UNIVERSITY of GREENWICH

Alliance with FPT Education

PROJECT FRONT SHEET

Class	**Object Oriented Programming**		
Submission date	22/04/2025		
Student Name	Nguyen Manh Hung	**Student ID**	001419592
Class	**COMP-1752**	**Assessor name**	Nguyen The Nghia
❒ **Summative Feedback:** **Feedback:**			
Grade:	**Assessor Signature:**	**Date:**	

Table of Contents

1. Introduction

This report presents the development of a Jukebox simulation application designed to manage and organize a music library. The project was developed using Python, with the Tkinter library utilized to build a user-friendly graphical interface. The application enables users to browse music tracks, create and manage playlists, and update track metadata efficiently.

Track data is stored and managed using CSV files:

- **music.csv**: Main track database containing track name, artist, and rating
- **user.csv**: User login information
- **list1.csv, list2.csv**: Saved playlists with selected track IDs and status

The development process followed the five key stages outlined in the coursework, integrating various features such as a unified GUI design, dynamic playlist control, track search capabilities, and data validation. The application meets all fundamental requirements and incorporates enhancements aimed at improving usability and functionality.

2. Design and Development

a. Overview

The Jukebox application was designed as a single-window interface utilizing multiple tabs to streamline navigation and functionality. This integrated approach improves usability and provides a consistent user experience, compared to earlier designs that relied on multiple separate windows. The application's core components include:

- **View Tracks**: Displays all available tracks with integrated search and filtering options
- **Playlists**: Supports the creation, editing, and saving of user-defined playlists
- **Update Library**: Offers tools to add new tracks and update existing track information, including validation

This modular design promotes maintainability while ensuring all essential features are easily accessible.

b. Project Structure

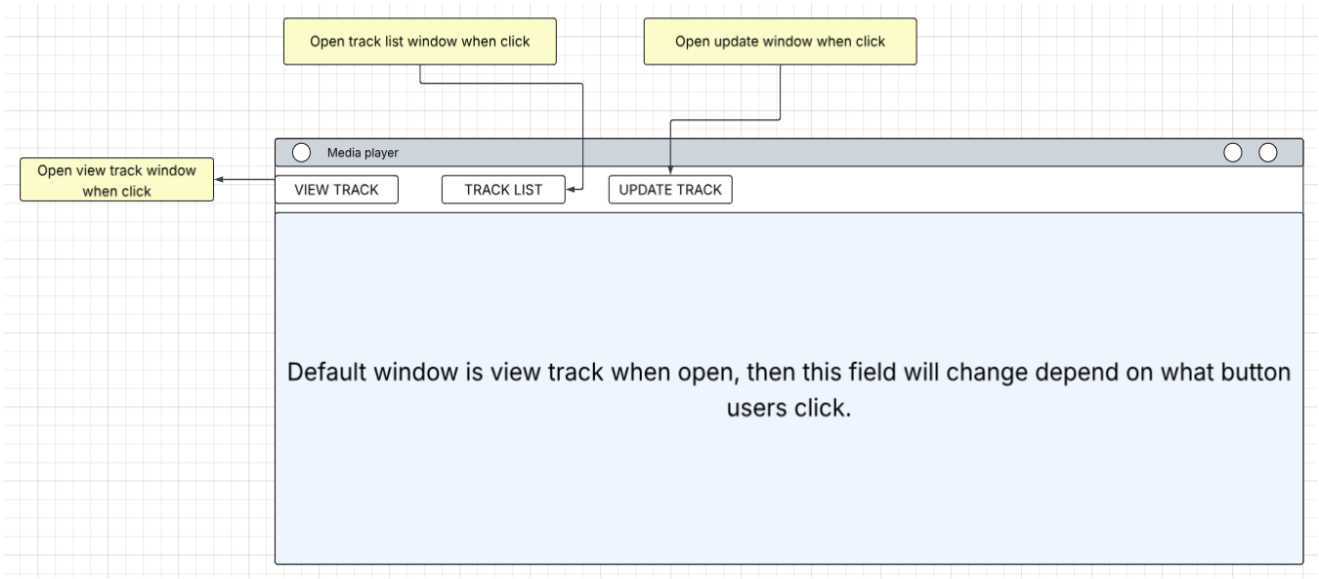
The application is organized into several Python files, each fulfilling a distinct role in the system:

- **Login.py**: Handles user authentication and registration (additional feature)
- **jukebox.py**: Main entry point that initializes the graphical user interface and integrates core components
- **library_item.py**: Defines the LibraryItem class representing individual music tracks
- **track_library.py**: Manages the music database and provides functions to retrieve and update track data
- **viewtrack.py**: Enables users to view, search, and display track details
- **tracklist.py**: Manages playlist creation, track selection, and playlist persistence
- **update.py**: Allows users to add, edit, and remove tracks from the music library
- **font_manager.py**: Applies custom font settings across the application interface

