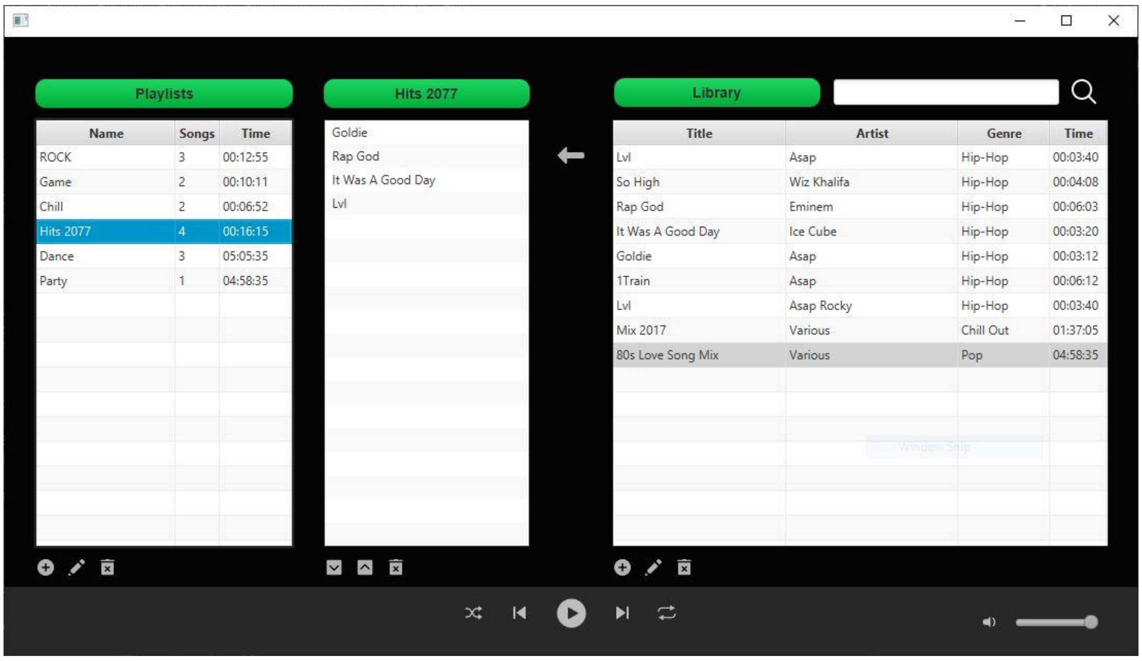
MyTunes
Compulsory Assignment #4

CSe2019B MyTunes Group C



Name	Songs	Time
ROCK	3	00:12:55
Game	2	00:10:11
Chill	2	00:06:52
Hits 2077	4	00:16:15
Dance	3	05:05:35
Party	1	04:58:35

Goldie
Rap God
It Was A Good Day
Lvl

Title	Artist	Genre	Time
Lvl	Asap	Hip-Hop	00:03:40
So High	Wiz Khalifa	Hip-Hop	00:04:08
Rap God	Eminem	Hip-Hop	00:06:03
It Was A Good Day	Ice Cube	Hip-Hop	00:03:20
Goldie	Asap	Hip-Hop	00:03:12
1Train	Asap	Hip-Hop	00:06:12
Lvl	Asap Rocky	Hip-Hop	00:03:40
Mix 2017	Various	Chill Out	01:37:05
80s Love Song Mix	Various	Pop	04:58:35

Handed-in by Team C

1. **Abdiqafar Mohamud Abas Ahmed**
2. **Radoslav Backovsky**
3. **Anne Luong**
4. **Michael Haaning Pedersen**

Date: December 18, 2019

Contents

State of delivery

myTunes is a traditional music player application, where a user can manage songs and playlists. The interface has a song table on the right with all songs (library), playlist table on the left and a list of songs on the playlist in the middle. The state of delivery is as follows:

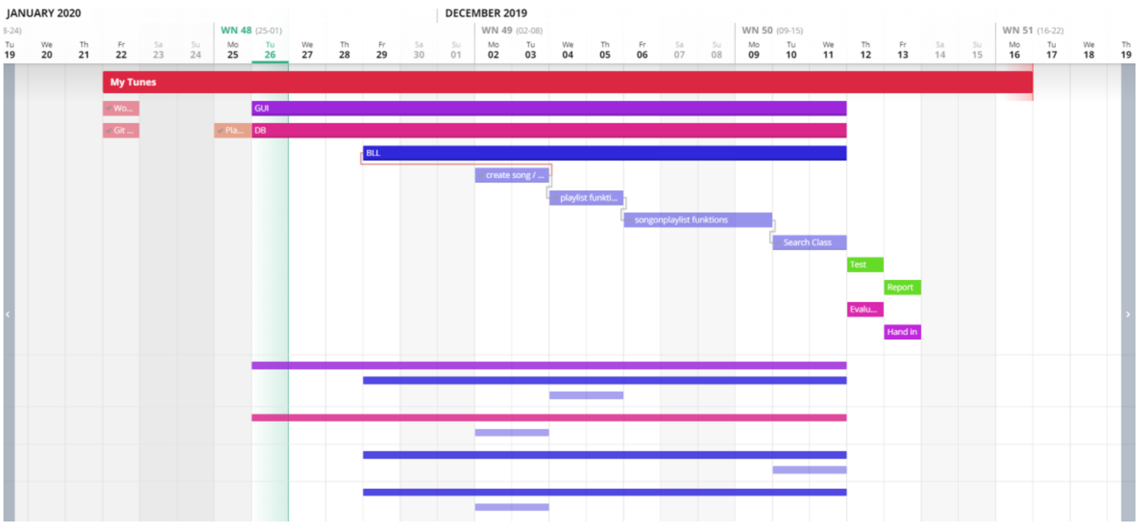
Functional requirement met?		Yes/No
The application must be a desktop application with a graphical user interface.		Yes
Song table	All songs are shown in the song table.	Yes
	The user can sort the table using the built-in sorting functionality of the table columns.	Yes
	The user can create a new song. A dialog is shown when creating.	Yes
	The user can edit a selected song. A dialog is shown when editing. *In the edit dialog, the current information is shown.*	Yes
	The user can create and delete genres. Clicking the (+) button, shows an originally hidden text field to enter a genre name. The user can save the genre by clicking the save button beside the text field. Both the text field and save button disappear after clicking the save button. The user can delete a genre by selecting the genre in the ChoiceBox and clicking the (-) button. The changes are reflected immediately in the ChoiceBox.	Yes
	The user can delete a selected song. A dialog is shown to confirm the deletion. *The deleted song is removed from all playlists.*	Yes
	In the Delete dialog, there is an option to delete the song file itself.	No
Filter	The user can enter a filter query in the search box above the song table.	Yes
	When the filter button is clicked, only songs with the title or artist containing the query are shown. The filter button changes to a clear button, when the filter is active. Clicking the clear button will show all songs and the button is changed to the filter button.	No
	When the user enters a filter query in the search box, only songs containing the query in the title or artist are shown. The filter is active, when a filter query is entered. Filter button is not needed.	Yes
Playlist	A playlist is identified by name and contains a sequence of songs in a specific order.	
	The user can create a new playlist. A dialog is shown when creating.	Yes
	The user can edit a selected playlist. A dialog is shown when editing. *In the edit dialog, the current information is shown.*	Yes
	The user can delete a selected playlist. A dialog is shown to confirm the deletion.	Yes
Songs in playlist	When a specific playlist is selected, the songs of that playlist is shown in the list in the middle (songs in playlist list).	Yes
	The user can add a selected song to a selected playlist. The selected song will appear in the end of the playlist.	Yes
	The user can move a selected song in the playlist. *The move is temporary and resets when exiting the playlist.*	Yes
	The user can delete a selected song from a playlist. *All duplicates of the selected song will be removed.*	Yes

	The name of the selected playlist is displayed in the label above.	Yes
Playing Songs	The user interface contains controls for playing songs.	Yes
	The song being played is the currently selected song. *The user can only play songs after adding them to a playlist.* *The name of the currently playing song is displayed in the label above the Library.*	Yes
	When a song has finished playing, the next song in the list plays automatically.	Yes
	If the selection is set on a playlist, all songs in that playlist is being played in the stated sequence.	Yes
Saving playlists	Playlists and the song list are saved to a database using JDBC.	Yes
	Changes performed on playlists or songs are reflected in the database.	Yes
Filetypes	The application can play the common music files .wav and .mp3.	Yes
Controls	Songs can be shuffled on the currently selected playlist.	Yes
	When playing a song, the user can skip to the next song using the skip button.	Yes
	When playing a song, the user can access the previous song using the back button	Yes
	Loop will repeat any song when the loop button is clicked.	Yes
	The volume button can change the volume of the song currently playing.	Yes
	Play button is functional. It stops a song when playing and auto play songs whenever current song is playing. Bug: Slight delay when playing another song immediately from the playlist. Issue: pause/resume does not work due to the nature of the handle_stop method implemented to the play button (completely stops the song playing), cannot make both work	Yes

