



CST2550 – COURSEWORK 2

M00685151



Contents

Abstract	2
Introduction.....	2
Design	3
Use case diagram.....	3
Pseudocode	4
Add new records – songs.....	4
Search song title.....	4
Add song to end of playlist.....	5
Play next song in (and remove from) play-list.....	5
Analysis of time complexity	7
Hash Table – Insert Operation	7
Insert Operation – Estimate Running Time	8
Hash Table – Search Operation.....	8
Search Operation – Estimate Running Time.....	9
Wireframe Diagrams – GUI Mock ups.....	10
Main Menu.....	10
Message Box	10
Media Player	11
Library	11
Add New Record	12
Playlist	12
Testing	13
JUnit.....	13
Functional testing.....	14
Conclusion	14
Limitations and critical reflections	15
Limitations.....	15
Critical reflections	15
Future approach	15
References.....	16
Appendices	17
Appendix A	17
Appendix B	17
Appendix C	17
Appendix D	18
Appendix E.....	18

Abstract

This project is about analysing data structure using java programming language in order to determine which data structure is more suitable to handle millions of data with ease. A step by step procedure was laid out once each of the coursework specifications were fully analysed. Subsequently, a lot of effort was placed on the designing and testing stage. The resulted into an easy to use, fully functional karaoke application. To conclude, this is a complete karaoke application running without bugs and errors.

Introduction

This paper will be based on the realisation of a karaoke application using one of the most popular programming languages, Java. This whole application has been designed using JavaFX, a Java graphic library, in order to provide an intuitive interface to end users. As its name would suggest, this karaoke application will allow song lovers to sing along a variety of karaoke songs with music or video lyrics. The imperative objective of this project was to produce a user-friendly karaoke application allowing end users to benefit media player functionalities as well as choose desired songs from a library. The application has been developed to be robust, attractive and performant.

In addition, this report consists of two crucial stages of the development phase of the application: the designing phase and the testing phase. The designing phase will give an insight of the pseudocodes used to represent an informal high-level description of the program algorithm, an analysis of the time complexity to support the use the current data structure in the application, mock-ups of the graphical user interface to outline the application functionalities as well as a use case diagram to represent the functional requirements The testing phase will then cover the testing approaches and test cases used for the application.

Design

Before coding and implementation, it is crucial that each of the user requirements have been understood clearly by the developer. An application would be inefficient and useless if it does not meet its specific requirements. Hence it is important to emphasise on the designing stage as it will assist to define and understand the whole system architecture. (Gomaa, 2011)

Use case diagram

A use case diagram is a unified modelling diagram that represent the functionality of a system by making use of associations, use cases and actors as shown below: (Bittner and Spence, 2008)

Use cases: Represent the core functionalities of the system (for ex: Play Media, View playlist songs).

Actors: Users that interacts with the system or triggers the use cases.

Associations: Represent the communication between actors to uses cases, or within uses cases.

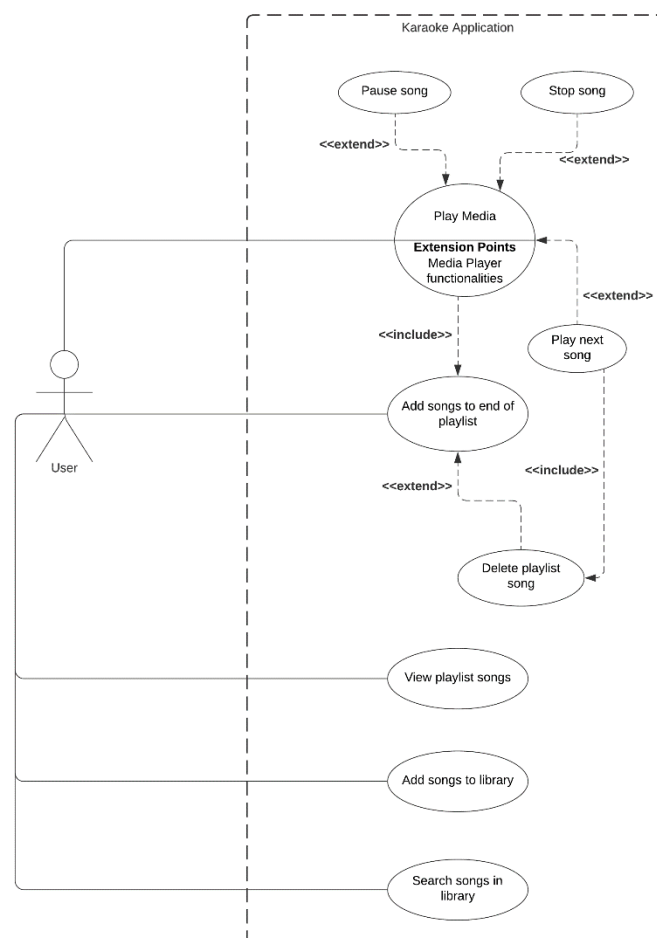


Figure 1: Use Case Diagram

Pseudocode

Whenever an algorithm to a problem has been designed, we use pseudocode to represent the algorithm. Pseudocode is a way of using identifiers and keywords to outline an algorithm without using any particular programming language syntax. (Langfield and Duddell, 2015)

Below pseudocodes will represent each of the functionalities implemented in the karaoke application.