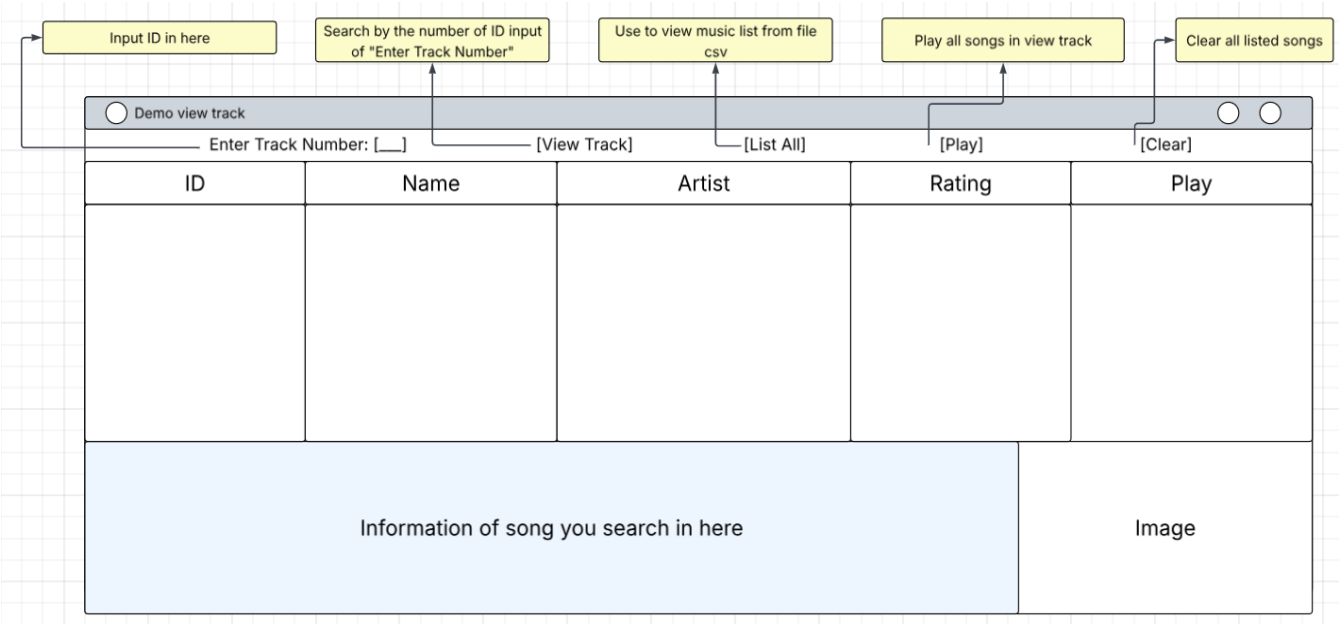
c. UI Design

The interface was designed using Tkinter with a modern, user-friendly layout. Key design elements include:

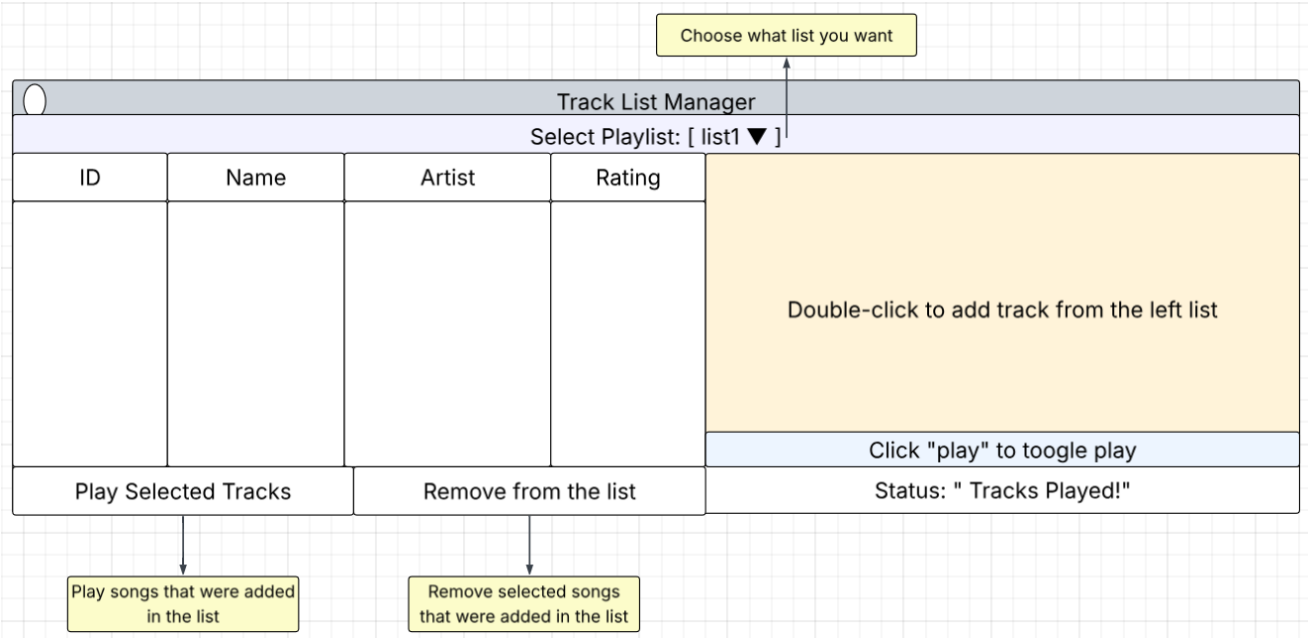
1. **Main Interface**



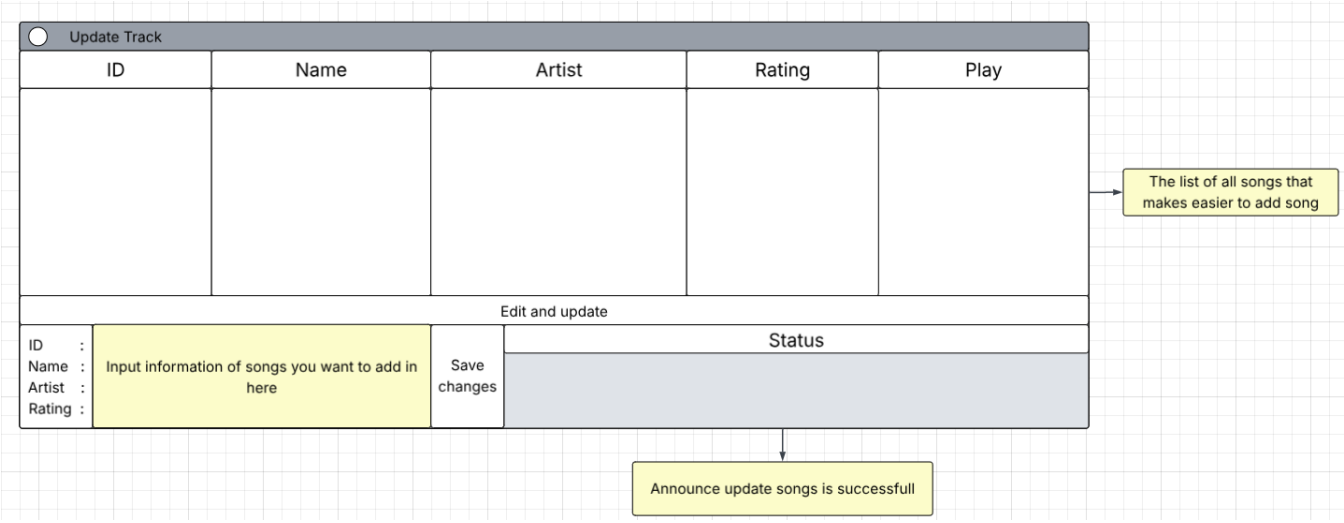
2. **View Track**



3. **Track List**



4. **Update Track**



d. Data Management

File Structures:

1. **music.csv**:

```
name,artist,rating
"Another Brick in the Wall","Pink Floyd",4
"Stayin' Alive","Bee Gees",5

```

2. **Playlist files (list1.csv):**

```
01,1
03,0
```

Implementation Details:

Track Library: track_library.py loads/saves from music.csv

```
def load_library(filename="music.csv"):
    with open(filename, "r") as file:
        reader = csv.reader(file)
        next(reader)  # Skip header
        for index, row in enumerate(reader, start=1):
            library[f"{index:02d}"] = LibraryItem(row[0], row[1], int(row[2]))

```

3. Development Stages

a. Stage 1: Basic Understanding

The Jukebox application is a Python program with a GUI built using tkinter. It allows users to view local tracks, manage playlists, and update song metadata. All track data is stored in a CSV file.

The app is divided into three main features:

- **View Tracks**: Displays all songs in a table format.
- **Playlists**: Users can add or remove tracks to create custom playlists.
- **Update Library**: Users can edit song title, artist, and number of plays.

Each feature is implemented in a separate Python file for modularity. The GUI includes a top navigation bar to switch between tabs. The system does not include online search or music playback, focusing only on local data management.

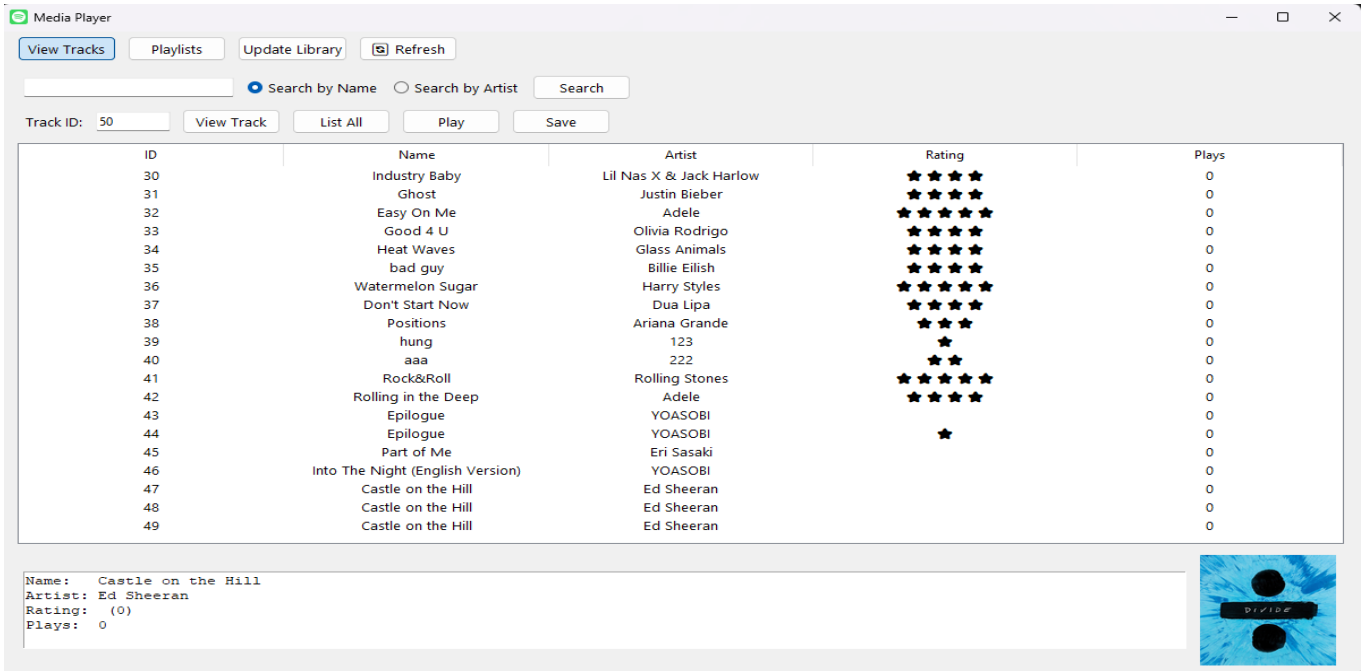
b. Stage 2: Outline Implementation

1. Overview

The Jukebox application was developed using Python and the tkinter library for the graphical user interface. The application is focused on managing track data and playlists. The two core functionalities remaining are:

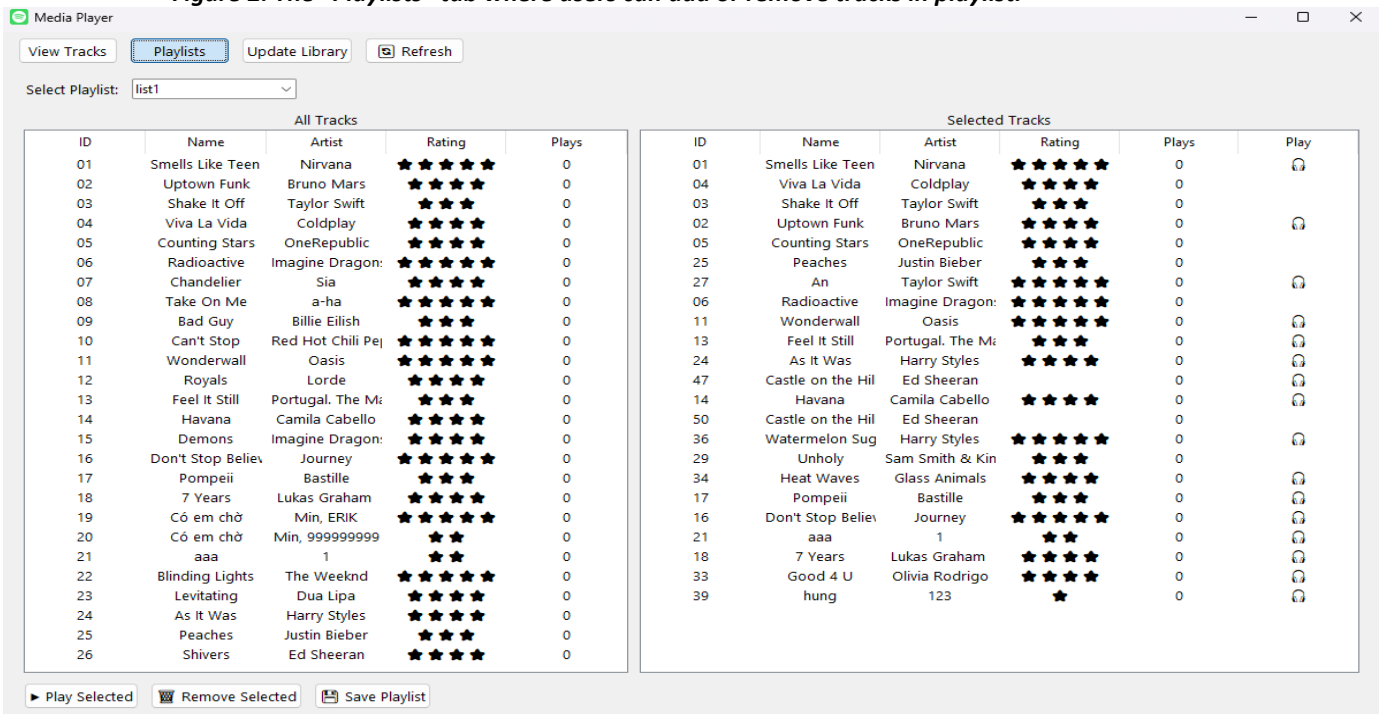
- **View Tracks**: Allows users to browse all available songs stored locally.

Figure 1: The "View Tracks" tab showing a table of all available songs stored locally.



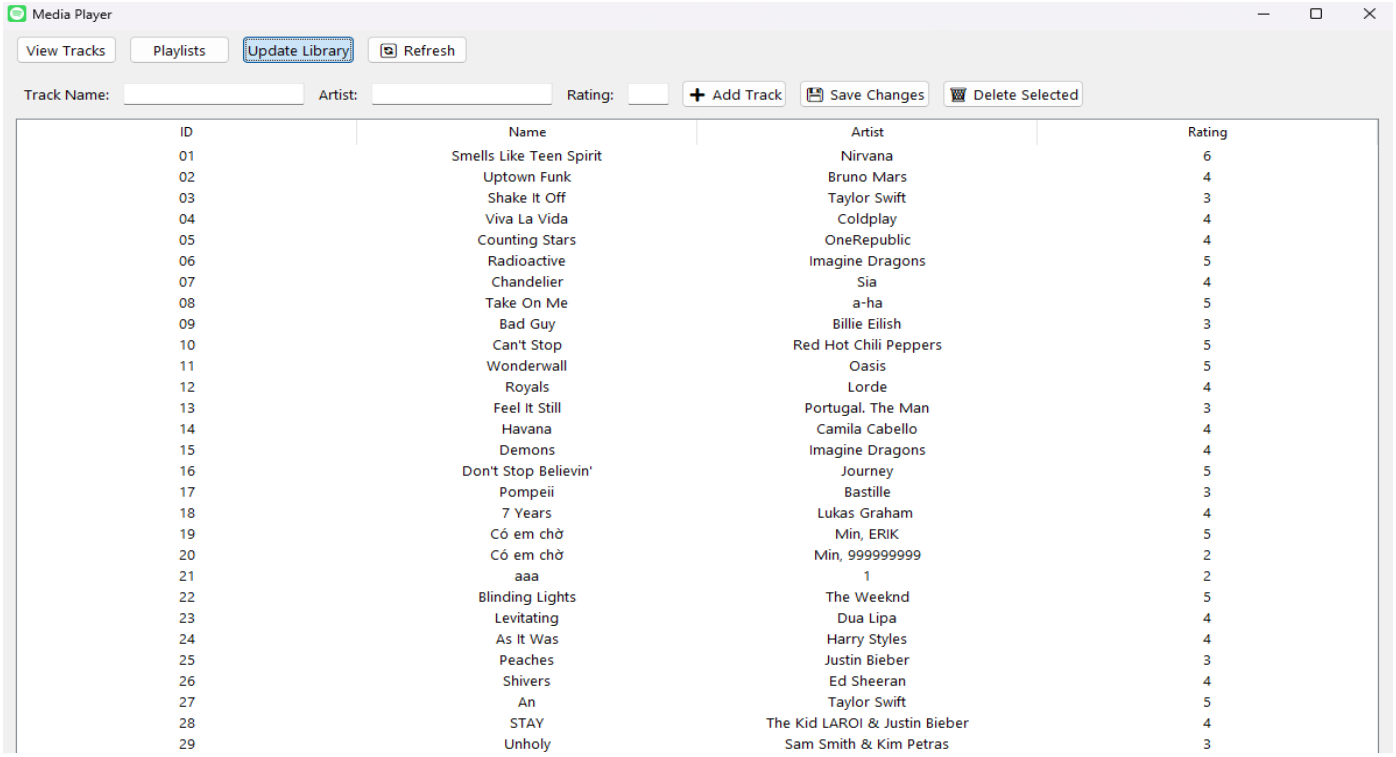
- **Playlists**: Users can build and save custom playlists from the available tracks.

