



UNIVERSITY OF GREENWICH

Alliance with  Education

Student name:	Nguyen Anh Tu
ID number:	GCS230421
Academic year:	2024-2025
Module:	COMP- 1752 Object-Oriented Programming
Module Assessment Title:	Jukebox Simulation
Submission Date:	01/12/2024
Due Date	04/12/2024

I. Introduction.....	3
II. COURSEWORK IMPLEMENTATION	
Basic Understandings	4
Design and Development	10
TESTING AND VALIDATION.....	16
III. Conclusion.....	18

Jukebox Simulation

I. Introduction

The Jukebox Simulation is an incomplete project in python programming, focusing on designing GUI (graphic user interface), developing by using Tkinter, PIL, etc. The core objective is to redesign, functioning missing details such as create playlist and update track rating.

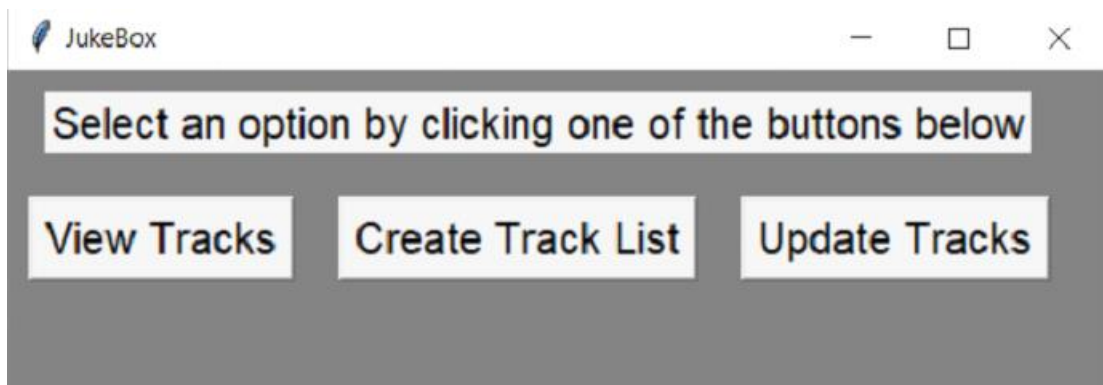
To achieve this scenario , the project needs to involved 5 main stages (basic understanding, outlining implementation, basic working version, testing and validation and innovation developments). Inconclusion , this project requires a solid understanding of Python programming concepts, object-oriented design principles, and GUI development using Tkinter. It also emphasizes the importance of effective testing and debugging to ensure the reliability and robustness of the application.

II. Coursework implantation

1. Basic understandings:

*The course work provided students with template code in Pycharm project called Jukebox(containing track_player.py , views_track.py and font manager.py), in this section, we are going to explain the template code.

-Track_player.py



```
import tkinter as tk # import tkinter library

import font_manager as fonts #import font style
from view_tracks import TrackViewer # import track viewer class on view_track

def view_tracks_clicked(): # this is a function
    status_lbl.configure(text="View Tracks button was clicked!")
    TrackViewer(tk.Toplevel(window))
```

