

CST2550

**Software Engineering Management and
Development**

Coursework 2

Submission: Friday 8th May 2020, 17:00hrs

CAMPUS: Mauritius

TABLE OF CONTENTS

Abstract	2
Introduction.....	2
Design	3
Wireframe	3
Main Window	3
Adding new Songs Window.....	4
Media Player Window	4
Pseudo Code	5
Add Song to Library	5
Testing	6
Test Table.....	6
Evidence of Testing	6
Conclusions.....	7
Limitations	7
Improvements.....	7
References.....	8
Appendices	9

ABSTRACT

The karaoke was built using the data structures: HashMap and Linked list. Data encapsulation was used, that is data attributes are kept private and can only be accessed through setters and getters. Also, it was used so that whenever the implementation of the class changes, the interface remains the same. Furthermore, All the functionalities of the application namely: Add to Library, Search from Library, Refresh Library, add to playlist, delete from Playlist and mediaplayer work fine. All 7 tests conducted were passed (refer to Testing Section pg-6) in a time of 0.083s.

INTRODUCTION

This report is a documentation to allow a user to better understand the source code of the application and the ideas behind this project. It is based on a karaoke application built using Java. The application can store over millions of songs and has several functionalities such as: Displaying its whole Library, it also allows the user to add a new song to its library, Songs can also be searched in the Library by song title. Moreover, the user can create his own playlist and add songs chosen from the library to it. The playlist can also be edited by removing unwanted songs. Finally, the User can use the media player functionality to play videos of songs from his playlist.

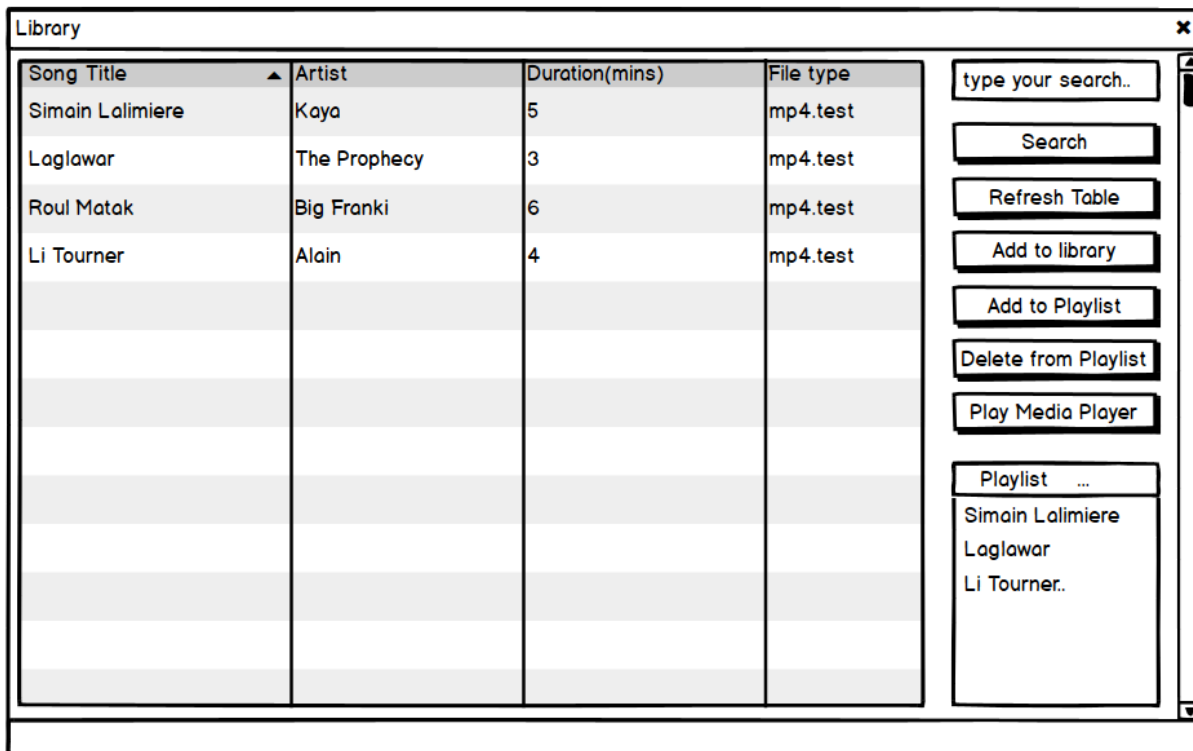
This paper is about describing and explaining what the application does and how it does it. Technical details are given about how this project was thought, approached and completed. The purpose of this report is to help the user have a clear-cut understanding of what the application is about and how it works. To better achieve this this paper was divided into 7 sections which are as follows:

- **Abstract:** A description of the work results and components
- **Introduction:** A brief introduction of the project along with report layout description.
- **Design phase:** Consists of 3 sections namely:
 - Graphical User Interface Mock-ups
 - Pseudo Codes
 - Analysis (Analysis of Time complexity and justification on choice of Data Structure)
- **Testing:** The testing approach used has been broken down and explained in the testing section.
- **Conclusion:** A summary of the project, the limitations and a critical reflection on the project.
- **References**
- **Appendices**

DESIGN

Wireframe

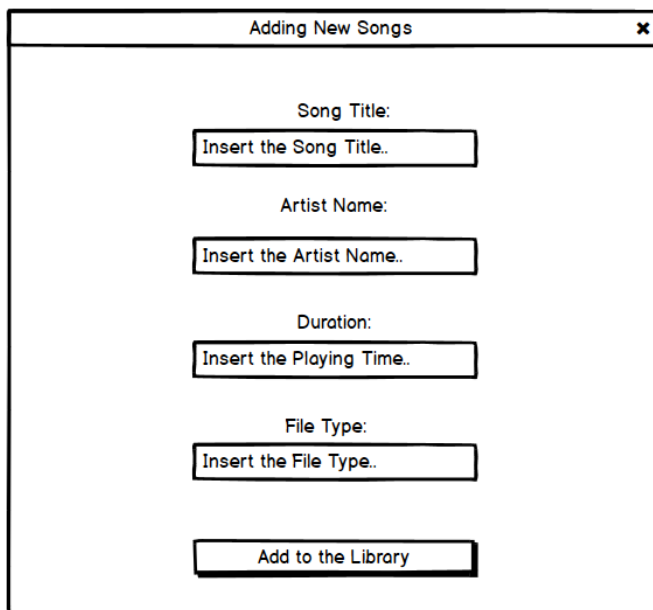
MAIN WINDOW



This is the main application window. It holds the following components:

- **Library table:** - Where the Song Title, Artist name, Duration and File type of all the songs in the library are displayed.
- **Search Bar:** - A text field allowing the user to input text for searching.
- **Search Button:** - Search song by song title from library.
- **Refresh Button:** - Reloads the Library table to display all songs.
- **Add to Library Button:** - Opens a window allowing user to add a new song to the Library.
- **Add to Playlist Button:** - Allows user to select a song from Library and add to playlist table.
- **Delete from Playlist:** -Allows User to remove a song from playlist table.
- **Play Media Player:** - Opens a window containing video player.
- **Playlist:** - A List consisting of songs chosen by the user from the song Library.