Figure 2: The "Playlists" tab where users can add or remove tracks in playlist.



- **Update Library**: Enables editing of track metadata such as title, artist, and number of plays.

Figure 3: The "Update Library" tab allowing users to edit song metadata such as title, artist, and number of plays.



Track data is stored and managed using CSV files, ensuring persistence between sessions.

c. Stage 3: Basic Working Version

- Each feature is implemented in a dedicated Python module:
    - view_tracks.py: Displays all track records in a table format.
    - create_track_list.py: Allows adding/removing tracks from playlists.
    - update_tracks.py: Handles editing track metadata and updating the CSV file.
    - jukebox.py: The main GUI file that connects all modules and controls navigation.
- The GUI is organized into frames, with navigation handled by custom-styled buttons at the top of the window.

d. Stage 4: Testing and Validation

Extensive testing was conducted to ensure proper functionality:

- Input validation for track IDs, ratings, etc.
- Unit tests for core functionality
- Integration testing of all components

e. Stage 5: Innovations

Several innovative features were added:

1. **Enhanced Playlist System**: Save/load playlists with track selection
2. **Click to add tracks to list**: Add tracks more quickly
3. **Search Functionality**: Search tracks by name or artist

4. **Track Images**: Display album art for tracks
5. **Single GUI**: Improve ease of use and testing
6. **Allow editing by clicking in update**: Edit information of track more quickly

4. Testing and Validation

A comprehensive testing strategy was implemented:

A. Unit Testing

PyTest was used to test core functionality of the LibraryItem class:

1. Test loading data from CSV file

```
1.  def test_load_library():
2.      """Test loading library from CSV"""
3.      lib.load_library("music.csv")
4.      assert len(lib.library) > 0  # Check if library is loaded
5.      assert lib.get_name("01") == "Smells Like Teen Spirit"
6.      assert lib.get_artist("01") == "Nirvana"
7.      assert lib.get_rating("01") == 5  # Rating 6 should be converted to 5
8.
```

3. Test displaying the list of tracks

```
1. def test_list_all():
2.      """Test list_all() output format"""
3.      lib.load_library("music.csv")
4.      output = lib.list_all()
5.      assert "01\tSmells Like Teen Spirit\tNirvana\t★ ★ ★ ★ ★\t0" in output
6.      assert "02\tUptown Funk\tBruno Mars\t★ ★ ★ ★\t0" in output
```

4. Test information retrieval functions

```
1. def test_get_methods():
2.      """Test all getter methods"""
3.      lib.load_library("music.csv")
4.      assert lib.get_name("01") == "Smells Like Teen Spirit"
5.      assert lib.get_artist("01") == "Nirvana"
6.      assert lib.get_rating("01") == 5
7.      assert lib.get_play_count("01") == 0
8.      assert lib.get_name("99") is None  # Non-existent ID
9.
```

5. Test changing track ratings

```
def test_set_rating():
    """Test updating track rating"""
    lib.load_library("music.csv")
    lib.set_rating("01", 2)
    assert lib.get_rating("01") == 2
    # Verify unchanged play count
    assert lib.get_play_count("01") == 0

```

6. Test incrementing play count

```
def test_increment_play_count():
    """Test play count increment"""
    lib.load_library("music.csv")
    lib.increment_play_count("01")
    assert lib.get_play_count("01") == 1
    lib.increment_play_count("01")
    assert lib.get_play_count("01") == 2
```

7. Test handling of Non-existent track in data

```
def test_nonexistent_track_operations():
    """Test operations on non-existent tracks"""
    lib.load_library("music.csv")
    # Should not raise errors but return default values
    assert lib.get_name("99") is None
    assert lib.get_play_count("99") == -1
    # These should silently fail (no exception)
    lib.set_rating("99", 3)
    lib.increment_play_count("99")

```

8. Test handling of missing files

```
def test_file_not_found():
    """Test handling of missing CSV file"""
    with pytest.raises(FileNotFoundError):
        lib.load_library("nonexistent.csv")

```

9. Tests detailed track information

```
def test_track_information():
    """Test getting information of specific tracks"""
    lib.load_library("music.csv")

    assert lib.get_name("01") == "Smells Like Teen Spirit"
    assert lib.get_artist("01") == "Nirvana"
    assert lib.get_rating("01") == 5
    assert lib.get_play_count("01") == 0

```

10. Tests rating conversion

```
def test_track_rating_conversion():
    """Test rating conversion for different values"""
    lib.load_library("music.csv")

    # Test rating above 5 (should be converted to 5)
    assert lib.get_rating("01") == 5  # Original rating is 6

    # Test rating below 0 (should be converted to 0)
    assert lib.get_rating("43") == 0  #  Original rating is -1
```

11. Tests handling of duplicate tracks

```
def test_track_duplicates():
    """Test handling of duplicate tracks"""
    lib.load_library("music.csv")

    # Test tracks with same name but different artists
    assert lib.get_name("09") == "Bad Guy"
    assert lib.get_artist("09") == "Billie Eilish"
    assert lib.get_rating("09") == 3

    assert lib.get_name("35") == "bad guy"
    assert lib.get_artist("35") == "Billie Eilish"
    assert lib.get_rating("35") == 4

    # Test multiple tracks with same name and artist
    assert lib.get_name("43") == "Epilogue"
    assert lib.get_artist("43") == "YOASOBI"
    assert lib.get_rating("43") == 0

    assert lib.get_name("44") == "Epilogue"
    assert lib.get_artist("44") == "YOASOBI"
    assert lib.get_rating("44") == 1