The “view_tracks_clicked” is a function when the user clicks it, it will pop up the “view_tracks” and displaying it in a separated window

```

window = tk.Tk() #Creates the main application window using tk.Tk()
window.geometry("520x150") #window size
window.title("JukeBox") # title of the window set to "Jukebox"
window.configure(bg="gray") # background color

fonts.configure() #adding font style from font_mangager.py

header_lbl = tk.Label(window, text="Select an option by clicking one of the buttons below") #create label using tk.Label
header_lbl.grid(row=0, column=0, columnspan=3, padx=10, pady=10)# location to set the label

view_tracks_btn = tk.Button(window, text="View Tracks", command=view_tracks_clicked) #create button called view tracks
view_tracks_btn.grid(row=1, column=0, padx=10, pady=10)

create_track_list_btn = tk.Button(window, text="Create Track List")#create button called create tracks lists
create_track_list_btn.grid(row=1, column=1, padx=10, pady=10)

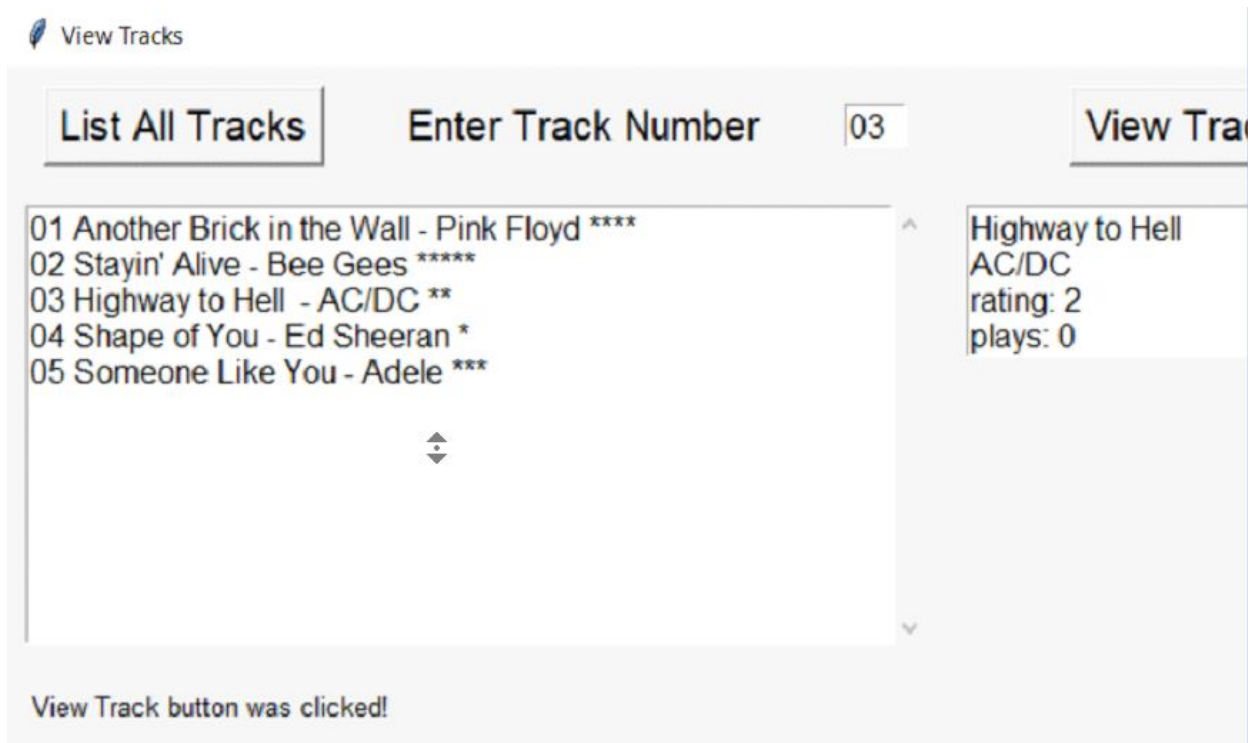
update_tracks_btn = tk.Button(window, text="Update Tracks")#create button called update tracks
update_tracks_btn.grid(row=1, column=2, padx=10, pady=10)

status_lbl = tk.Label(window, bg='gray', text="", font=("Helvetica", 10)) # adding another label
status_lbl.grid(row=2, column=0, columnspan=3, padx=10, pady=10)

window.mainloop()

```

-View_track.py



* The View_track.py functioning by using user's input to list the track's key, from then on, it will play the track the user wants to hear

```
import tkinter as tk # import module
import tkinter.scrolledtext as tkst # import module

import track_library as lib # import library module dictionary
import font_manager as fonts # import font styling for font in this file

def set_text(text_area, content):
    #deletes all existing content using
    text_area.delete("1.0", tk.END)
    #inserts the new content
    text_area.insert(1.0, content)

class TrackViewer(): # setting TrackerViewer window
    def __init__(self, window):
        window.geometry("750x350") #size of the window
        window.title("View Tracks") #setting the title of window to View Tracks
        #create button from "tk.Button" then set the label if the button to "List All Tracks"
        list_tracks_btn = tk.Button(window, text="List All Tracks", command=self.list_tracks_clicked)
        list_tracks_btn.grid(row=0, column=0, padx=10, pady=10)
        #create label widget in window then set the text to display on label
        enter_lbl = tk.Label(window, text="Enter Track Number")
        enter_lbl.grid(row=0, column=1, padx=10, pady=10)
        # create an entry field
        self.input_txt = tk.Entry(window, width=3)
        self.input_txt.grid(row=0, column=2, padx=10, pady=10)
```

```

#creat View Track button
check_track_btn = tk.Button(window, text="View Track", command=self.view_tracks_clicked)
check_track_btn.grid(row=0, column=3, padx=10, pady=10)
#Imports the ScrolledText class from Tkinter for displaying multi-line text with a scrollbar.
self.list_txt = tkst.ScrolledText(window, width=48, height=12, wrap="none")
self.list_txt.grid(row=1, column=0, columnspan=3, sticky="W", padx=10, pady=10)
# create text widget
self.track_txt = tk.Text(window, width=24, height=4, wrap="none")
self.track_txt.grid(row=1, column=3, sticky="NW", padx=10, pady=10)
#set window label font
self.status_lbl = tk.Label(window, text="", font=("Helvetica", 10))
self.status_lbl.grid(row=2, column=0, columnspan=4, sticky="W", padx=10, pady=10)

self.list_tracks_clicked()# list all track

def view_tracks_clicked(self):# funtion for View Track button
    key = self.input_txt.get() #get user input then find key by lib ffrom Track_library
    name = lib.get_name(key)
    if name is not None:
        artist = lib.get_artist(key)# get artist name by key
        rating = lib.get_rating(key)#get rating by key
        play_count = lib.get_play_count(key) #get play count by key
        track_details = f"{name}\n{artist}\nrating: {rating}\nplays: {play_count}" #formats the track details into a string.
        #Calls the set_text function to update the self.track_txt text widget with the formatted track details.
        set_text(self.track_txt, track_details)
    else:
        set_text(self.track_txt, f"Track {key} not found")# if inout user is not found, put will be"Track {user in put} not found"
    self.status_lbl.configure(text="View Track button was clicked!")

def list_tracks_clicked(self):#Fetches the entire list of tracks from the library.
    track_list = lib.list_all()
    set_text(self.list_txt, track_list)
    self.status_lbl.configure(text="List Tracks button was clicked!")