Add new records – songs

```
AddNewRecordButton.onClick =>  
call PassValidation(title,artist,time,videoPath)  
  if PassValidation == true then  
    AddProcess(title,artist,time,videoPath)  
  endif
```

Call 'PassValidation' method to check for invalid data.

Proceed to add new record if 'PassValidation' method return true.

```
function AddProcess(title,artist,time,videoPath)  
  libraryhash.put(title,Song[title,artist,time,videoPath])  
endfunction
```

Add the new data in the library data structure. 'Put' function will be discussed in details in time complexity analysis.

Search song title

```
SearchButton.onClick =>  
  if searchField.isEmpty() then  
    display "Please enter a song title!"  
  else  
    call songsFound ← SearchSong(searchTitle,libraryHash)  
    if songsFound.isEmpty() then  
      display "No song found!"  
    endif  
  endif
```

Check for search field when user pressed on search button.

Display error message for empty field.

Call 'SearchSong' function which returns a list of songs found.

Inform user for zero songs found.

```
function SearchSong(searchTitle,libraryHash)  
  List songsFound <Song>  
  for each songTitle in libraryHash  
    if Lcase[songTitle].contains(searchTitle) then  
      songsFound.add(libraryHash.song)  
    endif  
  endfor  
  return songsFound  
endfunction
```

Loop through library song data structure to find songs.

Check if lower case of each song's title contains the search title.

Insert into song list if true.

Return the song list.

Add song to end of playlist

```
AddToPlaylistButton.onClick =>
  NewSongIndex ← Table.selectedItem[index]
  if NewSongIndex <> -1 then
    NewPlaylistSong ← [Song] NewSongIndex
    if playlist.contains(NewPlaylistSong) then
      display "Song is already present in playlist!"
      return
    endif
    playlist.add(NewPlaylistSong)
    display " Song successfully added!"
  else
    display "Please select a song!"
  endif
```

Retrieve selected song index from library table.

Check if index is valid (Not equal to -1).

Inform user if did not select a song (index = -1)

Convert song index to a song object.

Check if selected song is already present in playlist.

If yes, inform user for duplicate song else add selected song to end of playlist.

Play next song in (and remove from) play-list

```
MediaPlayer.onLoad =>
  MediaPlayer.play()
  NextSongButton.onClick =>
    MediaPlayer.stop()
    call PlayNextAudio(playlist)
```

Stop current song if pressed next song button.

Call 'PlayNextAudio' function to play next song.

```
function PlayNextAudio(playlist)
  playlistDataStructure.remove(getFirstSong(playlist))
  if playlistDataStructure.isEmpty() then
    display " Playlist is empty!"
    return
  endif
  NextSong ← getFirstSong(playlist)
  AudioPath ← getVideoFilePath(NextSong)
  MediaPlayer.setNextAudio(AudioPath)
  MediaPlayer.play()
endfunction
```

Call 'getFirstSong' function to retrieve current song on playlist (will be first song in playlist by default) and delete from playlist.

Check if playlist is empty after the deletion and inform user if same.

Get the next song in playlist and set to media player.

```
function getFirstSong(playlist)
  if playlist.hasNext() then
    return playlist.next()
  else
    return null
  endif
endfunction
```

Check if play list is not empty.

Return the song on top of the playlist.

Retrieve songs in play-list (to view)

Playlist.onLoad =>

try:

PlaylistColumn.setText("SongTitle")

call DisplayPlaylistSong(playlist)

PlaylistTable.setColumns(PlaylistColumn)

catch Exception

display "Error in loading table"

endtrycatch

Set playlist column title to "Song Title"

Call 'DisplayPlaylistSong' method to load song data into playlist table.

function DisplayPlaylistSong(playlist)

List playlistSong<Song>

for each Song **in** playlist

playlistSong.add(Song)

endfor

PlaylistTable.setItems(playlistSong)

endfunction

Create a list and loop through playlist song.

Insert each playlist song in the newly created list.

Set list to table items and display them in the playlist table.

Delete song from play-list

DeletePlaylistSongButton.onClick =>

SongIndex ← Table.selectedItem[index]

if SongIndex <> -1 **then**

SelectedSong ← [Song] SongIndex

DeletePlaylistSong(SelectedSong,playlist)

else

if playlist.isEmpty() **then**

display "Playlist is already empty"

endif

display "Please select a song!"

endif

Check if user selected a song from playlist table by comparing index.