```

12. Test the creation of a Library_Item object

```
1. def test_library_item_creation(sample_library_item):
2.     """Test LibraryItem initialization with real data"""
3.     assert sample_library_item.name == "Smells Like Teen Spirit"
4.     assert sample_library_item.artist == "Nirvana"
5.     assert sample_library_item.rating == 5  # Rating 6 should be converted to 5
6.     assert sample_library_item.play_count == 0
```

13. Test changing the rating

```
1. def test_library_item_rating_setter(sample_library_item):
2.     """Test rating setter with valid values"""
3.     sample_library_item.rating = 5
4.     assert sample_library_item.rating == 5
5.     sample_library_item.rating = 0
6.     assert sample_library_item.rating == 0
7.
```

14. Test handling of invalid ratings

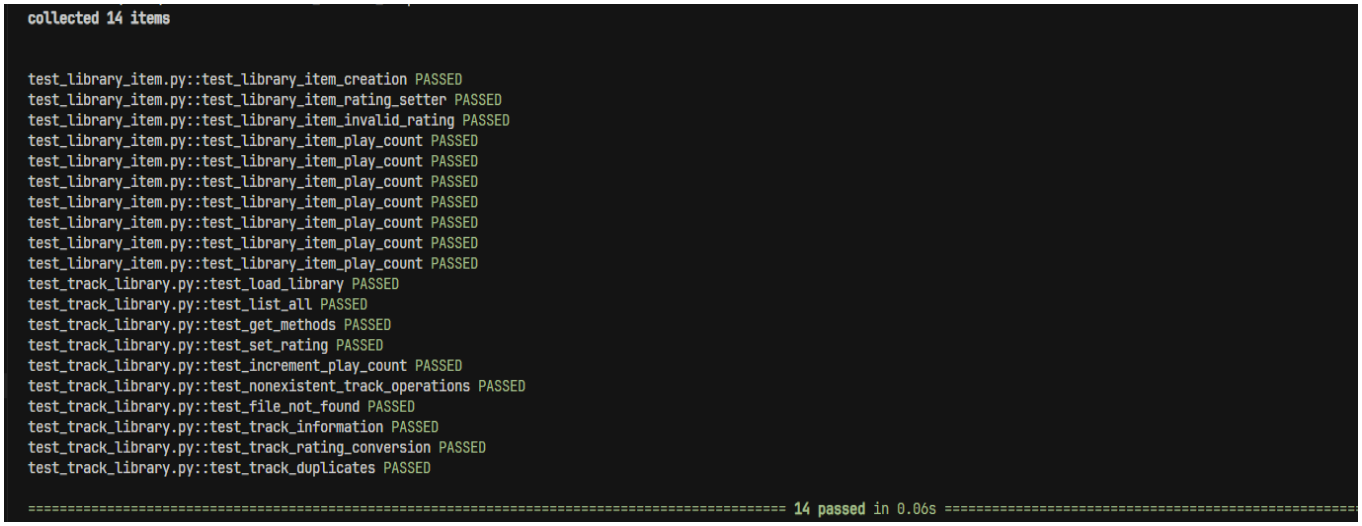
```
 1. def test_library_item_invalid_rating(sample_library_item):
 2.     """Test rating validation"""
 3.     # Test with non-integer value
 4.     sample_library_item.rating = "invalid"  # Should be converted to 0
 5.     assert sample_library_item.rating == 0
 6.
 7.     # Test with out of range values
 8.     sample_library_item.rating = -1  # Should be converted to 0
 9.     assert sample_library_item.rating == 0
10.
11.     sample_library_item.rating = 6  # Should be converted to 5
12.     assert sample_library_item.rating == 5
13.
```

15. Test incrementing play count

```
1. def test_library_item_play_count(sample_library_item):
2.     """Test play count increment"""
3.     sample_library_item.play_count += 1
4.     assert sample_library_item.play_count == 1
```

```
5.      sample_library_item.play_count += 5
6.      assert sample_library_item.play_count == 6
7.
```

Test Result



Test Case 1: Search by ID

ID	Description	Input	Expected Output	Actual Output	Evidence	Result
1.1	Single-digit ID	ID = "1"	Track 01 found	Not found	Evidence 1.1	FAILED
1.2	Two-digit ID	ID = "01"	Track 01 found	Track found	Evidence 1.2	PASSED
1.3	Negative ID	ID = "-01"	Error message	No message	Evidence 1.3	FAILED
1.4	ID = 0	ID = "00"	Error message	No message	Evidence 1.4	FAILED
1.5	Non-existent ID	ID = "99"	Track not found	Track not found	Evidence 1.5	PASSED

Test Case 2: Search by Name/Artist

ID	Description	Input	Expected Output	Actual Output	Evidence	Result
2.1	Uppercase name	"ROLLING"	Find "Rolling"	Found	Evidence 2.1	PASSED
2.2	Lowercase name	"rolling"	Find "Rolling"	Found	Evidence 2.2	PASSED
2.3	Name with spaces	" rolling"	Find "Rolling"	Not found	Evidence 2.3	FAILED
2.4	Name with special characters	"Rock&Roll"	Song found	Found	Evidence 2.4	PASSED
2.5	Name with accents	"Có em chờ"	Song found	Found	Evidence 2.5	PASSED