```

```

if __name__ == "__main__": # only runs when this file is run as a standalone
    window = tk.Tk()        # create a TK object
    fonts.configure()        # configure the fonts
    TrackViewer(window)      # open the TrackViewer GUI
    window.mainloop()        # run the window main loop, reacting to button presses, etc

```

-Library_item:

```

class LibraryItem: # make a Library class creating objects
    def __init__(self, name, artist, rating=0):
        self.name = name#The name of the track (string).
        self.artist = artist# The artist or band of the track (string).
        self.rating = rating#the rating of the track (integer)
        self.play_count = 0#This attribute likely keeps track of how many times the track has been played (not provided as an argument)
    #returns a formatted string representing the track information
    def info(self):
        return f"{self.name} - {self.artist} {self.stars()}"
    # generates a visual representation of the track's rating using asterisks ("*")
    def stars(self):
        stars = ""
        for i in range(self.rating):
            stars += "*"
        return stars

```

-Track_library

```

#Creates an empty dictionary named library.
# This dictionary will be used to store track information using track IDs (strings) as keys and
library = {}
#objects as values
#The code uses the dictionary library to add several tracks.
#For each track:
#A unique ID is used as the key (e.g., "01", "02", etc.)
library["01"] = LibraryItem("Another Brick in the Wall", "Pink Floyd", 4)
library["02"] = LibraryItem("Stayin' Alive", "Bee Gees", 5)
library["03"] = LibraryItem("Highway to Hell ", "AC/DC", 2)
library["04"] = LibraryItem("Shape of You", "Ed Sheeran", 1)
library["05"] = LibraryItem("Someone Like You", "Adele", 3)

def list_all():
    output = ""#- Initializes an empty string `output`.
    for key in library:#Iterates through all keys in the `library` dictionary.
        item = library[key]
        output += f"{key} {item.info()}\n"#Returns the complete `output` string containing information for all tracks.
    return output

def get_name(key):#retrieve an item from the `library` dictionary using the provided `key`
    try:
        item = library[key]
        return item.name #Returns the item's name using the `item.name` attribute.
    except KeyError:
        return None #Returns `None` to indicate the track wasn't found

#These functions follow the same structure as `get_name(key)`
#They use `item.artist`, `item.rating`, and `item.play_count` attributes to retrieve artist, rating, and play count respectively.
#In case of a `KeyError`, they return default values (`None` for artist and rating, -1 for play count).

```



```

def get_artist(key):
    try:
        item = library[key]
        return item.artist
    except KeyError:
        return None

def get_rating(key):
    try:
        item = library[key]
        return item.rating
    except KeyError:
        return -1

def set_rating(key, rating):#retrieve an item from the `library` dictionary using the provided `key`
    try:
        #successful (`try` block):
        #Updates the item's rating by assigning the `rating` argument to `item.rating`.
        item = library[key]
        item.rating = rating
    except KeyError:
        return

def get_play_count(key):
    try:
        item = library[key]
        return item.play_count

    except KeyError:
        return -1

def increment_play_count(key):
    # If successful, increments the `item.play_count` by 1 to record a play
    try:
        item = library[key]
        item.play_count += 1
        #If the key is not found (`except KeyError`), the function doesn't return anything
    except KeyError:
        return