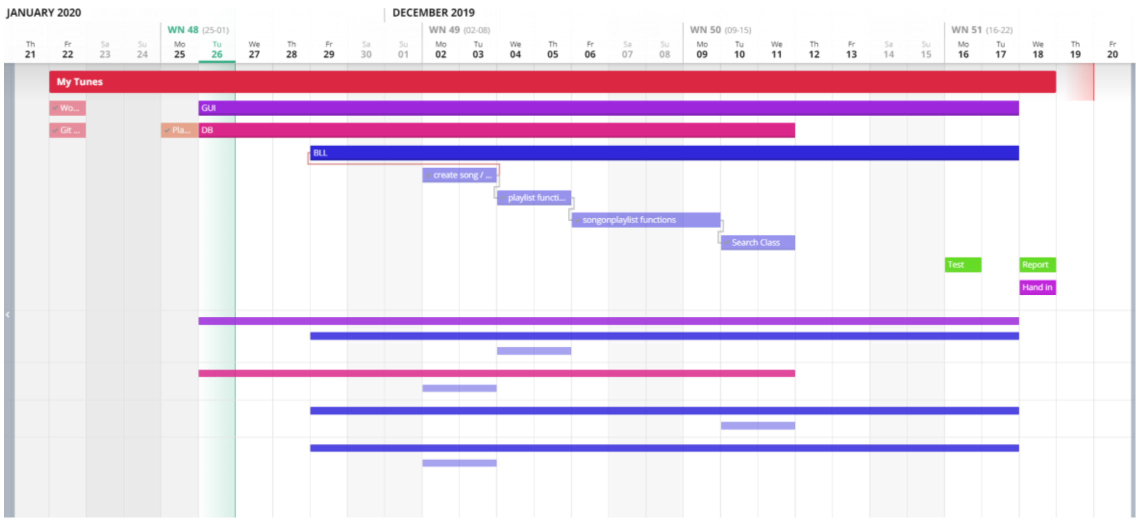
Process documentation

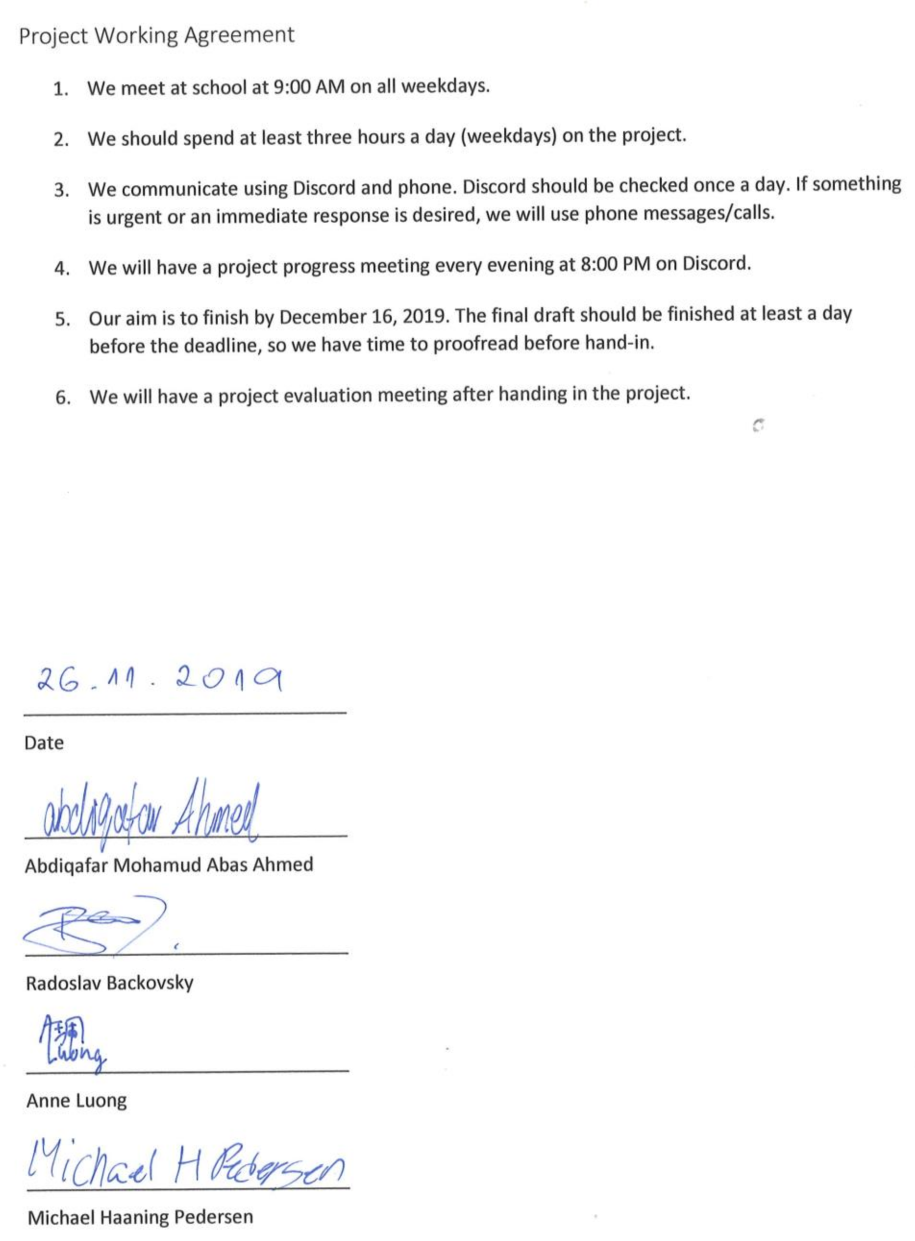
The project was planned using the Gantt Chart Tool Agantty. It did not go exactly as planned and some tasks were switched around. The deadlines were also not all upheld.
https://app.agantty.com/#/project/554897/task/1848801

Initial planning of project work:



Outcome due to many tasks being delayed:





Project Working Agreement

1. We meet at school at 9:00 AM on all weekdays.
2. We should spend at least three hours a day (weekdays) on the project.
3. We communicate using Discord and phone. Discord should be checked once a day. If something is urgent or an immediate response is desired, we will use phone messages/calls.
4. We will have a project progress meeting every evening at 8:00 PM on Discord.
5. Our aim is to finish by December 16, 2019. The final draft should be finished at least a day before the deadline, so we have time to proofread before hand-in.
6. We will have a project evaluation meeting after handing in the project.

26.11.2019

Date

abdiqafar Ahmed

Abdiqafar Mohamud Abas Ahmed

[illegible]

Radoslav Backovsky

[illegible]

Anne Luong

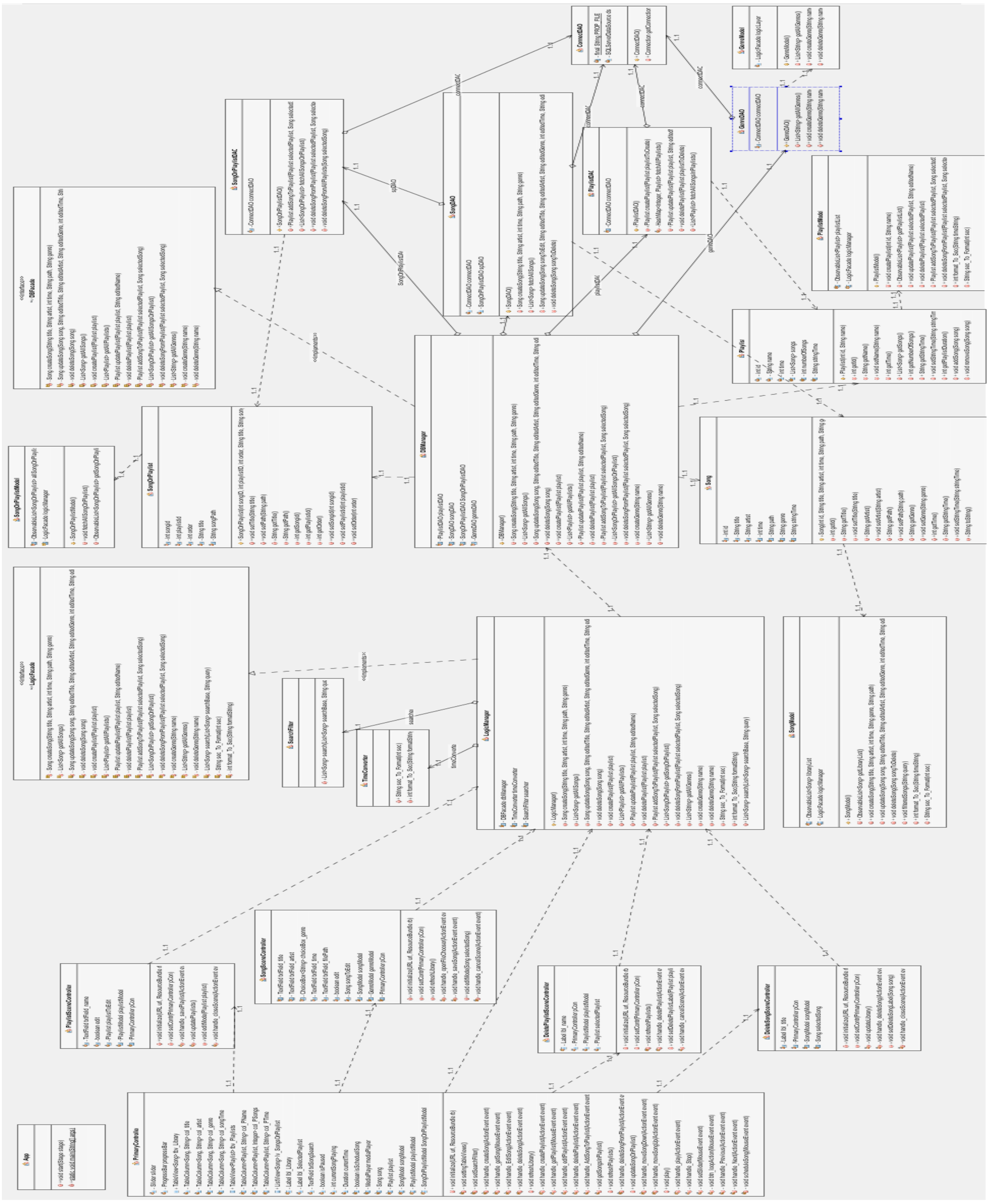
Michael H Pedersen

Michael Haaning Pedersen

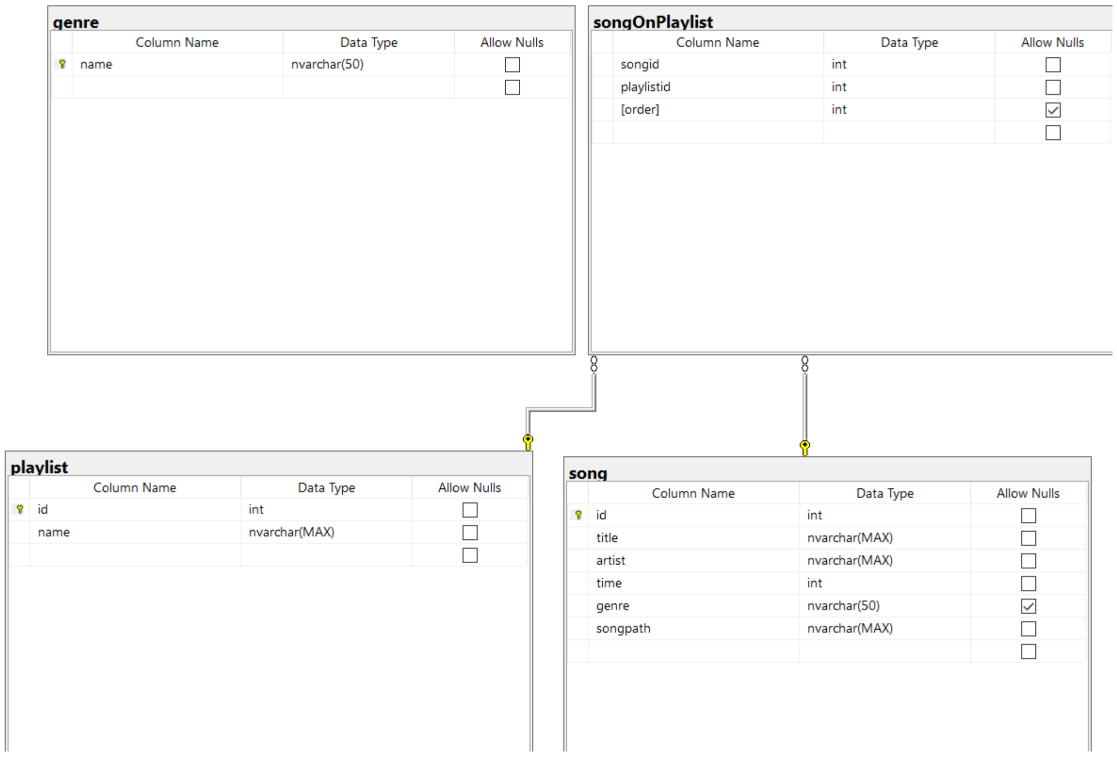
Application structure

The UML diagram is on the next page on its own as it would not fit otherwise.