Test Case 3: Playlist Management

ID	Description	Input	Expected Output	Actual Output	Evidence	Result
3.1	Duplicate playlist	Name = "list1"	Ask to overwrite	Still direct to old “list1”		FAILED
3.2	Name with space	"my list"	Valid file created	File created	Evidence 3.2	PASSED
3.3	Name with special characters	"my@list"	Valid file created	File created	Evidence 3.3	PASSED
3.4	Overly long name	"a"*100	Error message	No limit	Evidence 3.4	FAILED
3.5	Play Selected songs	Click songs to appear “ 🎧 ” icon and play	Play selected	Play selected	Evidence 3.5	PASSED

			songs and count	songs and count		
3.6	Move song	Drag and drop	Position changed	Not supported		FAILED
3.7	Add the same songs to the list	Choose the song	Error message	Error message	Evidence 3.7	PASSED
3.8	Delete multiple songs in the list	Ctrl + click to choose multiple songs	Delete successfully	Delete successfully	Evidence 3.8	

Test Case 4: Update track validation

ID	Description	Input	Expected Output	Actual Output	Evidence	Result
4.1	Add duplicate song	Song already in list	Ask user, add based on choice	Added successfully	Evidence 4.1	PASSED
4.2	Remove song	Choose and Click Remove	Ask user and remove from list	Removed successfully	Evidence 4.2	PASSED
4.3	Empty name		Error message	Error message	Evidence 4.3	PASSED
4.4	Delete multiple songs	Select and delete multiple	All deleted	Deleted successfully	Evidence 4.4	PASSED
4.5	Save added-songs to csv file	Click Save changes	Save songs to csv file	Work correctly	Evidence 4.5	PASSED
4.6	Refresh all views	Click “Refresh” button	Data in all tabs updates correctly	Work correctly		PASSED

Test Case 5: Validate Rating

ID	Description	Input	Expected Output	Actual Output	Evidence	Result
5.1	Alphabetic rating	Rating = "abc"	Error message	Error message	Evidence 5.1	PASSED
5.2	Decimal rating	Rating = "3.5"	Error message	Error message	Evidence 5.2	PASSED
5.3	Negative rating	Rating = "-1"	Error message	Error message	Evidence 5.3	PASSED
5.4	Rating > 5	Rating = "6"	Error message	Error message	Evidence 5.4	PASSED
5.5	Empty rating	Rating = ""	Set default = 0	Set rating = 0	Evidence 5.5	PASSED

5. Conclusions, Further Development and Reflection

The Jukebox application successfully implements all required functionality while incorporating several innovative features. The single-window design provides a more modern user experience compared to the original multi-window approach.

Further Development

With additional time, several improvements could be made:

1. **Cloud Integration**: Sync playlists across devices
2. **Advanced Playback Features**: Equalizer, crossfade
3. **Metadata Editing**: Edit ID3 tags of audio files
4. **Social Features**: Share playlists with other users

Reflection

This project provided valuable experience in GUI development and object-oriented design in Python. Key challenges included:

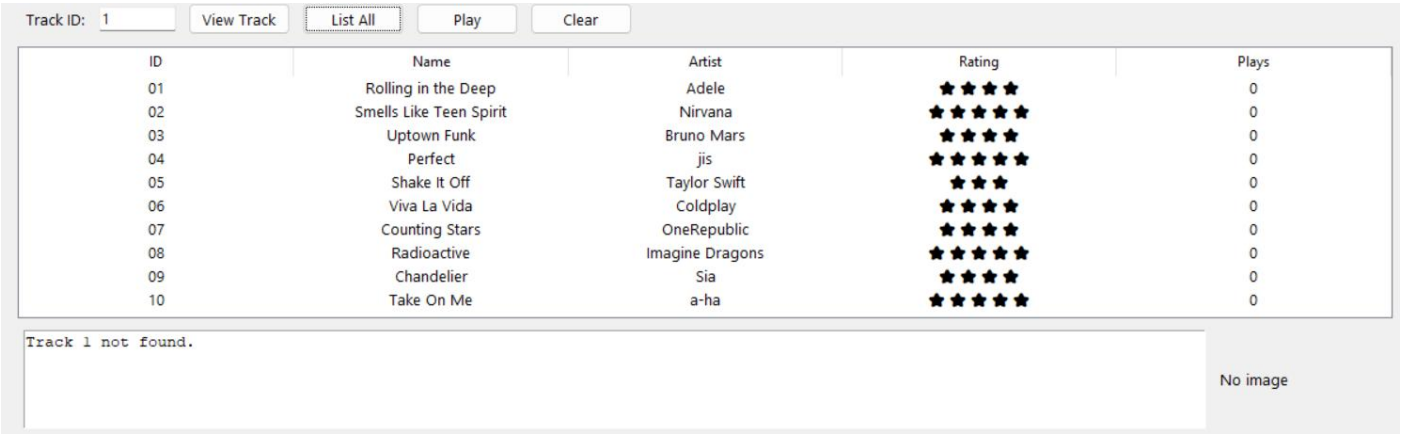
- Managing the state of the music player and keeping UI elements synchronized
- Designing an intuitive interface that accommodates all functionality
- Implementing smooth audio playback with seek functionality

The most rewarding aspect was seeing the music player functionality come together, transforming the simulation into an actual working application. This project has significantly improved my understanding of event-driven programming and GUI design principles.