If index is -1, inform user to select a song.

If index is not -1, meaning user has selected a song, call 'DeletePlaylistSong'.

function DeletePlaylistSong(selectedSong,playlist)

playlist.**remove**(selectedSong)

endfunction

Delete the selected song from playlist.

Analysis of time complexity

Throughout this entire coursework, a hash symbol table has been chosen to represent the karaoke application library as it supports dictionary like operation such as “PUT” and “GET”. Assuming it’s uniform hashing, these dictionary operations will require only $O(1)$ time on average. Although it is expected that the number of songs will grow in millions in the future, the running time for both searching and insertion will still remain constant.

Linked Hash Set data structure has been implemented for the karaoke playlist. Since a First In First Out (FIFO) order is maintained in a linked hash set, this data structure is ideal to represent a playlist. A performance of order $O(1)$ is also maintained for insert, delete and retrieve operations. (LinkedListSymbolTable (Java Platform SE 7), 2020)

Hash Table – Insert Operation

Below gives an insight about a Separate Chaining type pseudocode for inserting key K into the hash symbol table with capacity “NumOfChains”. We have assumed that uniform hashing takes place where we will store a set a new data(key).

```
function put(Key,Value)
if NumOfPairs >= 10* NumOfChains then
    resize( 2*NumOfChains)
endif
HashIndex ← hash(Key)
NumOfPairs = NumOfPairs + 1
LinkedListSymbolTable[HashIndex] ←
    Constructor Node(Key,Value,LinkedListSymbolTable[HashIndex])
endfunction
```

Resize hash if reach load factor threshold.

Generate a hash index for the new key and add into the hash symbol table.(Appendix A)

```
function resize(Chains)
HashST temp ← HashST<Key,Value>
for i ← 0 to NumOfChains -1 do
    for Node x ← LinkedListSymbolTable[i] to x <> null do
        temp.put(x.key,x.value)
        x ← x.next
    endfor
endfor
NumOfChains ← temp.NumOfChains
NumOfPairs ← temp.NumOfPairs
LinkedListSymbolTable ← temp.LinkedListSymbolTable
NumOfChains ← NumOfChains +1
endfunction
```

Resize the hash table.

Create a temporary hash and load all data from original hash table.

Replace old hash to new hash.

(Appendix B)

```
function hash(Key)
i ← key.hashCode() & 0x7fffffff % numOfChains
return i
endfunction
```

Generate a hash key index for the new data. (Appendix C)

The average insertion time will be therefore be $O(1)$ for both best case and worst case.

Insert Operation – Estimate Running Time

A stopwatch class has been implemented to determine the actual running time for insertion algorithm as shown below:

Data size	Time(seconds)	Ratio(r)
500	0.042	-
1000	0.071	1.69
2000	0.109	1.54
4000	0.148	1.36

Table 1: Insert Running Time

Expected time was supposed to be $(0.109 * 1.54) = 0.167$ s

But the actual time resulted to 0.148 s

Hash Table – Search Operation

Below gives an insight about a Separate Chaining type pseudocode for searching key K into the hash symbol table with capacity “NumOfChains” with backing array “linkedListSymbolTable”.

```
function get(Key)
i ← hash(Key)
for Node x ← linkedlistSymbolTable[i] to x <> null do
    if Key.equals(x.key) then
        return x.value
    endif
    x ← x.next
endfor
return null
```

Generate the hash index for the data key to be searched.

Loop in the backing array until the corresponding object value has been found.

Return null for any unsuccessful search.

(Appendix D)

Assuming that the hash function to generate the hash index is computed in $O(1)$ time, then the total time that is required for searching an element of key K will be directly proportional to the chain length which is basically N/M (also known as the load factor) when there are N keys with M chains in a table.

Load factor $\alpha = N/M$.

Average case for searching

Expected running time for searching will be $O(1 + \alpha)$ where 1 for applying the hash function and α to search in the linked list symbol table.

Best case for searching assuming uniform hashing

If data is found at start, α will be $O(1)$ resulting $O(1 + \alpha)$ to be $O(1) \rightarrow$ i.e $M = \Omega(N)$.

Worst case for searching

$\Theta(n)$: This will happen if an improper hashing has been implemented resulting all N key hash to be present in the same slot, thus creating a linked list of length N .

Unsuccessful search

Ideal average length of a list is α . Searching for a key that is not present in the table will require searching throughout the whole list in order to determine that the key is not present in the table. Without neglecting the cost of computing hashing method "hash (Key)", the total work will then be $\Theta(1 + \alpha)$.