```

-Front_manager:

```
import tkinter.font as tkfont # importing font dictionary

def configure():
    # family = "Segoe UI"
    family = "Helvetica"# select font family
    #Modifies Default Fonts: Configures the size and family of the default, text, and fixed-width fonts used in the application
    default_font = tkfont.nametofont("TkDefaultFont")
    default_font.configure(size=15, family=family)
    text_font = tkfont.nametofont("TkTextFont")
    text_font.configure(size=12, family=family)
    fixed_font = tkfont.nametofont("TkFixedFont")
    fixed_font.configure(size=12, family=family)
```

2. Design and Development

The Track_library implementation introduces a dynamic way to manage the music library by importing track data from a CSV file. It defines the `import_from_csv` function, which reads the CSV file, parses each row, and creates `LibraryItem` objects based on the data in the file. Each track's details, such as name, artist, rating, and play count, are stored in a dictionary where the track's unique identifier is the key. If the file is not found or if any errors occur during the data import process, appropriate exceptions are handled. The rest of the functionality for managing track details, such as retrieving and updating track information or incrementing play counts, remains similar to the previous implementation, ensuring consistent track management within the library. This approach allows for easier maintenance and updates to the library by simply modifying the CSV file rather than hardcoding the data.

```

ack_library.py 7 import_from_csv
import csv
from library_item import LibraryItem

# Initialize an empty library
library = {}

def import_from_csv(file_path="library_data.csv"):
    #import library data from a CSV file.
    try:
        with open(file_path, mode='r', encoding='utf-8') as file:
            reader = csv.DictReader(file)
            for row in reader:
                # Create LibraryItem objects and populate the library dictionary
                key = row["Key"]
                name = row["Name"]
                artist = row["Artist"]
                rating = int(row["Rating"])
                play_count = int(row["Play Count"])

                library[key] = LibraryItem(name, artist, rating)
                library[key].play_count = play_count
    except FileNotFoundError:
        print(f"Error: {file_path} not found.")
    except Exception as e:
        print(f"An error occurred while importing data: {e}")

```

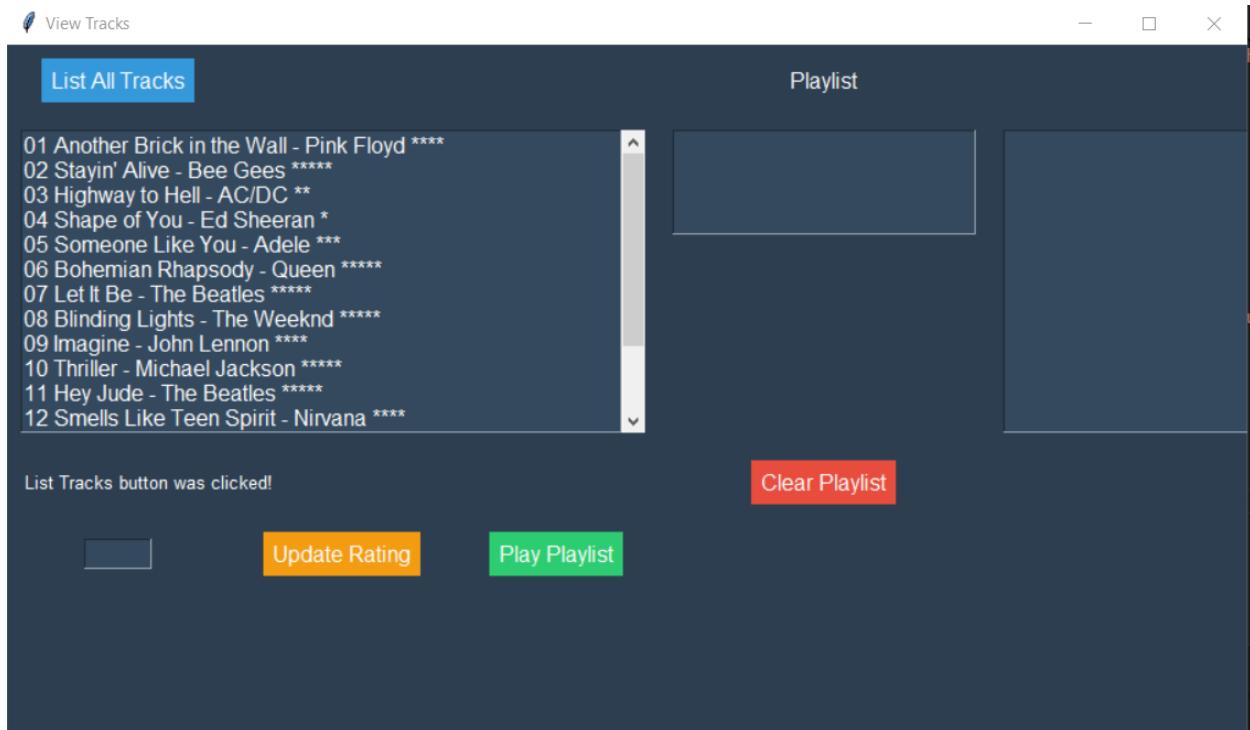
```

1  Key,Name,Artist,Rating,Play Count
2  01,Another Brick in the Wall,Pink Floyd,4,0
3  02,Stayin' Alive,Bee Gees,5,0
4  03,Highway to Hell,AC/DC,2,0
5  04,Shape of You,Ed Sheeran,1,0
6  05,Someone Like You,Adele,3,0
7  06,Bohemian Rhapsody,Queen,5,0
8  07,Let It Be,The Beatles,5,0
9  08,Blinding Lights,The Weeknd,5,0
10 09,Imagine,John Lennon,4,0
11 10,Thriller,Michael Jackson,5,0
12 11,Hey Jude,The Beatles,5,0
13 12,Smells Like Teen Spirit,Nirvana,4,0
14 13,Hotel California,Eagles,4,0
15 14,Rolling in the Deep,Adele,5,0
16 15,Take on Me,a-ha,4,0
17 |

