the open rooms from a *ViewModel*, they are exposed through a *LiveData* object.
The *RoomCreationService* handles the creation of a new room, and all the calls to the *Google Nearby Connections API* that is needed.

5.2.6 View

The view logic in the application is featured in the view module. Here resides all activities and fragments used throughout the application.

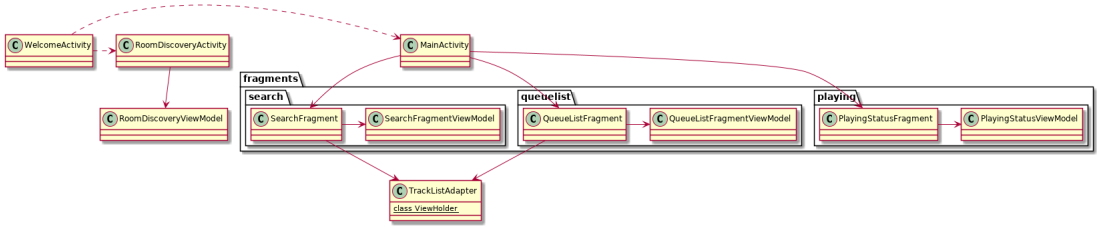

Figure 13: View package dependencies

Qdio features extensive usage of the *Model-View-ViewModel* pattern, resulting in a separation of concern in the view logic that otherwise may become highly coupled to the business logic behind it. Therefore every class except the ones that barely contain any view logic also features a viewmodel that exposes observables in the form of *LiveData*.

To separate the logic even further, the application uses Android fragments that are self contained view objects that may feature their own viewmodel. *Search*, *QueueList* and *PlayingStatus* are the fragments used in the application.

6 Access control

The application *Qdio* has no permissions that are managed on account level. In other words there is no functionality that specifies, for instance, an admin user vs a regular user. Instead of using different roles, the application is built up by the implementation of the interface *Room* by using the method *getType*. This method returns either master or slave.

The two different roles are visible to the end user only in the context of a slave being connected to a room or a master having opened a room. For the master the play/pause button and the seek bar will be visible and available for use while they are hidden for the slave. The reason for this is so that only the user playing the music has the possibility to manipulate it.

7 Peer Review

The following is a review with improvement suggestions of the *Gymcompanion* application created by Alexander Bergsten, Marcus Svensson, Erik Bock, Augustas Eidikis and Daniel Olsson. The review is divided in categories that are important for a good structured software project. See subsections below.

7.1 Design principles

The coding style in the project is somewhat inconsistent, with some parts featuring the usage of Lombok[18] and some parts that do not. An example of this is the *ModelUser* class, which both uses Lombok annotations and regular getter methods. However a lot of parts of the code is consistent, an example of this is all control flow structures which are consistently expanded.
Inheritance and composition is the largest contributors to code reusability in the project. The most prevalent usage of inheritance in the project is featured in the *ViewModel* package, where every concrete class inherits the abstract *ObservableViewModel* which inherits the *BaseViewModel* that contains the model as a final static member. Composition is mostly used in the Model package, where the *GymCompanion* delegates a lot of its functionality to other classes. The usage of Composite classes also contributes to the maintainability of the project. The maintainability could however be improved by depending on interfaces and using factories for the instantiation of objects. The usage of factories would also greatly improve the ability to add and remove functionality.

The model features usage of the strategy and template method design patterns. The strategy pattern is used for both filtering and sorting lists containing class instances implementing their interface *ISortable*. The template method pattern is used in the *ModelWorkoutExercises* package through the abstract class *Exercise*.
The project also makes use of the Observable design pattern in conjunction with the Model-View-ViewModel pattern. All of the ViewModel classes implements the *Observable* interface with the view classes observing these.

7.2 Code documentation

The code of the application, apart from the model package, is poorly documented. All classes in the model package features an overview of title, author, created date, purpose and location of use as comments in the Javadoc format. Many of the public methods in the model are also commented using Javadoc. However there is a lot of complexity in the model that is not explained by comments. For the rest of the application the code documentation is less consistent. Comments occur in very few places and there are classes missing java comments and Javadoc comments all together.

7.3 Class & Method Naming

The naming style in the project is for the most part consistent, using upper case letters for first letter in classes and all lowercase letters for methods. This is in line with the Google Java Style Guide for naming referring to the *CamelCase* style [19]. However, enumerations in the project uses all uppercase letters which contradicts the guide since they are considered to be classes. Packages should follow *lowerCamelCase* but are uppercase in the project.