Successful search

Assuming that all types of lists are of defined length $\alpha = N/M$ (and where N is divided by M evenly in order for $\alpha = N/M$ to be an integer type)

The number of elements to search for, will be $1 +$ number of elements in list entered before key K .

So the average comparisons is $((1 + N/M) / 2) \rightarrow \Theta(1/2 + \alpha/2)$

Yet when we include the cost of computing, the hashing method gives $\Theta(3/2 + \alpha/2) \rightarrow \Theta(1 + \alpha)$ which produces the same result to unsuccessful searching.

Search Operation – Estimate Running Time

Data size	Time(seconds)	Ratio
500	0.0017	-
1000	0.0028	1.64
2000	0.0033	1.18
4000	0.0036	1.1

Expected time was supposed to be $(0.0033 * 1.18) = 0.0039$ s

But the actual time resulted to 0.0036 s

Table 2: Search Running Time

General analysis

If the number of slots in the hash table becomes at least proportional to the number of elements in the table, we shall have $N = O(M)$, therefore, $\alpha = N/M \rightarrow O(M)/M \rightarrow O(1)$. Hence it will take a constant time on average for data searching.

One of the major advantages of separate chaining, is that it is not sensitive to the table size. That is, if the number of keys has been more than anticipated, the cost of searching will be worse and if the number has been less, this will lead to only a small amount of space to be wasted and searching cost will be faster.

Wireframe Diagrams – GUI Mock ups

Graphical user interface mock-ups are artifacts that provide a visual understanding of what a future software of application will look like. They assist in demonstrating what interface elements will exist on the system. (Young, 2019)

A mock-up is of no use if it does not provide a good visual understanding of the future application. This is why, when designing mock-ups, we should take note to produce high quality mock-ups where each of them should respect the “Nielsen Heuristics” principles as these principles, they identify any design issues and ensure usability for any interface design. (Nielsen, 1994)

Main Menu



The karaoke menu has been designed with Nielsen’s principle “**Match between the system and the real word**” which states that systems should speak user’s language with familiar concepts and words. In this respect, matching icons and plain language has been used on each button to indicate it’s functionality. (Nielsen, 1994)

Figure 2: Main Menu GUI

Message Box



Error messages have been expressed in simple words so as to indicate the problem clearly to the user, taking the principle “**Help users recognize, diagnose, and recover from errors**” into consideration. (Nielsen, 1994)

Figure 3: Message Box GUI

Media Player

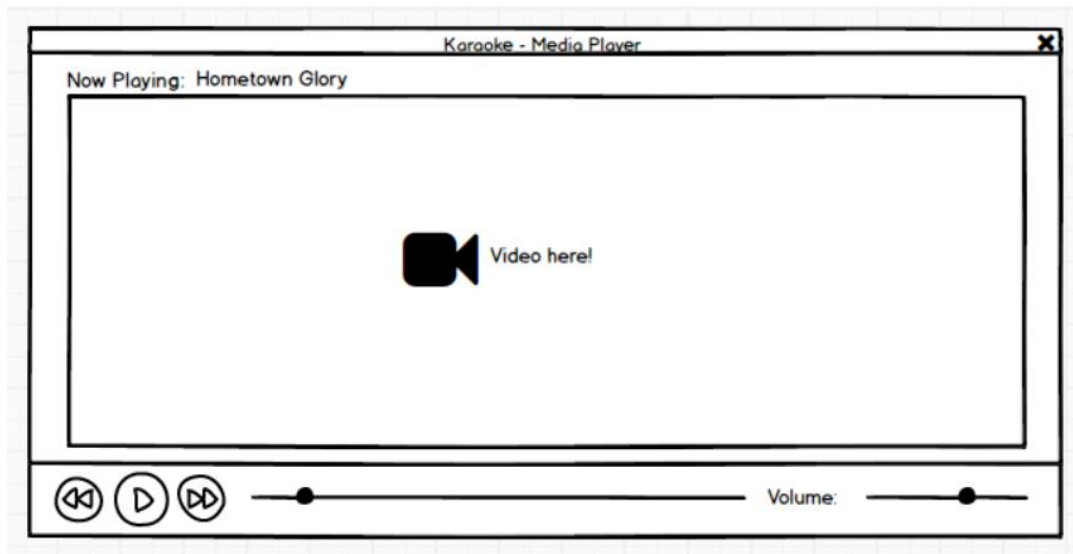


Figure 4: Media Player GUI

Principle “**Visibility of system status**” has been maintained when designing the media player interface since both the song label and the slider keep the user informed about what is going on as stated in the principle. (Nielsen, 1994)

Media player buttons standards (Principle “**Consistency and standards**”) have been also maintained to refrain users from wondering what these buttons are meant for.