```

In the Track_play.py, I implemented 2 more function button which are “Create Track List” and “Update Track” will lead to the “view_track.py” which will have all the functions mentioned above

After click at one of them, the window of “track_player.py” will disappear , and “view_track.py” display will pop up:



The `TrackViewer` class handles the primary interactions. The `list_tracks_clicked` function populates the track list, while the `track_clicked` function updates the details when a user selects a track. Users can update the track's rating through an entry field and can play the playlist, which increments play counts for the tracks. Additionally, the playlist can be cleared, and users can see the updated list of tracks in the playlist. Furthermore, replace the input key to play the song with click on the song to play.

In this window, users can play a song when they click on a track, at the same times, the song will be added to the playlist box (GUI display only). Moreover users can play the same song, but it will not be added again to the playlist box

List All Tracks

Playlist

01 Another Brick in the Wall - Pink Floyd ****

02 Stayin' Alive - Bee Gees *****

03 Highway to Hell - AC/DC **

04 Shape of You - Ed Sheeran *

05 Someone Like You - Adele ***

06 Bohemian Rhapsody - Queen *****

07 Let It Be - The Beatles *****

08 Blinding Lights - The Weeknd *****

09 Imagine - John Lennon *****

10 Thriller - Michael Jackson *****

11 Hey Jude - The Beatles *****

12 Smells Like Teen Spirit - Nirvana ****

Playing:

Blinding Lights

The Weeknd

Rating: 5

Playlist:

Shape of You

Bohemian Rhapsody

Blinding Lights

Imagine

Let It Be

Someone Like You

'Blinding Lights' is already in the playlist.