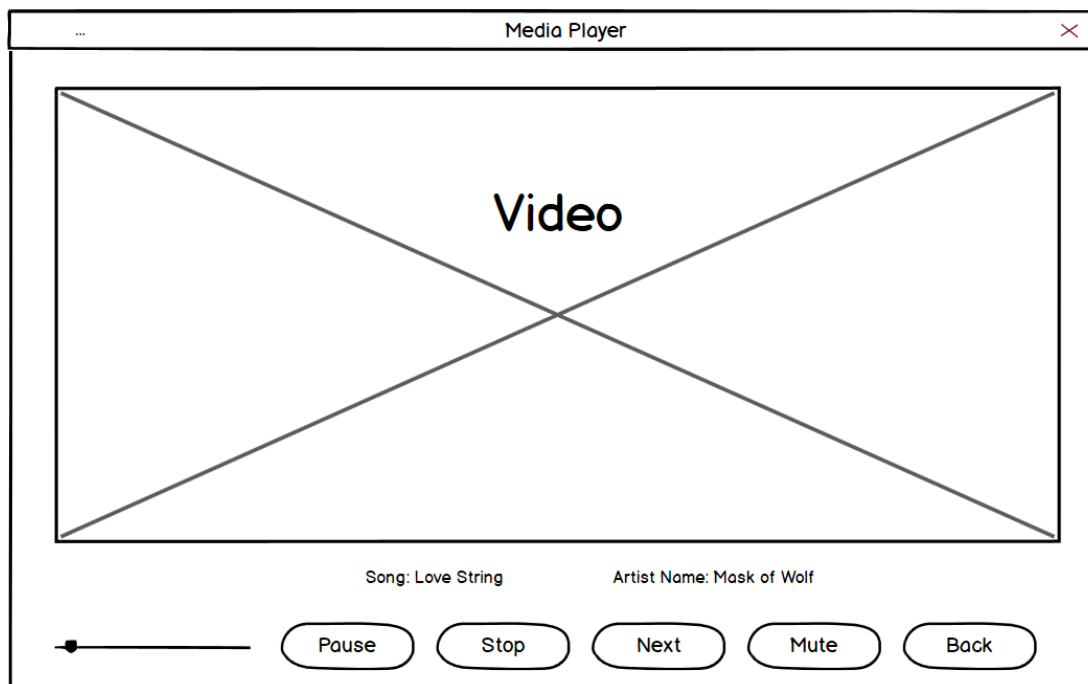
ADDING NEW SONGS WINDOW



A window titled "Adding New Songs" with a close button (X) in the top right corner. It contains five input fields and one button, all arranged vertically. The input fields are labeled "Song Title:", "Artist Name:", "Duration:", and "File Type:", each followed by a text box containing the placeholder text "Insert the Song Title..", "Insert the Artist Name..", "Insert the Playing Time..", and "Insert the File Type.." respectively. At the bottom is a button labeled "Add to the Library".

This window allows the user to add new songs to the Library. It consists of the following 4 compulsory fields that the user has to fill, namely: - Song Title, Artist Name, Duration and File type.

MEDIA PLAYER WINDOW



The Media Player window displays a video player and gives the user the following controls on the video:

- **Track Slider:** - Allowing the user to Fast- forward/Rewind the video according to a specific time
- **Pause Button:** -Pause/Play the video.
- **Stop Button:** - Stops the video.
- **Next Button:** - Plays the next video.
- **Mute:** - Mutes/ unmutes the video.
- **Back:** - Goes back to the Main Window.

Pseudo Code

ADD SONG TO LIBRARY

```
if (songtitlebar.getText() == null || songtitlebar.getText().trim().isEmpty()) { A1
    JOptionPane.showMessageDialog(null, "Please insert a song title ");
}
else if (!songtitlebar.getText().matches("[a-zA-Z0-9 ]+$")) { A2
    JOptionPane.showMessageDialog(null, "Please use only alphanumeric characters in the Name field ");
}
else if (artistbar.getText() == null || artistbar.getText().trim().isEmpty()) { A3
    JOptionPane.showMessageDialog(null, "Please insert a Artist name ");
}
else if (!artistbar.getText().matches("[a-zA-Z0-9 ]+$")) { A4
    JOptionPane.showMessageDialog(null, "Please use only alphanumeric characters in the Artist field ");
}
else if (playingtimebar.getText() == null || playingtimebar.getText().trim().isEmpty()) { A5
    JOptionPane.showMessageDialog(null, "Please insert the Duration");
}
else if (!playingtimebar.getText().matches("[0-9]+$")) { A6
    JOptionPane.showMessageDialog(null, "Please using integers only ( 1-9) for the duration");
}
else if (videofilenamebar.getText() == null || videofilenamebar.getText().trim().isEmpty()) { A7
    JOptionPane.showMessageDialog(null, "Please insert the video file type");
}
else if (libraryMap.containsKey(songtitlebar.getText())) { A8
    JOptionPane.showMessageDialog(null, "This Song already exists in Library");
}
else { A9
    music secondobject =new music();

    secondobject.setSongtitle(songtitlebar.getText()); A10
    secondobject.setArtist(artistbar.getText()); A11
    secondobject.setPlayingtime(Integer.parseInt(playingtimebar.getText())); A12
    secondobject.setVideofilename(videofilenamebar.getText()); A13
    libraryMap.put(secondobject.getSongtitle(),secondobject); A14

    JOptionPane.showMessageDialog(null, "The song has been successfully added to the Library ");
}

});
```