Library

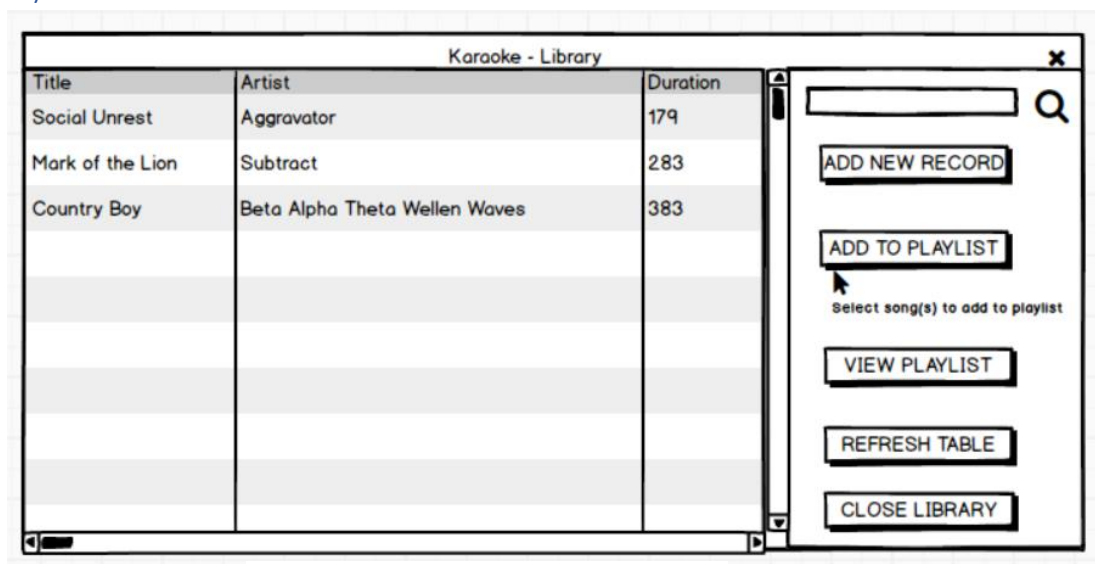
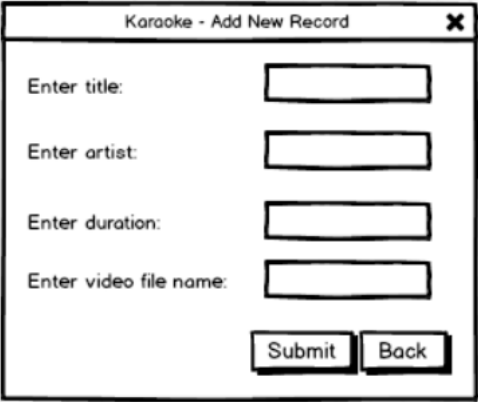


Figure 5: Library GUI

Search icon has been implemented to adopt principle “**Consistency and standards**”, for searching songs. The search bar has also been placed in the upper right-hand area to improve perceivability.

Add New Record

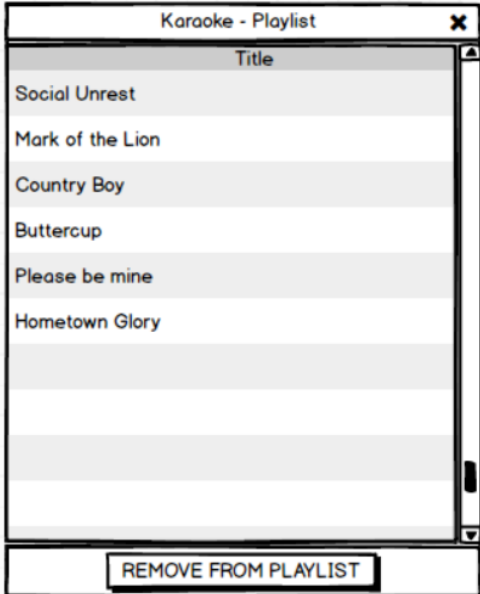


The image shows a window titled "Karaoke - Add New Record" with a close button (X) in the top right corner. Inside the window, there are four text input fields, each preceded by a label: "Enter title:", "Enter artist:", "Enter duration:", and "Enter video file name:". Below these fields are two buttons: "Submit" and "Back".

This window adopts the principle “**User control and freedom**” which states that user should possess an “emergency exit” to leave any unwanted state, as the back button will exit this entire window. (Nielsen, 1994)

Figure 6: Add new Record GUI

Playlist



The image shows a window titled "Karaoke - Playlist" with a close button (X) in the top right corner. Inside the window, there is a list box with a header "Title". The list contains the following items: "Social Unrest", "Mark of the Lion", "Country Boy", "Buttercup", "Please be mine", and "Hometown Glory". Below the list box is a button labeled "REMOVE FROM PLAYLIST".