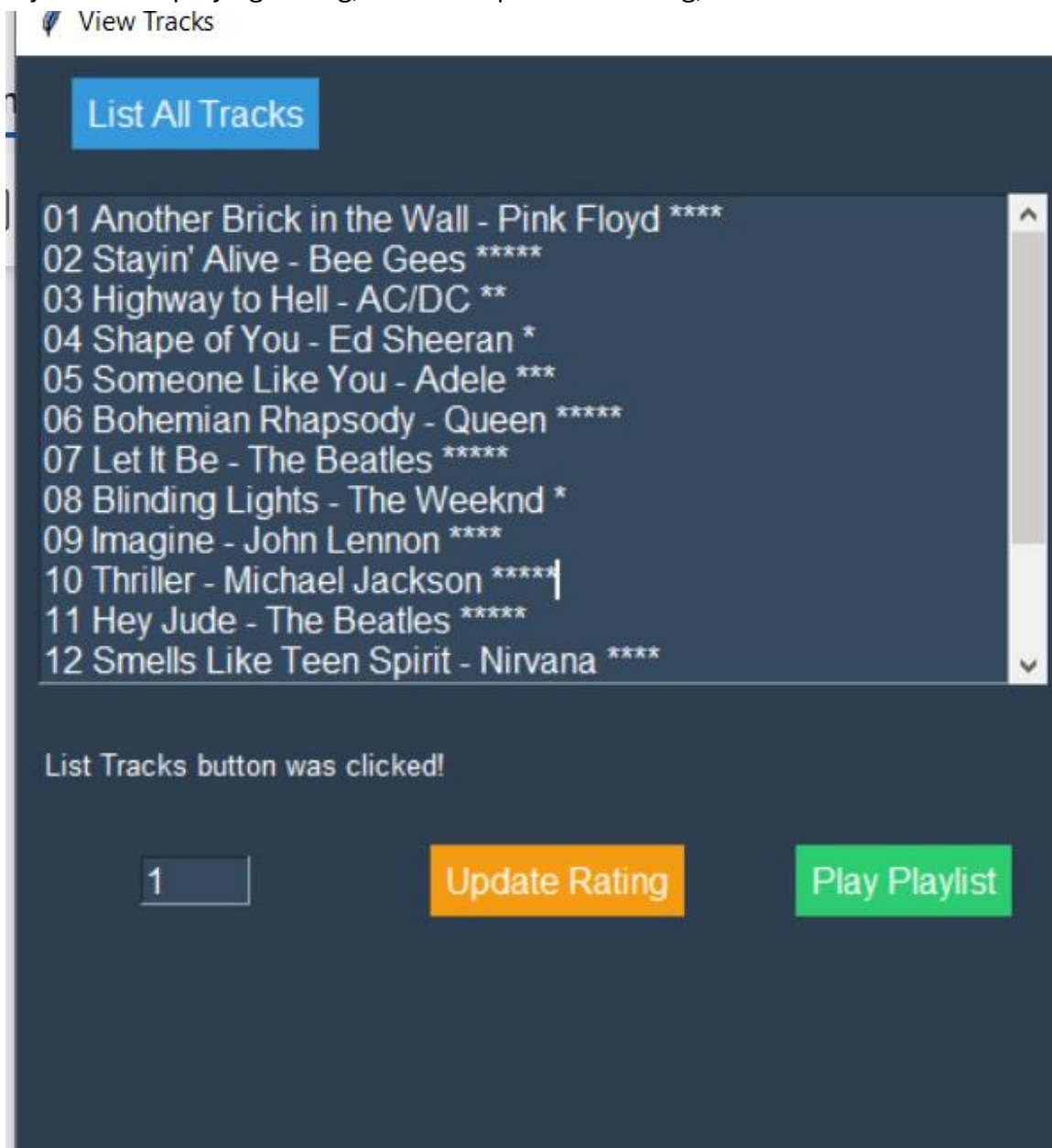
Clear Playlist

5

Update Rating

Play Playlist

By the times playing a song, user can update the rating, and it will be saved in the song list



This implementation provides an intuitive and interactive music track management system using a graphical user interface (GUI) built with tkinter. The design focuses on a user-friendly experience, enabling users to view, select, and interact with a list of tracks, modify their ratings, and manage a playlist. The layout includes a main window with buttons for different functionalities such as viewing, creating, or updating tracks, while a secondary window (TrackViewer) offers detailed track management. The interface allows users to see track details, update ratings, add tracks to a playlist, and view associated images.

The display is well-organized and includes several key elements. At the top, there are buttons for "View Tracks," "Create Track List," and "Update Tracks," which trigger various functions within the application. Below, a scrolled text area displays all available tracks, and users can click on a track to view its detailed information, such as its name, artist, rating, and play count. A separate section shows a playlist, which updates dynamically as users add or remove tracks, and a button is provided to clear the playlist. The track image is displayed next to its details, providing a visual representation of the selected track. An entry field is available for users to update the rating of the current track, with the updated rating reflected in the list. A status label at the bottom provides feedback or error messages, such as notifying the user when a track is added to the playlist or if any errors occur during an operation.

This layout is designed to be both functional and visually appealing, with a modern, clean interface and easy navigation. The color scheme and button styles enhance the user experience, making it simple for users to manage and interact with their music library and playlists.