Running time: - $(A1*1) + (A2*1) + (A3*1) + (A4*1) + (A5*1) + (A6*1) + (A7*1) + (A8*1) + (A9*1) + (A10*1) + (A11*1) + (A12*1) + (A13*1) + (A14*1) = 1$

Conclusion: Constant Running time

TESTING

For this project, the testing framework chosen is **Junit 4.12**. The latter is a type of Unit testing whereby individual units or components of the software are tested. The purpose is to verify that each unit of the application code works as expected. The following sections have been separately tested: Searching from Library, Refreshing Library Table, adding to Playlist and Setter methods.

Test Table

Test	Description	Expected Result
Search in Library	<p>-This test consists of 2 methods namely SearchTest and TestForSearching.</p> <p>-SearchTest In this method, data was read from a file using 'BufferedReader'. The elements were then inserted in an array called 'thearray'. They were then converted into an object and the object placed into a HashMap. The method returns the HashMap size.</p> <p>-TestForSearching Using an AssertEquals method, the actual HashMap size was compared to the expected HashMap size. If it matches, the test is passed.</p>	-The expected result for the test to be passed is the HashMap size returned as 4364(the number of lines in listofsong file)
Add to Library	<p>-The test consists of 2 methods: AddSongtoLibrary and TestForAddingSongsToLibrary.</p> <p>- AddSongtoLibrary An object was created and populated with the following 4 elements: songtitle, artist, playingtime and videofilename. The object was put into a HashMap, with song title as key. This method returns the HashMap.</p> <p>-TestForAddingSongsToLibrary Using a. containKey method, the data hardcoded was compared to the data found in the HashMap.</p>	-The test is passed if data if the .contain method returns true.
Refreshing Table	<p>The test consists of 2 methods: RefreshLibraryTest and TestForAddingSongsToLibrary</p> <p>- RefreshLibraryTest Elements from the HashMap were retrieved and added into an ObservableList. This method returns the ObservableList.</p> <p>- TestForAddingSongsToLibrary An assertnotnull method was used to check if observable List contains items.</p>	If the Observable List does not return null. The test is passed.

Refer to appendix 1.1

Evidence of Testing

```
[cst2550@localhost Project]$ javac --module-path $JAVAFX_HOME --add-modules javafx.controls,javafx.media *.java
Note: GUI.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[cst2550@localhost Project]$ java --module-path $JAVAFX_HOME --add-modules javafx.controls,javafx.media GUI
[cst2550@localhost Project]$ java --module-path $JAVAFX_HOME --add-modules javafx.controls org.junit.runner.JUnit4 MainTest
JUnit version 4.12
.....
Time: 0.082

OK (7 tests)
```

CONCLUSIONS

4 elements of songs were read from a file, they were converted into an object and stored into a HashMap. The Songs were then retrieved from the HashMap and stored as an ObservableList so that they can be displayed in a table. In addition, new songs were added to the Library by prompting an interface to the user allowing him to insert details about the new song. Once the appropriate details added, the new song was added into the HashMap and displayed. In order to search for a specific song, a search field was given to the user. The user needs manually type a song title and click on the search button. The program will iterate through the HashMap, find a corresponding song title and display all the details of that song to the user. Moreover, the user can get the display of the whole table at any moment by clicking on the refresh button. In doing so, the program will reload the whole Observable List again and display it to the user. Furthermore, the program allows the user to create his own playlist. A song has to be selected from the Table and the 'Add to playlist' button clicked. Once this is done, the program will retrieve the object from the HashMap and store it into a LinkedList. The data in the LinkedList is then converted again into an ObservableList and displayed to the User in the 'Playlist' table. The User can also remove a song from 'Playlist' by selecting it and clicking on the delete button. This will command the program to remove the selected song from the LinkedList. Finally, the User can use his 'Playlist' to play videos the clicking on the 'Play Media Player' button. This will open a window consisting of a video player and 5 buttons to control the video namely: Pause/Play, Next, Stop, Mute/Unmute and Back.