SCO & SDE 1st semester
MyTunes – compulsory assignment #4
Team C: Abdiqafar Mohamud Abas Ahmed, Radoslav Backovsky, Anne Luong and Michael Haaning Pedersen



Data Storage



genre

Column Name	Data Type	Allow Nulls
🔑 name	nvarchar(50)	☐
		☐

songOnPlaylist

Column Name	Data Type	Allow Nulls
songid	int	☐
playlistid	int	☐
[order]	int	☑
		☐

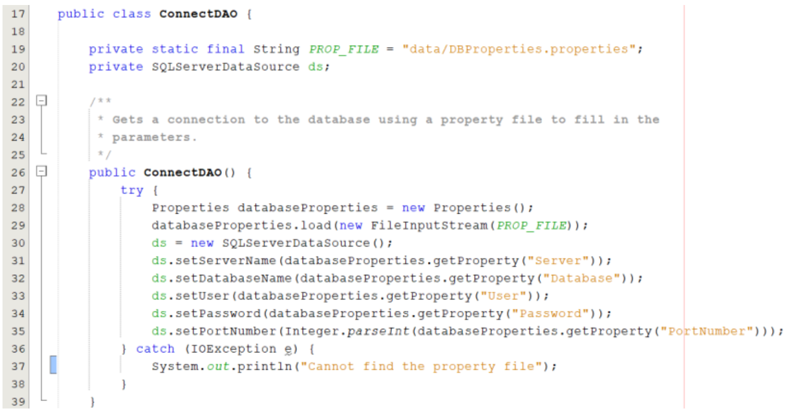
playlist

Column Name	Data Type	Allow Nulls
🔑 id	int	☐
name	nvarchar(MAX)	☐
		☐

song

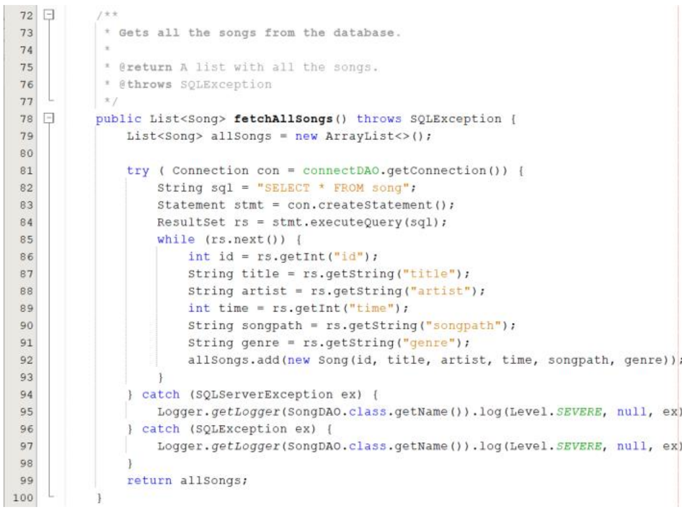
Column Name	Data Type	Allow Nulls
🔑 id	int	☐
title	nvarchar(MAX)	☐
artist	nvarchar(MAX)	☐
time	int	☐
genre	nvarchar(50)	☑
songpath	nvarchar(MAX)	☐
		☐

DAL implementations



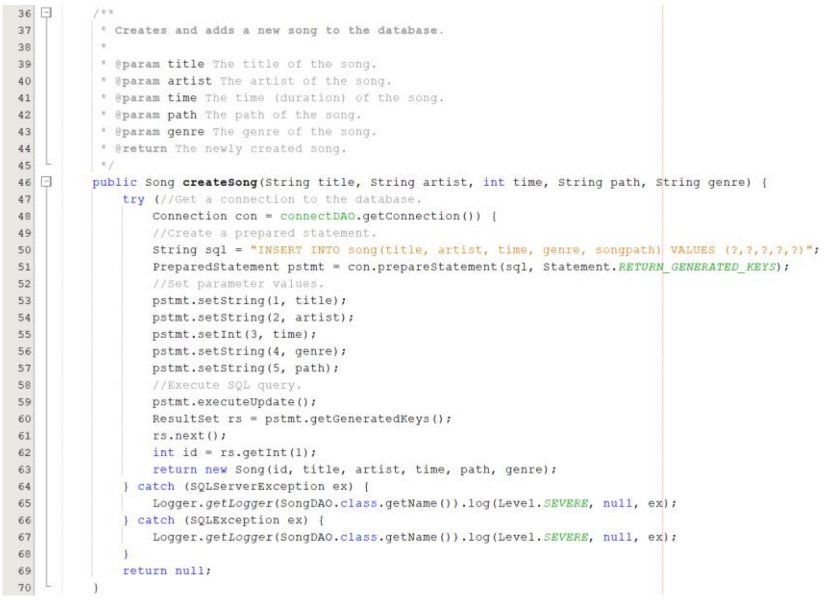
```java
public class ConnectDAO {

    private static final String PROP_FILE = "data/DBProperties.properties";
    private SQLServerDataSource ds;

    /**
     * Gets a connection to the database using a property file to fill in the
     * parameters.
     */
    public ConnectDAO() {
        try {
            Properties databaseProperties = new Properties();
            databaseProperties.load(new FileInputStream(PROP_FILE));
            ds = new SQLServerDataSource();
            ds.setServerName(databaseProperties.getProperty("Server"));
            ds.setDatabaseName(databaseProperties.getProperty("Database"));
            ds.setUser(databaseProperties.getProperty("User"));
            ds.setPassword(databaseProperties.getProperty("Password"));
            ds.setPortNumber(Integer.parseInt(databaseProperties.getProperty("PortNumber")));
        } catch (IOException e) {
            System.out.println("Cannot find the property file");
        }
    }
```

The method shows how to read a property file containing the information necessary to connect to the database.



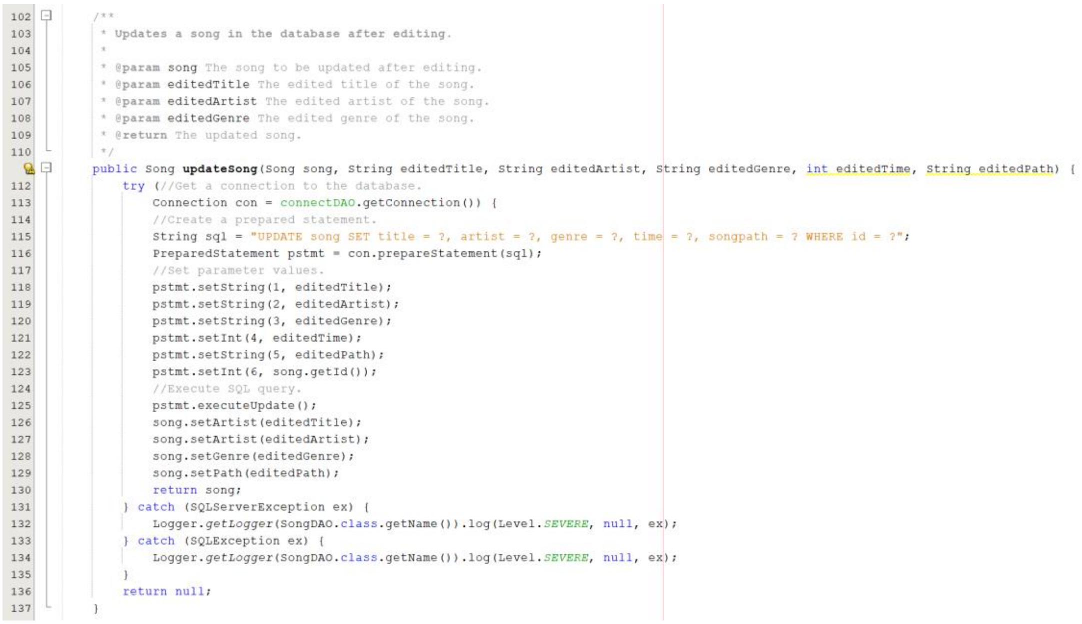
```java
/**
 * Gets all the songs from the database.
 *
 * @return A list with all the songs.
 * @throws SQLException
 */
public List<Song> fetchAllSongs() throws SQLException {
    List<Song> allSongs = new ArrayList<>();

    try ( Connection con = connectDAO.getConnection()) {
        String sql = "SELECT * FROM song";
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(sql);
        while (rs.next()) {
            int id = rs.getInt("id");
            String title = rs.getString("title");
            String artist = rs.getString("artist");
            int time = rs.getInt("time");
            String songpath = rs.getString("songpath");
            String genre = rs.getString("genre");
            allSongs.add(new Song(id, title, artist, time, songpath, genre));
        }
    } catch (SQLServerException ex) {
        Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
    } catch (SQLException ex) {
        Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return allSongs;
}
```

The method gives access to the information of all the songs stored in the database. First, a connection to the database is established. All the data stored in the table “song” is selected. The data returned is stored in the ResultSet object (result table). The returning ResultSet contains multiple rows and the rs.next() method is used to loop through each row of the ResultSet and add the data to an ArrayList. The ArrayList is returned.