Playlist window to display all songs in the user playlist. The user also has the ability to delete any song from the playlist.

Figure7: Playlist GUI

Testing

Testing was, is and will always remain one of the most important phases in the development life cycle of an application or software. The karaoke application would not reach deployment state if a proper set of tests were not conducted. These tests carried out; they assist in certifying that the entire application is operating according its requirements.

JUnit

In this respect, JUnit testing has been chosen as one of the testing approaches in the realisation of the karaoke application as it is an accurate and efficient testing process. JUnit, as its name applies, is a unit testing framework for java where it allows developers to create several small tests as units, and test whether they behave as expected. (JUnit (Java Platform SE 7). 2020)

In order to execute JUnit test framework, a set of test cases were implemented. These test cases, they contain sequence of actions, test data, preconditions and postconditions to conclude whether the karaoke application meets it's requirement by comparing expected test value to actual result value.

JUnit Test Case	Test Case Description	Expected result	Actual result	Pass/Fail
@testLoadFileData	Test if valid data can be read or loaded from a file to the implemented data structure.	Data to be successfully loaded.	Data has been loaded successfully.	Pass
@testAddProcess	Test if additional data can be added into the implemented data structure.	Additional data to be added into the data structure.	Additional data inserted successfully.	Pass
@testSearchSong	Search for a song that does not exist in the data structure.	No song found.	Fails to find song.	Pass
@testGetSongArtist	Test to check if a song objects have null values when initialising.	Null value for song artist.	Return null value for song artist.	Pass
@testPerformance	Performance test to check data structure efficiency.	Populate data within said time.	Data successfully populated.	Pass
@testGetFirstSong	Test to return first song in playlist data structure.	Returns only first song.	Return the first song.	Pass
@testLoadTableDataItems	Test to determine if return all library song.	Return library songs.	Return library songs.	Pass
@testCheckPlaylistSongDuplication	Checks for duplicated song in playlist data structure.	Return false as no duplicated song in playlist.	Return false.	Pass
@testDeletePlaylistSong	Test to see if deletion in playlist works.	Song to be deleted.	Song has been deleted.	Pass

Table 2: Testing result table

Functional testing

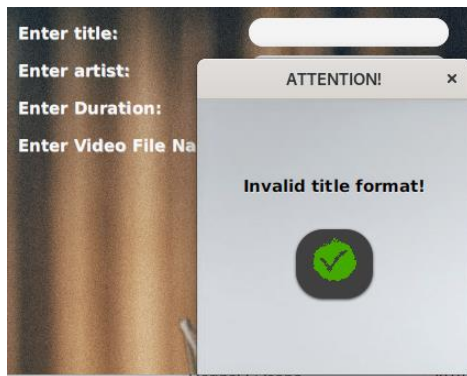
Functional testing has been implemented as the second testing approach for the karaoke application. It is a black box type testing, where we put focus only on the inputs and outputs without the need to have an internal knowledge of the testing application.

Functional testing can be used to test many functionalities but throughout this application, it was used to check for error conditions and to display appropriate error messages when adding a new record to the karaoke song library.

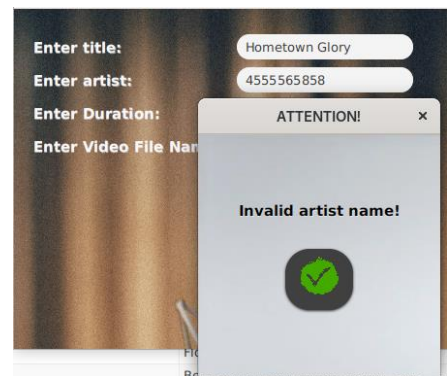
A decision table has been therefore applied to support the functional testing.

Conditions	Test 1	Test 2	Test 3	Test 4
Song title	Invalid data	Valid data	Valid data	Valid data
Song artist	Valid data	Invalid data	Valid data	Valid data
Song duration	Valid data	Valid data	Invalid data	Valid data
Vide path	Valid data	Valid data	Valid data	Invalid data
Expected Output	Invalid title format!	Invalid artist name!	Invalid playing time!	Invalid video file format!
Evidence	E1	E2	E3	E4