3. Testing and validation

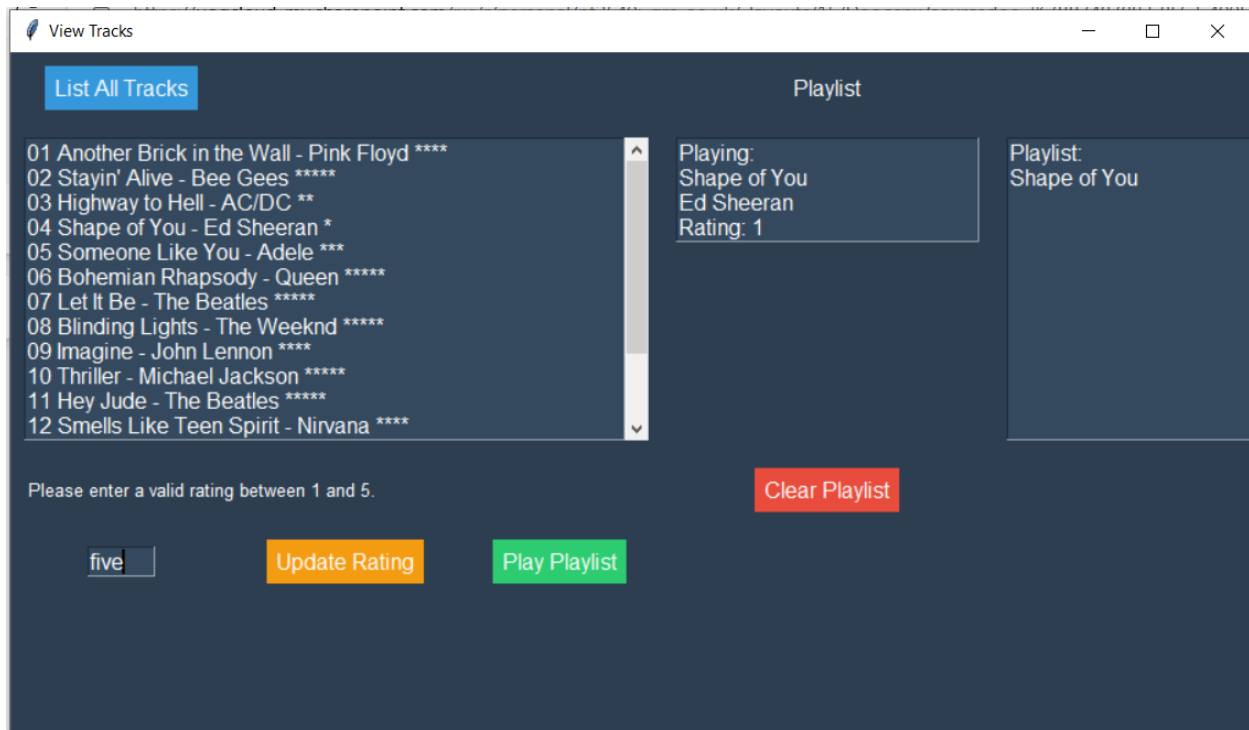
The validation needs to ensure that users enter valid inputs, such as numbers in a rating field, instead of strings like "four" or "five" in a rating entry field. We can achieve this by adapting the code to perform proper input checks, particularly where the user inputs a rating or other numeric values.

For example, the rating entry needs to ensure that:

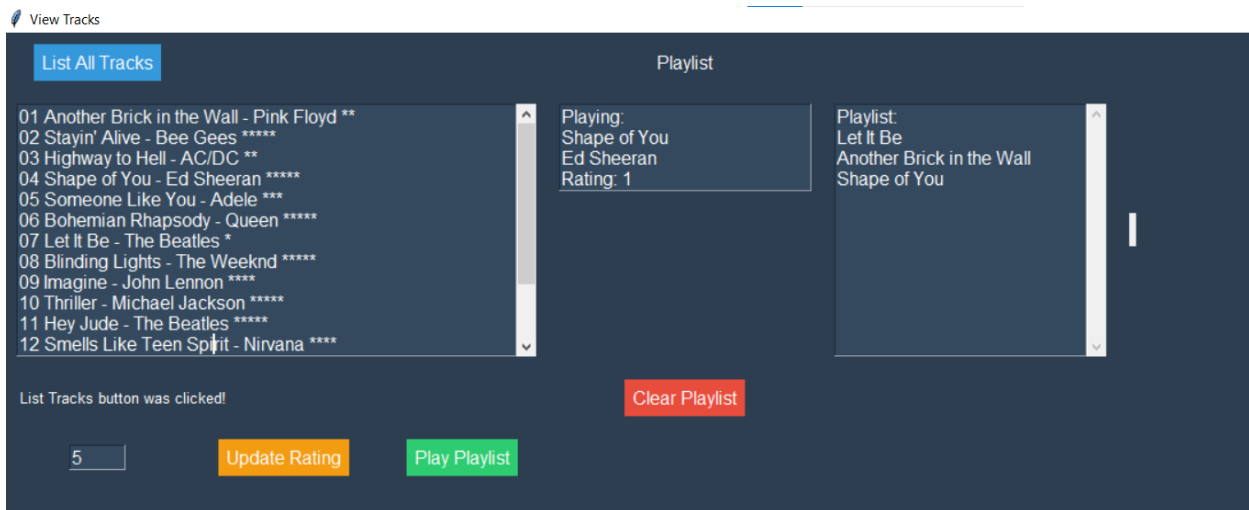
- The input is a valid number between 1 and 5.


- The input should not be a string like "four", but rather "4".

- If invalid input is detected, an error message should be displayed to the user.