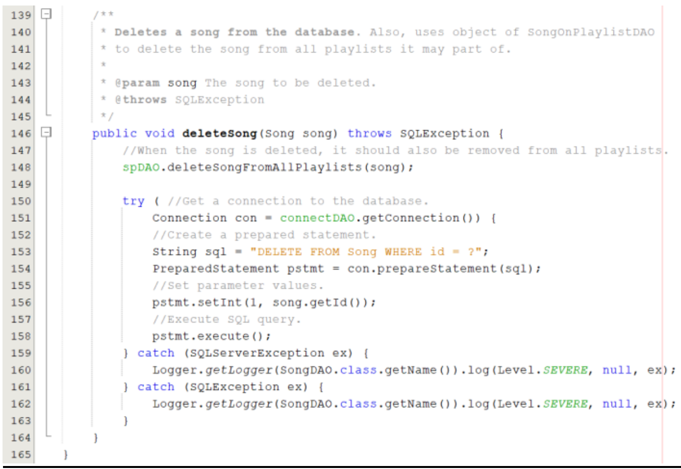
```java
/**
 * Creates and adds a new song to the database.
 *
 * @param title The title of the song.
 * @param artist The artist of the song.
 * @param time The time (duration) of the song.
 * @param path The path of the song.
 * @param genre The genre of the song.
 * @return The newly created song.
 */
public Song createSong(String title, String artist, int time, String path, String genre) {
    try (//Get a connection to the database.
        Connection con = connectDAO.getConnection()) {
        //Create a prepared statement.
        String sql = "INSERT INTO song(title, artist, time, genre, songpath) VALUES (?,?,?,?,?)";
        PreparedStatement pstmt = con.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        //Set parameter values.
        pstmt.setString(1, title);
        pstmt.setString(2, artist);
        pstmt.setInt(3, time);
        pstmt.setString(4, genre);
        pstmt.setString(5, path);
        //Execute SQL query.
        pstmt.executeUpdate();
        ResultSet rs = pstmt.getGeneratedKeys();
        rs.next();
        int id = rs.getInt(1);
        return new Song(id, title, artist, time, path, genre);
    } catch (SQLServerException ex) {
        Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
    } catch (SQLException ex) {
        Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return null;
}
```

The method inserts new data in the table “song”. First, a connection to the database is established. A prepared statement is created to execute the SQL INSERT INTO statement. The statement has parameters (labelled “?”) and the specific values to insert come from the song object passed from the BLL (and before that the GUI). When the values are inserted into the table, an id is auto-generated. The same method is used in PlaylistDAO and GenreDAO.

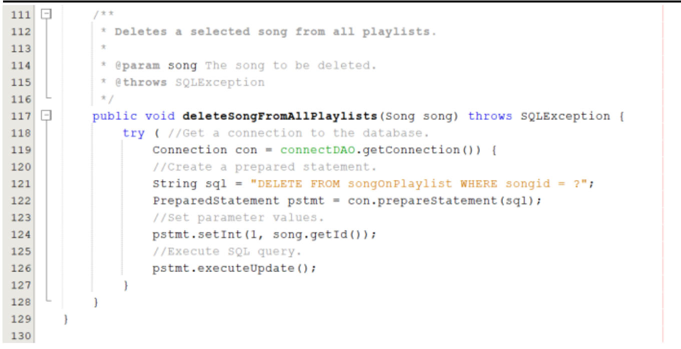


```java
    /**
     * Updates a song in the database after editing.
     *
     * @param song The song to be updated after editing.
     * @param editedTitle The edited title of the song.
     * @param editedArtist The edited artist of the song.
     * @param editedGenre The edited genre of the song.
     * @return The updated song.
     */
    public Song updateSong(Song song, String editedTitle, String editedArtist, String editedGenre, int editedTime, String editedPath) {
        try (//Get a connection to the database.
            Connection con = connectDAO.getConnection()) {
            //Create a prepared statement.
            String sql = "UPDATE song SET title = ?, artist = ?, genre = ?, time = ?, songpath = ? WHERE id = ?";
            PreparedStatement pstmt = con.prepareStatement(sql);
            //Set parameter values.
            pstmt.setString(1, editedTitle);
            pstmt.setString(2, editedArtist);
            pstmt.setString(3, editedGenre);
            pstmt.setInt(4, editedTime);
            pstmt.setString(5, editedPath);
            pstmt.setInt(6, song.getId());
            //Execute SQL query.
            pstmt.executeUpdate();
            song.setArtist(editedTitle);
            song.setArtist(editedArtist);
            song.setGenre(editedGenre);
            song.setPath(editedPath);
            return song;
        } catch (SQLServerException ex) {
            Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return null;
    }
```

The method uses the UPDATE statement to modify the existing data of a specific song in the “song” table. The id of the song is used to identify which record should be modified. The same method was used in PlaylistDAO.

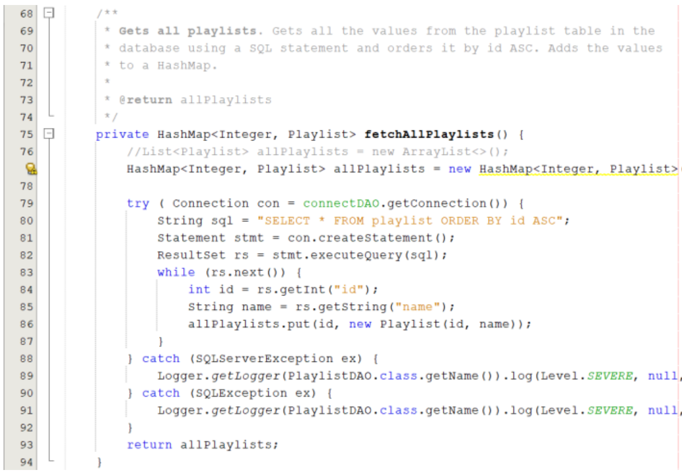


```java
    /**
     * Deletes a song from the database. Also, uses object of SongOnPlaylistDAO
     * to delete the song from all playlists it may part of.
     *
     * @param song The song to be deleted.
     * @throws SQLException
     */
    public void deleteSong(Song song) throws SQLException {
        //When the song is deleted, it should also be removed from all playlists.
        spDAO.deleteSongFromAllPlaylists(song);

        try ( //Get a connection to the database.
            Connection con = connectDAO.getConnection()) {
            //Create a prepared statement.
            String sql = "DELETE FROM Song WHERE id = ?";
            PreparedStatement pstmt = con.prepareStatement(sql);
            //Set parameter values.
            pstmt.setInt(1, song.getId());
            //Execute SQL query.
            pstmt.execute();
        } catch (SQLServerException ex) {
            Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(SongDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```



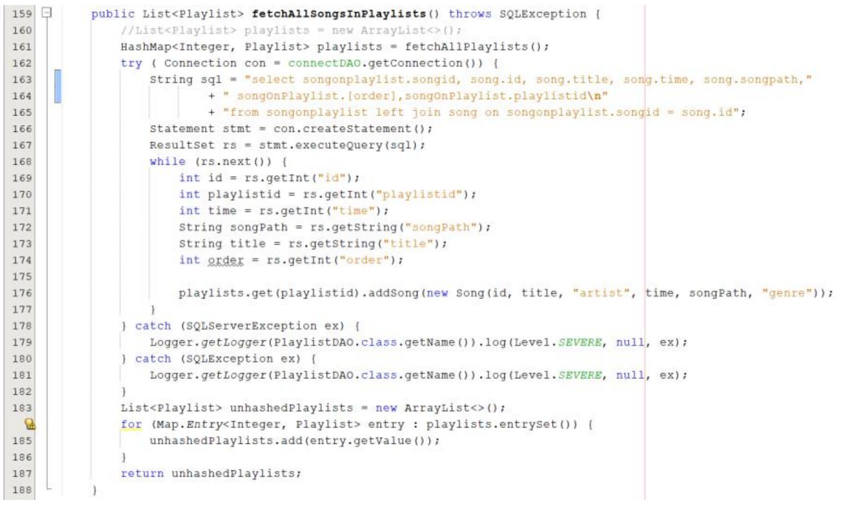
```java
    /**
     * Deletes a selected song from all playlists.
     *
     * @param song The song to be deleted.
     * @throws SQLException
     */
    public void deleteSongFromAllPlaylists(Song song) throws SQLException {
        try ( //Get a connection to the database.
            Connection con = connectDAO.getConnection()) {
            //Create a prepared statement.
            String sql = "DELETE FROM songOnPlaylist WHERE songid = ?";
            PreparedStatement pstmt = con.prepareStatement(sql);
            //Set parameter values.
            pstmt.setInt(1, song.getId());
            //Execute SQL query.
            pstmt.executeUpdate();
        }
    }
}
```

The method depicts the solution used to DELETE existing data in the “song” table. When the song is deleted from the “song” table, it will also be deleted from the “songOnPlaylist” table using a method from the SongOnPlaylistDAO. The method to delete records was used in PlaylistDAO and GenreDAO.



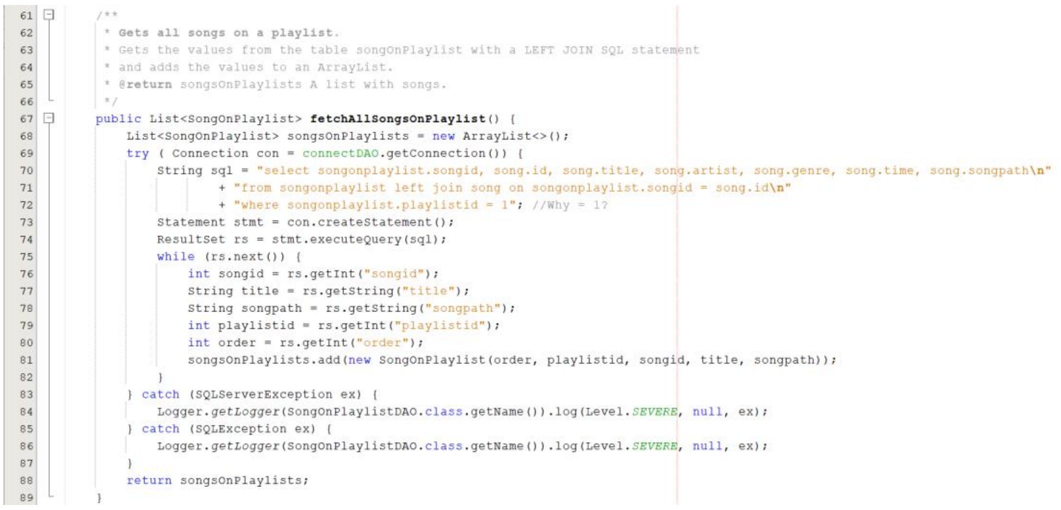
```
    /**
     * Gets all playlists. Gets all the values from the playlist table in the
     * database using a SQL statement and orders it by id ASC. Adds the values
     * to a HashMap.
     *
     * @return allPlaylists
     */
    private HashMap<Integer, Playlist> fetchAllPlaylists() {
        //List<Playlist> allPlaylists = new ArrayList<>();
        HashMap<Integer, Playlist> allPlaylists = new HashMap<Integer, Playlist>();