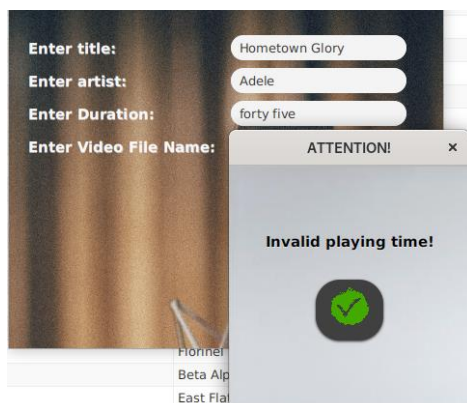
Table 3: Decision table



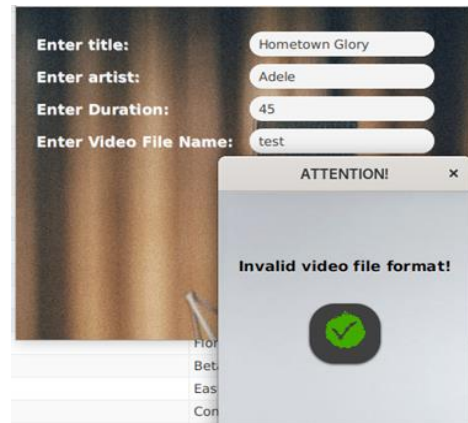
E1



E2



E3



E4

Conclusion

The main objective of the project was to build a karaoke application to assist users to get the best experience while singing. In this respect, all core features of the karaoke application were targeted and each one of them has been successfully implemented. The application was also constantly improved throughout its implementation, catering for unwanted constraints which popped up. In addition, an analysis of time complexity had to be carried out to determine which data structures were more suitable for both the karaoke library and playlist. The graphical user interface of the application designed such that the majority adhered to the 'Nielsen' principles. Finally, a set of unit testing was conducted to identify any defects present in the karaoke application so that it performs as expected once deployed.

Limitations and critical reflections

Limitations

The karaoke application has not adhered by 'Nielsen' principles for all of its interfaces. It has been assumed that no two songs can have the same title, if such happens by mistake, the data structure has not been designed to cater for song duplication.

Moreover, the interfaces of the application are also not appealing enough to motivate end users to use it and end users have only a limited number of choices to use, no additional functionality has been implemented to facilitate their usage.

Critical reflections

Although I was able to finish this coursework within the timeframe, I took a significant amount of time in understanding and applying time complexity of the data structure. This helped to realise where I need to focus more on future similar projects.

This karaoke application does consist some limitations but these does not affect its functionality at all. Hence this application can be considered as a success.

Future approach

More emphasis will be placed on planning and understanding each requirements of the future project. Each interface will be designed with 'Nielsen' principles make sure it is appealing to any end users. A database will be implemented to store user accounts and save any incomplete tasks (playlist song in this case). Subsequently, a more in-depth analysis will be performed to better support, the choice for data structure implemented.

Beta testers will also be included to limit any future abnormalities.

References

- Bittner, K. and Spence, I., 2008. *Use Case Modeling*. 5th ed. Boston, Mass.: Addison-Wesley, p.28.
- Docs.oracle.com. 2020. MediaPlayer (Javafx 2.2). [online] Available at: <https://docs.oracle.com/javafx/2/api/javafx/scene/media/MediaPlayer.html> [Accessed 3 March 2020].
- Gomaa, H., 2011. *Software Modeling And Design*. 1st ed. Cambridge: Cambridge University Press, p.40.
- Hashtable (Java Platform SE 7). 2020. Available at: <https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html> (Accessed: 2 May 2020).
- JUnit (Java Platform SE 7). 2020. Available at: <https://docs.oracle.com/javame/test-tools/javatest-441/html/junit.html> (Accessed: 3 May 2020).
- Langfield, S. and Duddell, D., 2015. Cambridge International AS And A Level. 1st ed. Cambridge: Cambridge, p.128.
- LinkedHashSet (Java Platform SE 7). 2020. Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html> (Accessed: 2 May 2020).
- LinkedList (Java Platform SE 7). 2020. Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html> (Accessed: 2 May 2020).
- Nielsen, J., 1994. *10 Heuristics For User Interface Design: Article By Jakob Nielsen*. [online] Nielsen Norman Group. Available at: www.nngroup.com/articles/ten-usability-heuristics [Accessed 29 April 2020].
- Sedgewick, R. and Wayne, K. 2011. *Algorithms*. Boston, MA: Pearson Education.
- Sedgewick, R., 2020. *Analysis Of Algorithms*. [online] Aofa.cs.princeton.edu. Available at: <https://aofa.cs.princeton.edu/10analysis> [Accessed 3 May 2020].
- Young, N., 2019. *What Is Wireframing?*. [online] Experience UX. Available at: www.experienceux.co.uk/faqs/what-is-wireframing [Accessed 22 April 2020].