Over all the project follows naming style conventions well. Method names are well associated with their function, classes well describe their responsibilities and variables are used as they are named.

7.4 Testing

The testing of the application is relatively thorough. When running all the *JUnit* tests with code coverage, the result shows that 88% of the classes in the model package are tested but merely 61% of the methods in said classes. Even thought this is not bad at all, classes like *GymCompantion* could use more tests as well as the whole *DataStorage* package which currently has zero tests. Also the *GymCompanionSearchTest* is unfinished and the only assertion is automatically true.

7.5 Project Structure & Understandability

With the lack of comments in consideration the code is still fairly easy to understand. This is because the application uses acceptable names for methods and objects. The project uses a model-view-viewmodel structure which is favorable for this type of android application. The model is strictly separated from the rest of the application. However the *Observable* and *Observer* interfaces could be separated from the model package since it is not a part of the domain model.

7.6 Modular Design & Unnecessary Dependencies

There are many modules in the application and for the most part they contain the correct classes that belong together. The *ViewModel* and *View* packages are very well defined and include all the classes that are expected. The model package however, contains a more diverse selection of classes that do not necessarily belong together in the same manner as the other main packages. There could for instance be a service layer that is located outside the model package. There are also some dependencies from the view package to the model package which contradicts the Model-View-Viewmodel pattern.

7.7 Security & performance issues

The application was installed on a android device to get higher performance compared to the Android Studio emulator. With *Gymcompanion* on the device, we found no performance issues. The application loading time was minimal and it worked smoothly. The calendar, workout statistics and all the filters worked fine and as expected. There were no security problems identified. The application is secure for its purpose.

References

[1] L. B̃. Martens. (Mar. 2016). The increase of music consumption and discovery, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167624516000068.

[2] Nielsen. (Nov. 13, 2017). We listen to music more than 4 1/2 hours a day, [Online]. Available: https://www.marketingcharts.com/industries/media-and-entertainment-81082.

[3] wikipedia. (Oct. 13, 2018). Comparison of on-demand music streaming services, [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_on-demand_music_streaming_services.

[4] (2018). Plantuml, [Online]. Available: http://plantuml.com/ (visited on 10/21/2018).

[5] (2018). Git repo, [Online]. Available: https://github.com/alrikkjellberg/IceTrailerGang (visited on 10/21/2018).

[6] (2018). Android architecture components, [Online]. Available: https://developer.android.com/topic/libraries/architecture/ (visited on 10/17/2018).

[7] (2018). Google nearby connections api, [Online]. Available: https://developers.google.com/nearby/connections/overview (visited on 10/17/2018).

[8] (2018). Spotify app remote sdk, [Online]. Available: https://developer.spotify.com/documentation/android/ (visited on 10/17/2018).

[9] (2016). Spotify web api for android, [Online]. Available: https://github.com/kaaes/spotify-web-api-android (visited on 10/17/2018).

[10] (2018). Junit - about, [Online]. Available: https://junit.org/junit4/ (visited on 10/17/2018).

[11] (2018). Tasty mocking framework unit tast, [Online]. Available: https://site.mockito.org/ (visited on 10/17/2018).

[12] (2018). Powermock, [Online]. Available: https://github.com/powermock/powermock (visited on 10/17/2018).

[13] (2018). Java faker, [Online]. Available: https://github.com/DiUS/java-faker (visited on 10/17/2018).

[14] (2018). Stan - structure analysis for java, [Online]. Available: http://stan4j.com/ (visited on 10/21/2018).

[15] (2018). Pmd an extensible cross-language static code analyzer, [Online]. Available: https://pmd.github.io/ (visited on 10/21/2018).

[16] (2018). Findbugs, find bugs in java programs, [Online]. Available: http://findbugs.sourceforge.net/ (visited on 10/21/2018).

[17] (2018). Pmd ruleset for android application, [Online]. Available: https://github.com/ribot/android-boilerplate/blob/master/config/quality/pmd/pmd-ruleset.xml (visited on 10/21/2018).

[18] T. P. L. Authors. (Oct. 24, 2018), [Online]. Available: https://projectlombok.org/.

[19] Google. (). Google java style guide, [Online]. Available: https://google.github.io/styleguide/javaguide.html#s5-naming (visited on 10/24/2018).