        try ( Connection con = connectDAO.getConnection()) {
            String sql = "SELECT * FROM playlist ORDER BY id ASC";
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                allPlaylists.put(id, new Playlist(id, name));
            }
        } catch (SQLServerException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return allPlaylists;
    }
```

The method gets access to the “playlist” table, which stores the name of a playlist and its auto-generated playlist id. All records in the table are selected and ordered ascending. The records are stored in a HashMap instead of an ArrayList to make certain the program will not crash if a playlist is created or deleted. The HashMap is returned.



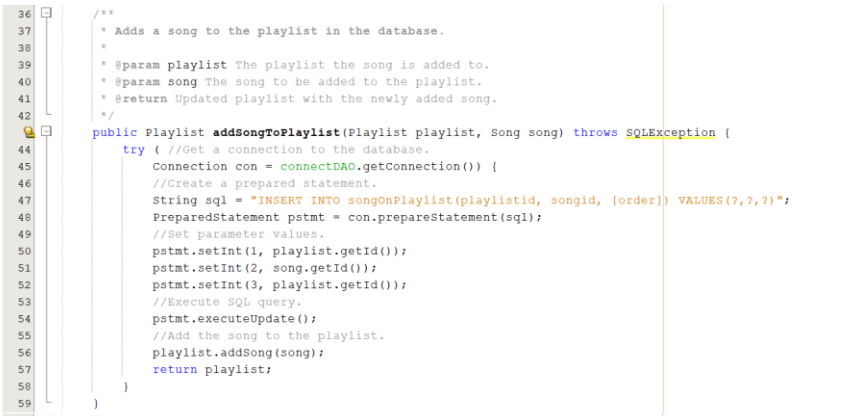
```
    public List<Playlist> fetchAllSongsInPlaylists() throws SQLException {
        //List<Playlist> playlists = new ArrayList<>();
        HashMap<Integer, Playlist> playlists = fetchAllPlaylists();
        try ( Connection con = connectDAO.getConnection()) {
            String sql = "select songonplaylist.songid, song.id, song.title, song.time, song.songpath,"
                    + " songOnPlaylist.[order],songOnPlaylist.playlistid\n"
                    + "from songonplaylist left join song on songonplaylist.songid = song.id";
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                int id = rs.getInt("id");
                int playlistid = rs.getInt("playlistid");
                int time = rs.getInt("time");
                String songPath = rs.getString("songPath");
                String title = rs.getString("title");
                int order = rs.getInt("order");

                playlists.get(playlistid).addSong(new Song(id, title, "artist", time, songPath, "genre"));
            }
        } catch (SQLServerException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        List<Playlist> unhashedPlaylists = new ArrayList<>();
        for (Map.Entry<Integer, Playlist> entry : playlists.entrySet()) {
            unhashedPlaylists.add(entry.getValue());
        }
        return unhashedPlaylists;
    }
```

The method uses the SQL JOIN statement to get the song id and playlist id from the “songOnPlaylist” table. It reads the “song” table to obtain the id, title and path of the song. The information is added to a list with a *for loop* in order convert from a HashMap to an ArrayList.

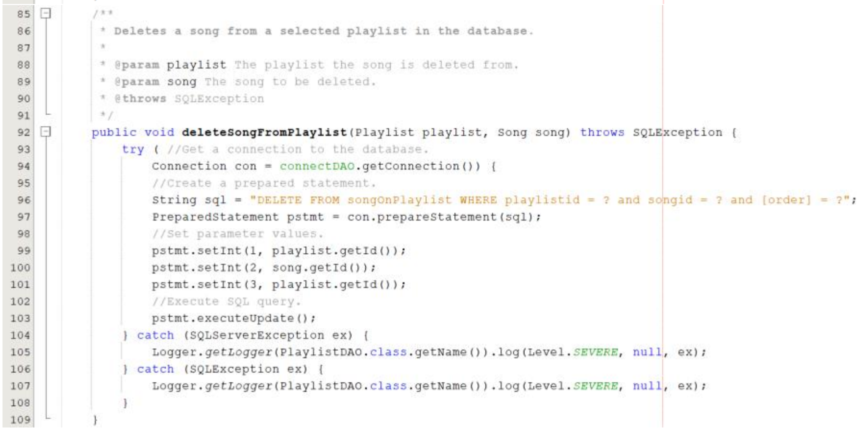


```
    /**
     * Gets all songs on a playlist.
     * Gets the values from the table songOnPlaylist with a LEFT JOIN SQL statement
     * and adds the values to an ArrayList.
     * @return songsOnPlaylists A list with songs.
     */
    public List<SongOnPlaylist> fetchAllSongsOnPlaylist() {
        List<SongOnPlaylist> songsOnPlaylists = new ArrayList<>();
        try ( Connection con = connectDAO.getConnection()) {
            String sql = "select songonplaylist.songid, song.id, song.title, song.artist, song.genre, song.time, song.songpath\n"
                    + "from songonplaylist left join song on songonplaylist.songid = song.id\n"
                    + "where songonplaylist.playlistid = 1"; //Why = 1?
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                int songid = rs.getInt("songid");
                String title = rs.getString("title");
                String songpath = rs.getString("songpath");
                int playlistid = rs.getInt("playlistid");
                int order = rs.getInt("order");
                songsOnPlaylists.add(new SongOnPlaylist(order, playlistid, songid, title, songpath));
            }
        } catch (SQLServerException ex) {
            Logger.getLogger(SongOnPlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(SongOnPlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return songsOnPlaylists;
    }
```

The method uses a SQL statement with a left JOIN to get results from the table songsOnPlaylist. The results are added to a list and the list is returned.



```
    /**
     * Adds a song to the playlist in the database.
     *
     * @param playlist The playlist the song is added to.
     * @param song The song to be added to the playlist.
     * @return Updated playlist with the newly added song.
     */
    public Playlist addSongToPlaylist(Playlist playlist, Song song) throws SQLException {
        try ( //Get a connection to the database.
            Connection con = connectDAO.getConnection()) {
            //Create a prepared statement.
            String sql = "INSERT INTO songOnPlaylist(playlistid, songid, [order]) VALUES(?,?,?)";
            PreparedStatement pstmt = con.prepareStatement(sql);
            //Set parameter values.
            pstmt.setInt(1, playlist.getId());
            pstmt.setInt(2, song.getId());
            pstmt.setInt(3, playlist.getId());
            //Execute SQL query.
            pstmt.executeUpdate();
            //Add the song to the playlist.
            playlist.addSong(song);
            return playlist;
        }
    }
```

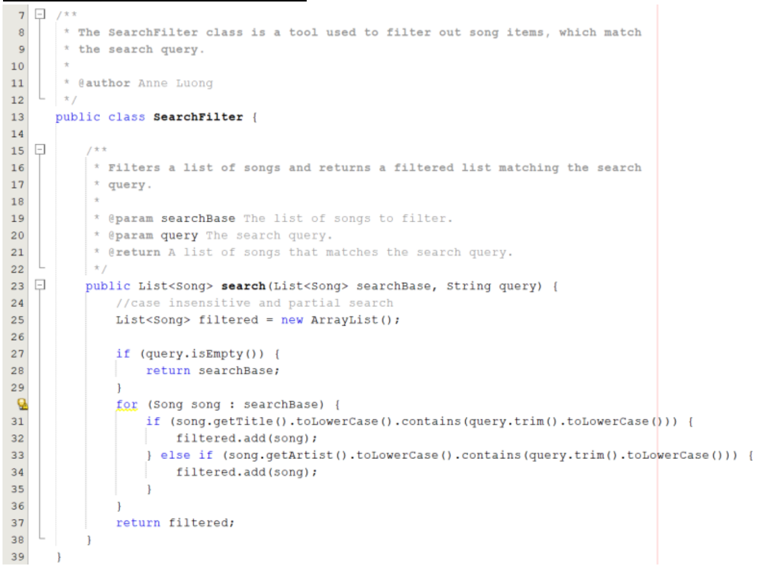


```
    /**
     * Deletes a song from a selected playlist in the database.
     *
     * @param playlist The playlist the song is deleted from.
     * @param song The song to be deleted.
     * @throws SQLException
     */
    public void deleteSongFromPlaylist(Playlist playlist, Song song) throws SQLException {
        try ( //Get a connection to the database.
            Connection con = connectDAO.getConnection()) {
            //Create a prepared statement.
            String sql = "DELETE FROM songOnPlaylist WHERE playlistid = ? and songid = ? and [order] = ?";
            PreparedStatement pstmt = con.prepareStatement(sql);
            //Set parameter values.
            pstmt.setInt(1, playlist.getId());
            pstmt.setInt(2, song.getId());
            pstmt.setInt(3, playlist.getId());
            //Execute SQL query.
            pstmt.executeUpdate();
        } catch (SQLServerException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(PlaylistDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

The current implementation of adding and deleting a song on a playlist has a problem. As it can be seen above, the order value (representing the placement of a song in a specific playlist) is currently the playlist id. This results in a problem with duplicate songs on a playlist. If three duplicate songs are on one playlist, the user cannot remove only one of the three. Instead the user is forced to remove all three duplicates.