Appendices

Appendix A

```
public void put(Key key, Value val) {  
    // double table size if average length of list >= 10  
    if (numOfPairs >= 10*numOfChains) resize(2*numOfChains);  
    int hashIndex = hash(key);  
    numOfPairs++;  
    linkedListSymbolTable[hashIndex] = new Node(key, val, linkedListSymbolTable[hashIndex]);  
}
```

Appendix B

```
private void resize(int chains) {  
    KaraokeST<Key, Value> temp = new KaraokeST<Key, Value>(chains);  
    for (int i = 0; i < numOfChains; i++) {  
        for (Node x = linkedListSymbolTable[i]; x != null; x = x.next) {  
            temp.put((Key) x.key, (Value) x.val);  
        }  
    }  
    this.numOfChains = temp.numOfChains;  
    this.numOfPairs = temp.numOfPairs;  
    this.linkedListSymbolTable = temp.linkedListSymbolTable;  
}
```

Appendix C

```
private int hash(Key key) {  
    return (key.hashCode() & 0x7fffffff) % numOfChains;  
}
```

Appendix D

```
public Value get(Key searchKey) {  
    int i = hash(searchKey);  
    for (Node x = linkedListSymbolTable[i]; x != null; x = x.next) {  
        if (searchKey.equals(x.key)) return (Value) x.val;  
    }  
    return null;  
}
```

Appendix E

@Test

```
public void testLoadFileData(){  
    KaraokeST<String, Song> libraryHash = new KaraokeST<String,Song>();  
    String filePath="sample_song_data.txt";  
    FileManagement file = new FileManagement();  
    file.loadFileData(filePath,libraryHash);  
    assertNotNull("Failed to load data",libraryHash);  
}
```

@Test

```
public void testAddProcess(){  
    KaraokeST<String, Song> libraryHash = new KaraokeST<String,Song>();  
    AddNewRecords.addProcess("MySong", "Adele",125,"test.mp4",libraryHash);  
    assertTrue("Failure to add new data",libraryHash.contains("MySong"));  
}
```

@Test

```
public void testSearchSong(){  
    KaraokeST<String, Song> libraryHash = new KaraokeST<String,Song>();  
    libraryHash.put("Fire in the rain",new Song("Fire in the rain", "Adele",325,"test.mp4"));  
    assertEquals(0,(Library.searchSong(libraryHash,"Hello").size()));  
}
```

@Test

```
public void testGetSongArtist(){  
    Song newSong = new Song();  
    assertNull("Object should be null",newSong.getSongArtist());  
}
```

@Test(timeout = 180)

```
public void testPerformance(){  
    KaraokeST<String, Song> libraryHash = new KaraokeST<String,Song>();  
    String filePath="sample_song_data.txt";  
    FileManagement file = new FileManagement();  
    file.loadFileData(filePath,libraryHash);  
}
```

@Test

```
public void testGetFirstSong(){  
    Song firstSong = new Song("Fire in the rain","Adele",200,"test.mp4");  
    Song otherSong = new Song("Hello","Adele",400,"test.mp4");  
    LinkedHashSet<Song> playlist = new LinkedHashSet<Song>();  
    playlist.add(firstSong);  
    playlist.add(otherSong);  
    assertNotSame("Same song objects",otherSong,Player.getFirstSong(playlist));  
}
```

@Test

```
public void testLoadTableDataItems(){  
    KaraokeST<String, Song> libraryHash = new KaraokeST<String,Song>();  
    Song firstTestSong = new Song("Fire in the rain","Adele",100,"test.mp4");  
    Song secondTestSong = new Song("Skyfall","Adele",200,"test.mp4");  
    Song thirdTestSong = new Song("Rumour Has It","Adele",300,"test.mp4");  
    libraryHash.put(firstTestSong.getSongTitle(),firstTestSong);  
    libraryHash.put(secondTestSong.getSongTitle(),secondTestSong);  
    libraryHash.put(thirdTestSong.getSongTitle(),thirdTestSong);  
    ObservableList<Song> libraryTableSong = Library.loadTableDataItems(libraryHash);  
    assertTrue("Error in loading table items",(libraryTableSong.contains(secondTestSong) && libraryTableSong.size() ==  
3));}
```

@Test

```
public void testCheckPlaylistSongDuplication(){  
    LinkedHashSet<Song> playlist = new LinkedHashSet<Song>();  
    Song firstSong = new Song("Fire in the rain", "Adele", 200, "test.mp4");  
    Song otherSong = new Song("Hello", "Adele", 200, "test.mp4");  
    playlist.add(firstSong);  
    assertFalse("Song already in playlist!", Library.checkPlaylistSongDuplication(playlist, otherSong));  
}
```

@Test

```
public void testDeletePlaylistSong(){  
    LinkedHashSet<Song> playlist = new LinkedHashSet<Song>();  
    Song selectedSong = new Song("Fire in the rain", "Adele", 200, "test.mp4");  
    playlist.add(selectedSong);  
    Playlist.deletePlaylistSong(selectedSong, playlist);  
    assertFalse(playlist.contains(selectedSong));  
}
```