A HashMap has been chosen to store the data from the files as in Inserting and deleting will be $O(1)$ + Hashing & Indexing (**amortized**). Moreover, though it takes little processing for the hashing and indexing, the processing time does not change even if the HashMap gets very large which is the case as over several millions of songs will be added. Moreover, When the HashMap gets full, it will increase its size. And, when the number of filled buckets is much smaller than the size of the HashMap, it will then decrease its size. Both operations take a complexity of $O(N)$. That's why insertion and deletion take $O(1)$ amortized [Elgabry, 2016].

To Populate the Playlist (List of song chosen by user), a LinkedList has been preferred as the operation calls for frequently adding songs to the front of the collection. For this type of process, a LinkedList has a time complexity of $O(1)$. In this case it is faster than other data structures like an array for example which would need to move all its elements to accomplish this task. Also, it is easy to retrieve the 1st element from a LinkedList.

Limitations

The data structures chosen however do have some limitations, the HashMap for example requires more space in memory than other data structures such as arrays. In addition, it is not guaranteed to be able to retrieve elements in a specific order. Most importantly, without knowing the Song Title (HashMap key) it is not possible to search for a specific song.

Likewise, the LinkedList used also do have some limitations. For example, the time complexity for sorting through a LinkedList is $O(N\log N)$. Furthermore, if songs were to be searched from the playlist the time complexity would have been $O(N)$.

Arguments have been used in the program; this forces the user to use command line to change file. This can prove to be inconvenient to people who are not very computer literate and wants to use this application.

Improvements

A Red Black Tree could have been used instead of the HashMap. Unlike for HashMap which has a worst case of $O(N)$ and best case of $O(1)$, the worst case, best case and average case of a Red Black tree is constant.

A file chooser approach could have been used instead of the arguments approach.

REFERENCES

Data Structures — A Quick Comparison (Part 2). 2020. Available at: <https://medium.com/omarelgabrys-blog/data-structures-a-quick-comparison-6689d725b3b0> [Accessed: 16 April 2020].

LinkedList (Java Platform SE 7). 2020. Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html> [Accessed: 18 April 2020].

Data Structures Comparison. 2020. Available at:
http://jcsites.juniata.edu/faculty/rhodes/cs2java/assignments/program_10_04.htm [Accessed: 24 April 2020].

HashMap (Java Platform SE 8). 2020. Available at: <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html> [Accessed: 25 April 2020].

APPENDICES

List of classes:

GUI, MainTest, mediaplayer, music, Readingfile, Testing, thenewlibrary.

Appendix 1.1:

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import org.junit.*;

import java.util.HashMap;

import static org.junit.Assert.*;

public class MainTest{

    //Test For Searching from Library
    @Test
    public void TestForSearching(){
        HashMap<String,music> libraryMap = new HashMap<String, music>();
        int expectedResult = 4364;
        assertEquals(expectedResult,Testing.SearchTest(libraryMap)); //Checking if SearchTest method is returning library
size as 4364(which is Number of lines in listofsong file).
    }

    //Test for Refreshing Library Table
    @Test
    public void TestForRefreshingLibrary() {
        ObservableList<music> Songs = FXCollections.observableArrayList();
        assertNotNull(Songs); //checking if Observablelist 'songs' returns null after refreshing
    }

    //Test for Adding new songs in Library
    @Test
    public void TestForAddingSongsToLibrary() {
        HashMap<String, music> libraryMap = Testing.AddSongtoLibrary();

        boolean check;

        if (libraryMap.containsKey("hishaam")) {
            check = true;
        } else {
            check = false;
        }
        assertTrue(check); //checking if the Hashmap finds the inserted key
    }

    //Test to verify Setter for songtitle
    @Test
    public void testSetsongtitle() {
        music music =new music();
        music.setSongtitle("test");
        assertEquals("songtitle","test", music.getSongtitle() ); }
}
```