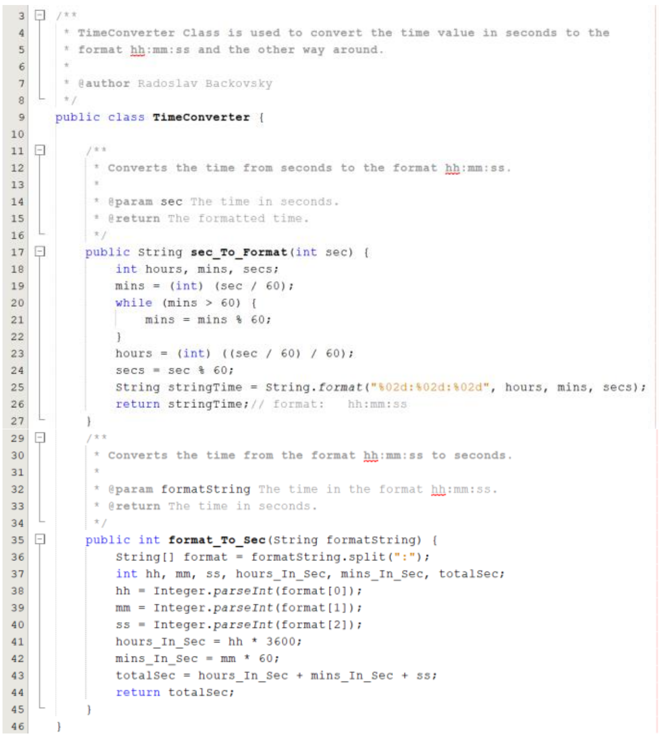
The solution to this would be to make the order value different for each song on the playlist. It would be desirable if the order value could represent the current placement of a song on the playlist. Due to time constraints, it was not implemented.

BLL implementations



```java
/**
 * The SearchFilter class is a tool used to filter out song items, which match
 * the search query.
 *
 * @author Anne Luong
 */
public class SearchFilter {

    /**
     * Filters a list of songs and returns a filtered list matching the search
     * query.
     *
     * @param searchBase The list of songs to filter.
     * @param query The search query.
     * @return A list of songs that matches the search query.
     */
    public List<Song> search(List<Song> searchBase, String query) {
        //case insensitive and partial search
        List<Song> filtered = new ArrayList();

        if (query.isEmpty()) {
            return searchBase;
        }
        for (Song song : searchBase) {
            if (song.getTitle().toLowerCase().contains(query.trim().toLowerCase())) {
                filtered.add(song);
            } else if (song.getArtist().toLowerCase().contains(query.trim().toLowerCase())) {
                filtered.add(song);
            }
        }
        return filtered;
    }
}
```

The method creates and returns a new ArrayList, where the list is filtered to only hold items containing the search word entered in the search bar.



```java
/**
 * TimeConverter Class is used to convert the time value in seconds to the
 * format hh:mm:ss and the other way around.
 *
 * @author Radoslav Backovsky
 */
public class TimeConverter {

    /**
     * Converts the time from seconds to the format hh:mm:ss.
     *
     * @param sec The time in seconds.
     * @return The formatted time.
     */
    public String sec_To_Format(int sec) {
        int hours, mins, secs;
        mins = (int) (sec / 60);
        while (mins > 60) {
            mins = mins % 60;
        }
        hours = (int) ((sec / 60) / 60);
        secs = sec % 60;
        String stringTime = String.format("%02d:%02d:%02d", hours, mins, secs);
        return stringTime;// format:   hh:mm:ss
    }
    /**
     * Converts the time from the format hh:mm:ss to seconds.
     *
     * @param formatString The time in the format hh:mm:ss.
     * @return The time in seconds.
     */
    public int format_To_Sec(String formatString) {
        String[] format = formatString.split(":");
        int hh, mm, ss, hours_In_Sec, mins_In_Sec, totalSec;
        hh = Integer.parseInt(format[0]);
        mm = Integer.parseInt(format[1]);
        ss = Integer.parseInt(format[2]);
        hours_In_Sec = hh * 3600;
        mins_In_Sec = mm * 60;
        totalSec = hours_In_Sec + mins_In_Sec + ss;
        return totalSec;
    }
}
```

The two methods are used to convert between sec (int) and the time format hh:mm:ss (String). The methods are used for the song duration and playlist duration. The first method converts from sec (int) and the format (String), so the duration can be displayed in the desired time format in the MyTunes app for the user. The second method is used to convert from the formatted time (String) to sec (int) for database storage. The duration is stored in seconds for easier calculation of the playlist duration.

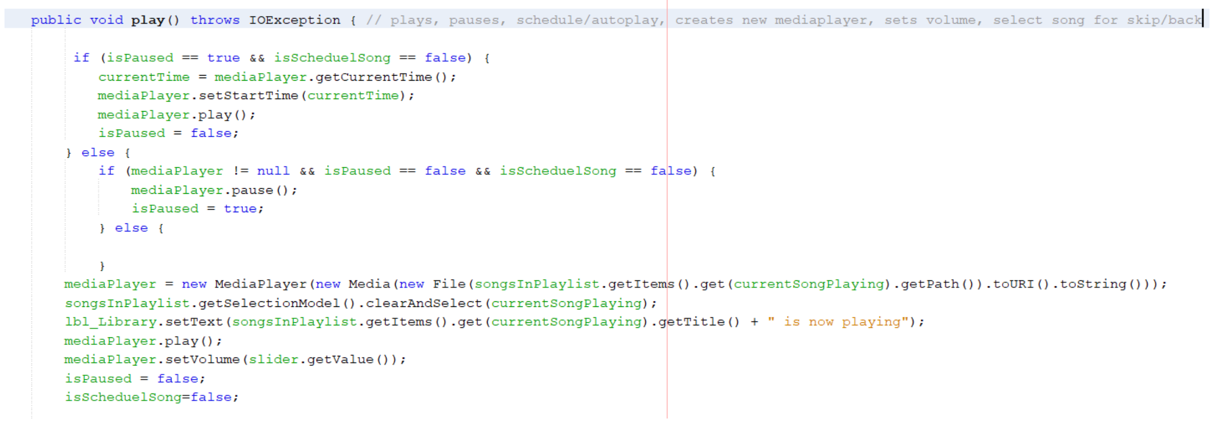
GUI implementations

The music player can function through the buttons that have been implemented inside Netbeans and SceneBuilder. Most of the controls are inside the PrimaryController.java class. The other controllers are simply scenes that pop to either add/delete/edit songs and playlists e.g. SongSceneController/PlaylistSceneController as FileChooser and name changer.

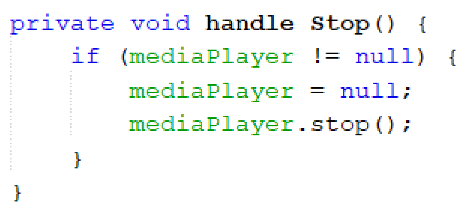
SceneBuilder is what has been used to create the GUI and its buttons, which they consist of imageviews. The GUI's layout resembles that of any other popular music player (Spotify, iTunes) so it has most core functions in one primary scene.

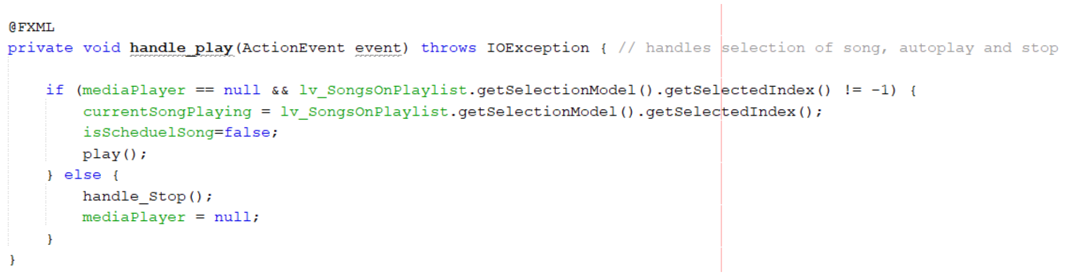
Let's cover the core buttons: the play button, skip/previous buttons:

The play button's method consists of all of the required uses for the player as it covers the playing, pausing and resuming of songs. The if statement covers the pause/resume functions. The play method also covers the auto-play function and is able to show the user what song is currently playing (by having the library's label text changed to the current song "is now playing"), which can be useful for users with huge amounts of songs stored.


```
public void play() throws IOException { // plays, pauses, schedule/autoplay, creates new mediaplayer, sets volume, select song for skip/back

     if (isPaused == true && isScheduelSong == false) {
        currentTime = mediaPlayer.getCurrentTime();
        mediaPlayer.setStartTime(currentTime);
        mediaPlayer.play();
        isPaused = false;
    } else {
        if (mediaPlayer != null && isPaused == false && isScheduelSong == false) {
            mediaPlayer.pause();
            isPaused = true;
        } else {

        }
    mediaPlayer = new MediaPlayer(new Media(new File(songsInPlaylist.getItems().get(currentSongPlaying).getPath()).toURI().toString()));
    songsInPlaylist.getSelectionModel().clearAndSelect(currentSongPlaying);
    lbl_Library.setText(songsInPlaylist.getItems().get(currentSongPlaying).getTitle() + " is now playing");
    mediaPlayer.play();
    mediaPlayer.setVolume(slider.getValue());
    isPaused = false;
    isScheduelSong=false;
```

Originally the play() logic was all bundled and cluttered and caused issues with the skip and previous buttons, which is why we've added some of the code to handle_play and created handle_stop to prevent the songs from stacking whenever play button is pressed more than once. However, it replaced the pause function and now won't pause/resume due to this. This isn't really a big loss as without handle_stop we'd not be able to even use the skip/previous buttons (even if they both functions perfectly).