As the user enters a non-value particular a string, words, the system will not recognize, and the output will be invalid “please enter a rating between 1 and 5”. In additional, users can enter number between 1 to 5 in following way “01” or “1”



 View Tracks

List All Tracks

01 Another Brick in the Wall - Pink Floyd **

02 Stayin' Alive - Bee Gees *****

03 Highway to Hell - AC/DC **

04 Shape of You - Ed Sheeran ***

05 Someone Like You - Adele ***

06 Bohemian Rhapsody - Queen *****

07 Let It Be - The Beatles *

08 Blinding Lights - The Weeknd *****

09 Imagine - John Lennon *****

10 Thriller - Michael Jackson *****

11 Hey Jude - The Beatles *****

12 Smells Like Teen Spirit - Nirvana *****

List Tracks button was clicked!

03

Update Rating

Play Playlist

Design unit testing of the LibraryItem class using PyTest:

```
test_library_item.py > test_library_item_initialization
1 import pytest
2 from library_item import LibraryItem
3
4 # Test Initialization
5 def test_library_item_initialization():
6     # Test with default rating
7     item = LibraryItem(name="Song 1", artist="Artist 1")
8     assert item.name == "Song 1"
9     assert item.artist == "Artist 1"
10    assert item.rating == 0
11    assert item.play_count == 0
12
13    # Test with specific rating
14    item2 = LibraryItem(name="Song 2", artist="Artist 2", rating=3)
15    assert item2.name == "Song 2"
16    assert item2.artist == "Artist 2"
17    assert item2.rating == 3
18    assert item2.play_count == 0
19
20    # Test the info() method
21    def test_info_method():
22        item = LibraryItem(name="Song 1", artist="Artist 1", rating=2)
23        expected_info = "Song 1 - Artist 1 ***"
24        assert item.info() == expected_info
25
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
===== 1 failed, 3 passed in 0.18s =====
PS C:\Users\tes\the pro> pytest
===== test session starts =====
platform win32 -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\tes\the pro
collected 4 items

test_library_item.py .... [100%]
===== 4 passed in 0.04s =====
PS C:\Users\tes\the pro>
```

III. Conclusion

In conclusion, the Jukebox Simulation project has successfully implemented a comprehensive music track management system using Python, with a strong focus on object-oriented programming and graphical user interface (GUI) development using Tkinter. Throughout the project, key features such as creating, viewing, updating, and rating tracks have been successfully integrated, offering users a seamless experience when interacting with the system. The ability to import track data from a CSV file and manage playlists dynamically further enhances the user's ability to manage their music collection effortlessly.

The development process also emphasized the importance of robust testing and validation. By implementing thorough unit tests using PyTest, we were able to ensure the correctness of the `LibraryItem` class and the overall functionality of the application. The validation of user input, such as ensuring numeric ratings between 1 and 5, guarantees that the system is both user-friendly and resistant to invalid inputs.

In terms of design, the GUI was carefully crafted to be both visually appealing and functional, incorporating modern design principles that make navigation intuitive and

engaging. With the clear organization of components such as track details, playlist management, and the ability to update ratings, the system not only meets the requirements but also provides an enjoyable and interactive experience for users.

Overall, this project demonstrates a solid understanding of core Python programming concepts, object-oriented design, and effective GUI development. The successful integration of these elements into a cohesive application underscores the value of testing, validation, and iterative development to create a reliable, user-friendly music track management system.

- *What did I achieve with this element of learning? Which were the most difficult parts, and why were they difficult for me? Which were the most straightforward parts, and why did I find these easy?*

Through this project, I achieved significant growth in my understanding of Python's object-oriented programming (OOP) principles and the development of graphical user interfaces (GUIs) using Tkinter. I successfully implemented key features such as track rating updates, playlist creation, and dynamic data management through CSV files, enhancing the usability and functionality of the Jukebox Simulation. Additionally, the project allowed me to strengthen my debugging skills by ensuring robust testing and validation, including user-friendly error handling for inputs.

However, certain aspects of the project were challenging. Understanding the provided template code, such as `track_player.py` and `view_tracks.py`, was initially difficult due to the lack of clear documentation. It required a deep dive into the code structure to trace the flow of functions and dependencies. GUI design also posed a challenge as Tkinter's limited styling options made it harder to create a modern, user-friendly interface while maintaining functionality. Furthermore, implementing input validation for track ratings required extra effort to account for edge cases like invalid characters or numbers outside the accepted range.

On the other hand, some parts of the project felt more straightforward. Implementing basic class structures and methods was manageable due to my familiarity with OOP concepts from previous experience. Importing and managing data from CSV files was also relatively easy, thanks to Python's user-friendly `csv` module and the structured format of the data. Similarly, adding functionality for playing tracks was clear-cut since the requirements for this feature were well-defined and straightforward to integrate.

Overall, this project balanced challenges that deepened my problem-solving skills with areas of familiarity that reinforced my confidence in programming