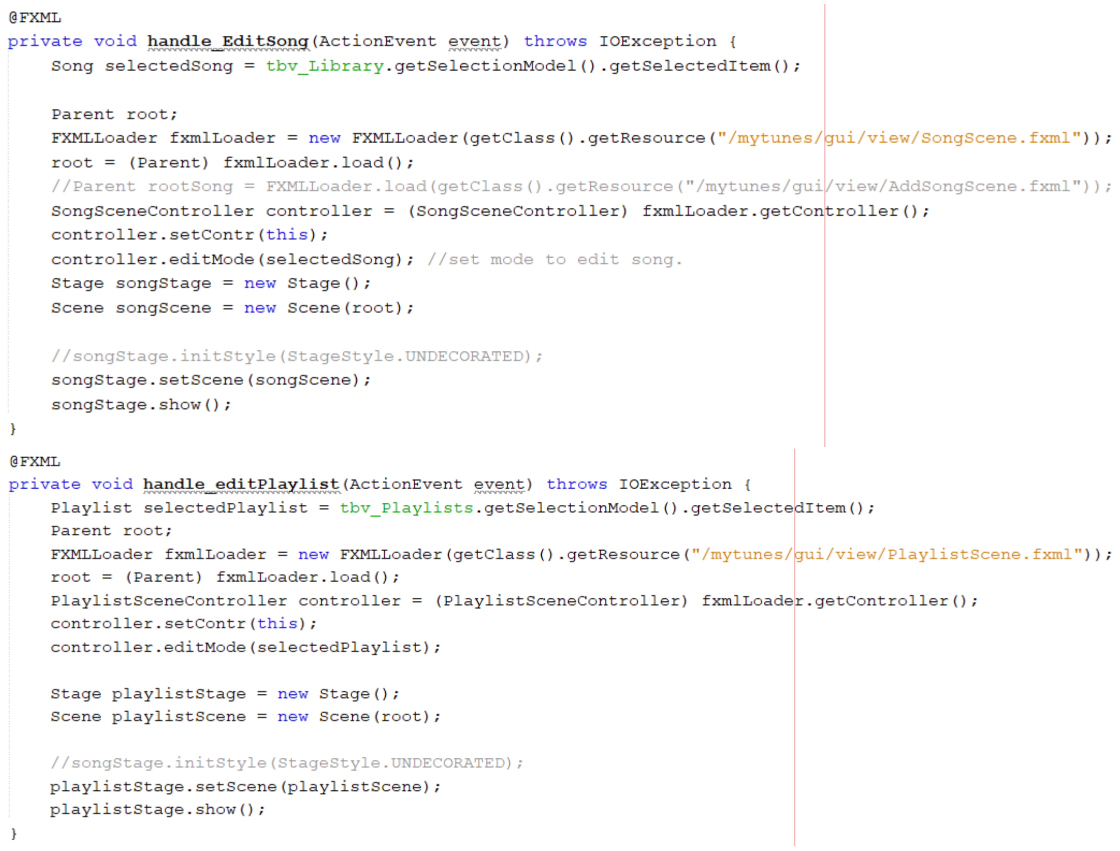

```
private void handle Stop() {
    if (mediaPlayer != null) {
        mediaPlayer = null;
        mediaPlayer.stop();
    }
}
```



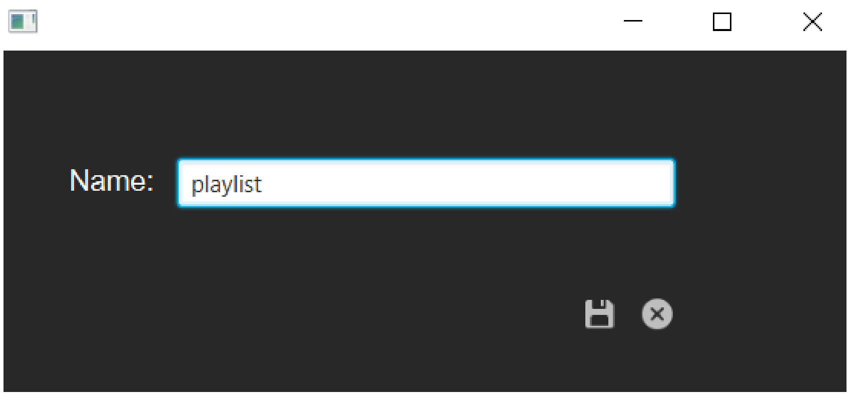
```
@FXML
private void handle_play(ActionEvent event) throws IOException { // handles selection of song, autoplay and stop

    if (mediaPlayer == null && lv_SongsOnPlaylist.getSelectionModel().getSelectedIndex() != -1) {
        currentSongPlaying = lv_SongsOnPlaylist.getSelectionModel().getSelectedIndex();
        isScheduelSong=false;
        play();
    } else {
        handle_Stop();
        mediaPlayer = null;
    }
}
```

Handle_play only had a play(); as a line code but after the cluttered public void play() method started to prevent the skip/back buttons to function, it was then necessary to move some of the code their and refactor it e.g. adding if statement. It changed into a constructor once handle_stop was implemented

setSchedule and isPaused are two instance variables used only for these three methods, as it’s the auto play for the songs – it begins whenever the song is ending and loops the songs endlessly.

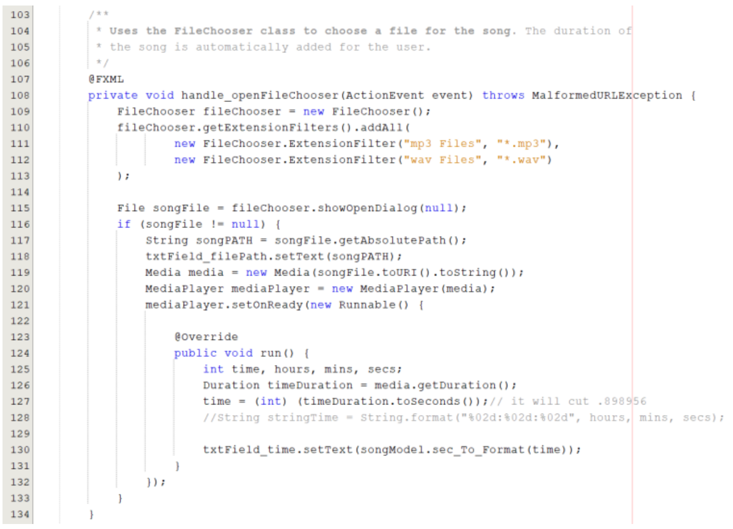


```
@FXML
private void handle_EditSong(ActionEvent event) throws IOException {
    Song selectedSong = tbv_Library.getSelectionModel().getSelectedItem();

    Parent root;
    FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("/mytunes/gui/view/SongScene.fxml"));
    root = (Parent) fxmlLoader.load();
    //Parent rootSong = FXMLLoader.load(getClass().getResource("/mytunes/gui/view/AddSongScene.fxml"));
    SongSceneController controller = (SongSceneController) fxmlLoader.getController();
    controller.setContr(this);
    controller.editMode(selectedSong); //set mode to edit song.
    Stage songStage = new Stage();
    Scene songScene = new Scene(root);

    //songStage.initStyle(StageStyle.UNDECORATED);
    songStage.setScene(songScene);
    songStage.show();
}
@FXML
private void handle_editPlaylist(ActionEvent event) throws IOException {
    Playlist selectedPlaylist = tbv_Playlists.getSelectionModel().getSelectedItem();
    Parent root;
    FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("/mytunes/gui/view/PlaylistScene.fxml"));
    root = (Parent) fxmlLoader.load();
    PlaylistSceneController controller = (PlaylistSceneController) fxmlLoader.getController();
    controller.setContr(this);
    controller.editMode(selectedPlaylist);

    Stage playlistStage = new Stage();
    Scene playlistScene = new Scene(root);

    //songStage.initStyle(StageStyle.UNDECORATED);
    playlistStage.setScene(playlistScene);
    playlistStage.show();
}
```

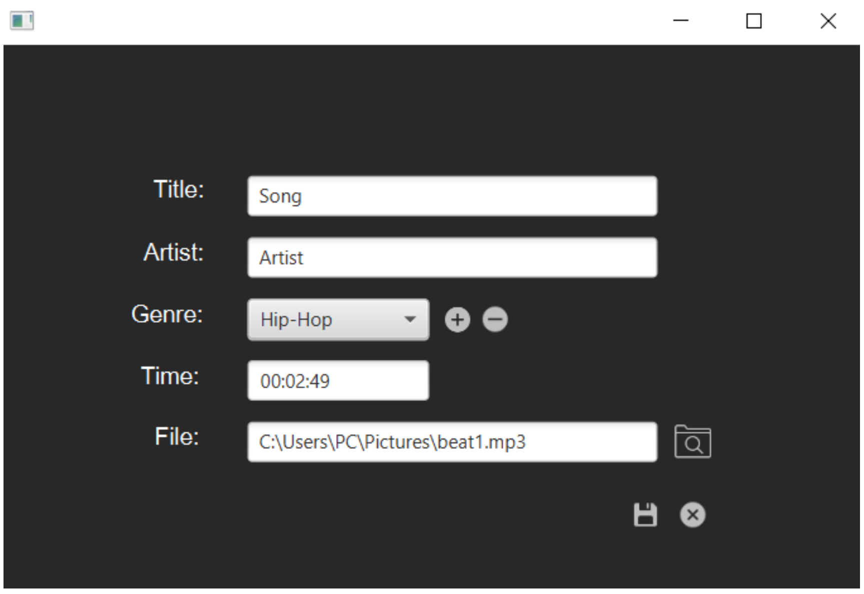


Name: playlist

And handle_createSong simply adds a song to the library by choosing the file in the FileChooser. This is done by having the handle_OpenFileChooser from the SongSceneController deal with the song's details. handle_editSong edits the song's details in the same scene. A Boolean is used to change the mode to edit mode.

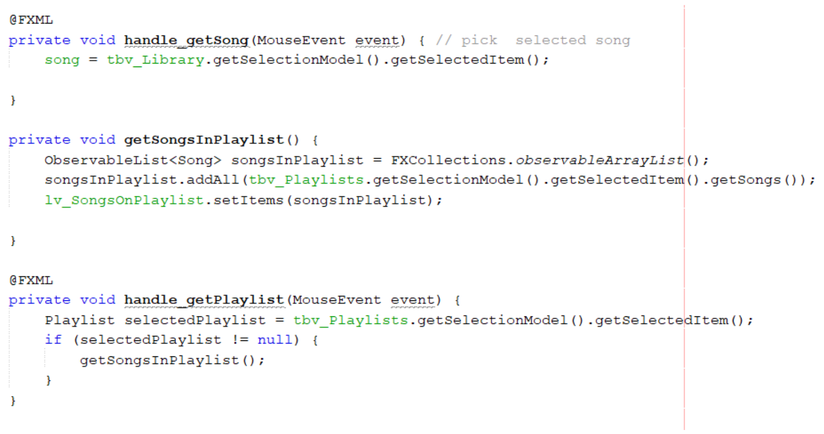


```java
/**
 * Uses the FileChooser class to choose a file for the song. The duration of
 * the song is automatically added for the user.
 */
@FXML
private void handle_openFileChooser(ActionEvent event) throws MalformedURLException {
    FileChooser fileChooser = new FileChooser();
    fileChooser.getExtensionFilters().addAll(
            new FileChooser.ExtensionFilter("mp3 Files", "*.mp3"),
            new FileChooser.ExtensionFilter("wav Files", "*.wav")
    );

    File songFile = fileChooser.showOpenDialog(null);
    if (songFile != null) {
        String songPATH = songFile.getAbsolutePath();
        txtField_filePath.setText(songPATH);
        Media media = new Media(songFile.toURI().toString());
        MediaPlayer mediaPlayer = new MediaPlayer(media);
        mediaPlayer.setOnReady(new Runnable() {

            @Override
            public void run() {
                int time, hours, mins, secs;
                Duration timeDuration = media.getDuration();
                time = (int) (timeDuration.toSeconds());// it will cut .898956
                //String stringTime = String.format("%02d:%02d:%02d", hours, mins, secs);

                txtField_time.setText(songModel.sec_To_Format(time));
            }
        });
    }
}
```



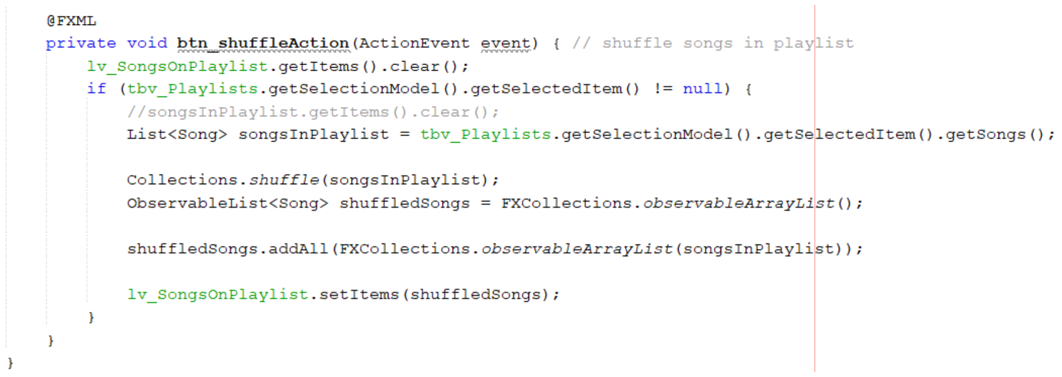
Title: Song
Artist: Artist
Genre: Hip-Hop
Time: 00:02:49
File: C:\Users\PC\Pictures\beat1.mp3

The methods below allow the user to select the song and playlist then click on whatever action they want e.g. delete/edit/add song.



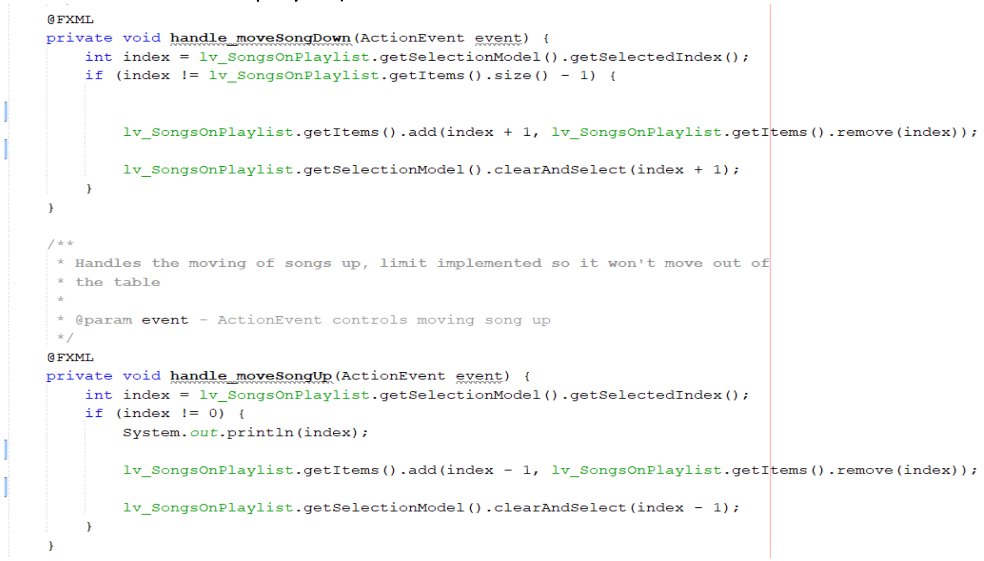
```
@FXML
private void handle_getSong(MouseEvent event) { // pick  selected song
    song = tbv_Library.getSelectionModel().getSelectedItem();

}

private void getSongsInPlaylist() {
    ObservableList<Song> songsInPlaylist = FXCollections.observableArrayList();
    songsInPlaylist.addAll(tbv_Playlists.getSelectionModel().getSelectedItem().getSongs());
    lv_SongsOnPlaylist.setItems(songsInPlaylist);

}

@FXML
private void handle_getPlaylist(MouseEvent event) {
    Playlist selectedPlaylist = tbv_Playlists.getSelectionModel().getSelectedItem();
    if (selectedPlaylist != null) {
        getSongsInPlaylist();
    }
}
```

A useful function for the user to use is the shuffle button as it can randomly shuffle the songs within the playlist, rather than the library. This is done by using ObservableArrayLists for the list view SongsOnPlaylist.



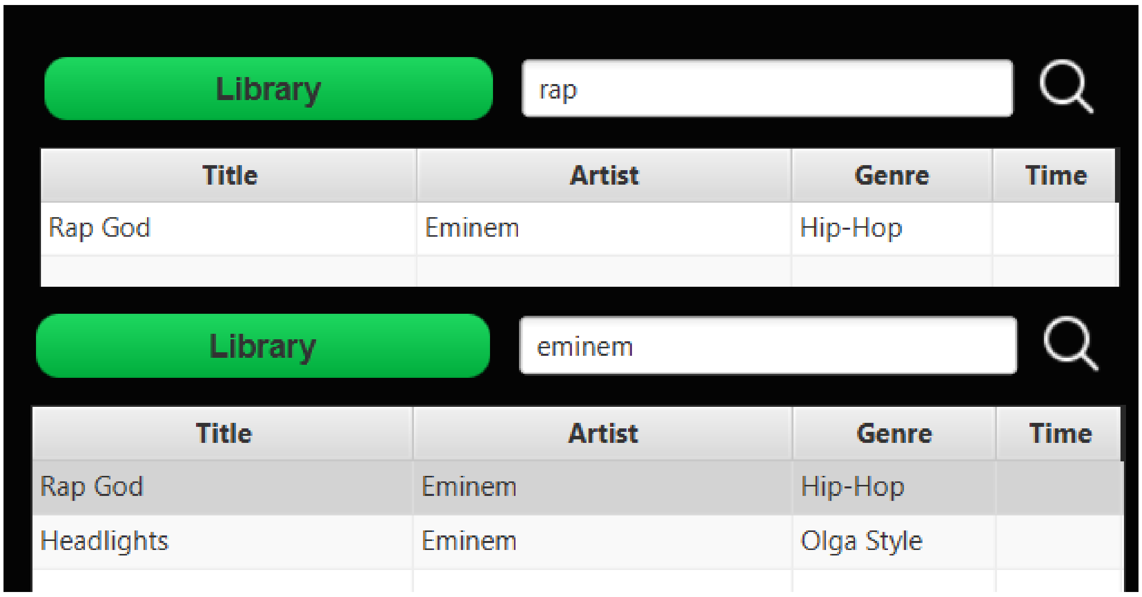
```
@FXML
private void btn_shuffleAction(ActionEvent event) { // shuffle songs in playlist
    lv_SongsOnPlaylist.getItems().clear();
    if (tbv_Playlists.getSelectionModel().getSelectedItem() != null) {
        //songsInPlaylist.getItems().clear();
        List<Song> songsInPlaylist = tbv_Playlists.getSelectionModel().getSelectedItem().getSongs();

        Collections.shuffle(songsInPlaylist);
        ObservableList<Song> shuffledSongs = FXCollections.observableArrayList();

        shuffledSongs.addAll(FXCollections.observableArrayList(songsInPlaylist));

        lv_SongsOnPlaylist.setItems(shuffledSongs);
    }
}
}
```

These two methods move the songs up and down inside the playlist. They simply select the songs and move them depending on where they are (e.g. a limit is set so they don't get "removed" from the playlist).



```
@FXML
private void handle_moveSongDown(ActionEvent event) {
    int index = lv_SongsOnPlaylist.getSelectionModel().getSelectedIndex();
    if (index != lv_SongsOnPlaylist.getItems().size() - 1) {


        lv_SongsOnPlaylist.getItems().add(index + 1, lv_SongsOnPlaylist.getItems().remove(index));

        lv_SongsOnPlaylist.getSelectionModel().clearAndSelect(index + 1);
    }
}

/**
 * Handles the moving of songs up, limit implemented so it won't move out of
 * the table
 *
 * @param event - ActionEvent controls moving song up
 */
@FXML
private void handle_moveSongUp(ActionEvent event) {
    int index = lv_SongsOnPlaylist.getSelectionModel().getSelectedIndex();
    if (index != 0) {
        System.out.println(index);

        lv_SongsOnPlaylist.getItems().add(index - 1, lv_SongsOnPlaylist.getItems().remove(index));

        lv_SongsOnPlaylist.getSelectionModel().clearAndSelect(index - 1);
    }
}
```

Search filter has also been included and is able to distinguish between name and artist, same goes for genre.



rap

Title	Artist	Genre	Time
Rap God	Eminem	Hip-Hop	

eminem

Title	Artist	Genre	Time
Rap God	Eminem	Hip-Hop	
Headlights	Eminem	Olga Style	

```java
private void setSearchFilter() {
    //Set the filter Predicate when the filter changes. Any changes to the
    //search textfield activates the filter.
    txtSongSearch.textProperty().addListener((obs, oldVal, newVal) -> {
        songModel.filteredSongs(newVal);
    });
}
```

Loop button has been included, it's able to just infinitely repeat a song picked by the user. This is done by using setCycleCount(MediaPlayer.INDEFINITE) (same goes for Duration.ZERO).

```java
@FXML
private void btn_loopAction(ActionEvent event) {
    mediaPlayer.setOnEndOfMedia(() -> {
        if (mediaPlayer != null) {
        }
        mediaPlayer.seek(Duration.ZERO);
        mediaPlayer.setCycleCount(MediaPlayer.INDEFINITE);
    });
}
```

Source Control

The Git system was used throughout the project for source control. From the first day, a shared repository, MyTunesC, was created for easy collaboration throughout the project. It was agreed among the team members that only functioning code would be committed to the master branch (default branch). Each member created a separate development branch, when working on an individual task. When the task was completed, the development branch was merged with the master branch. A designated team member was responsible for merging the master with each development branch. This approach was used until the last few days where some code was lost after a merge with a refactoring branch. Hence, the last changes were committed and pushed directly to the default branch.

Source code

GitHub repository:
https://github.com/mrbacky/MyTunesC