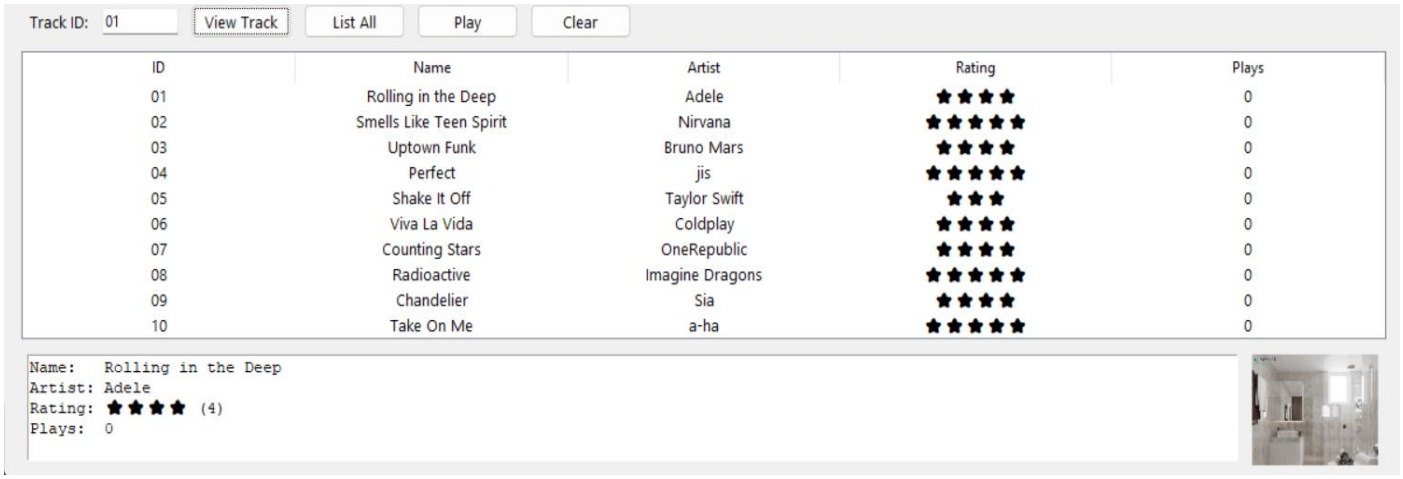
6. Appendix

B. Test Table and Results

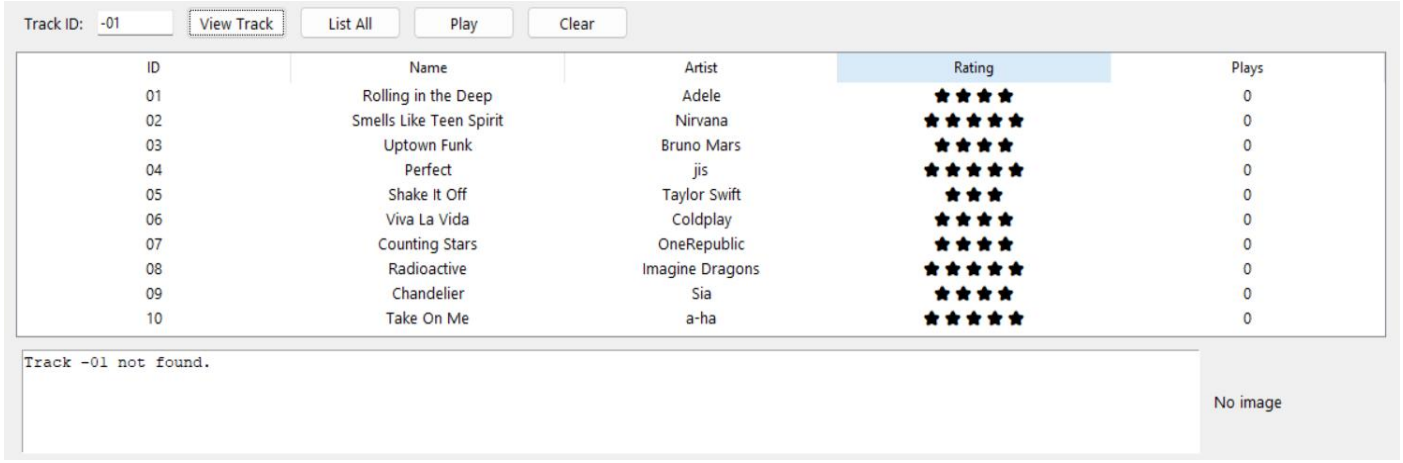
Evidence 1.1: Single-digit ID



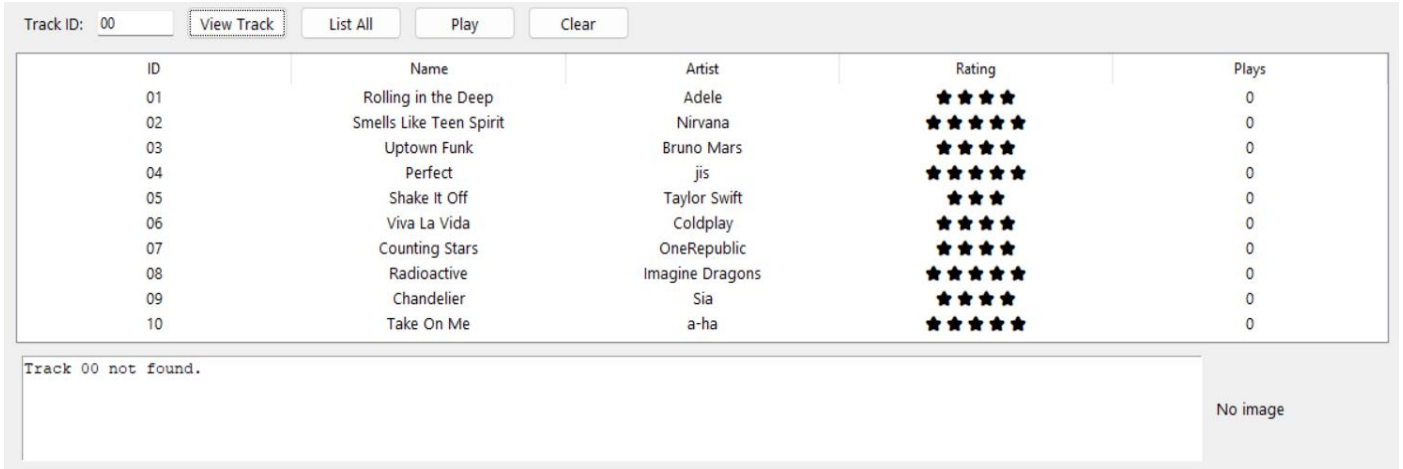
Evidence 1.2: Two-digits ID



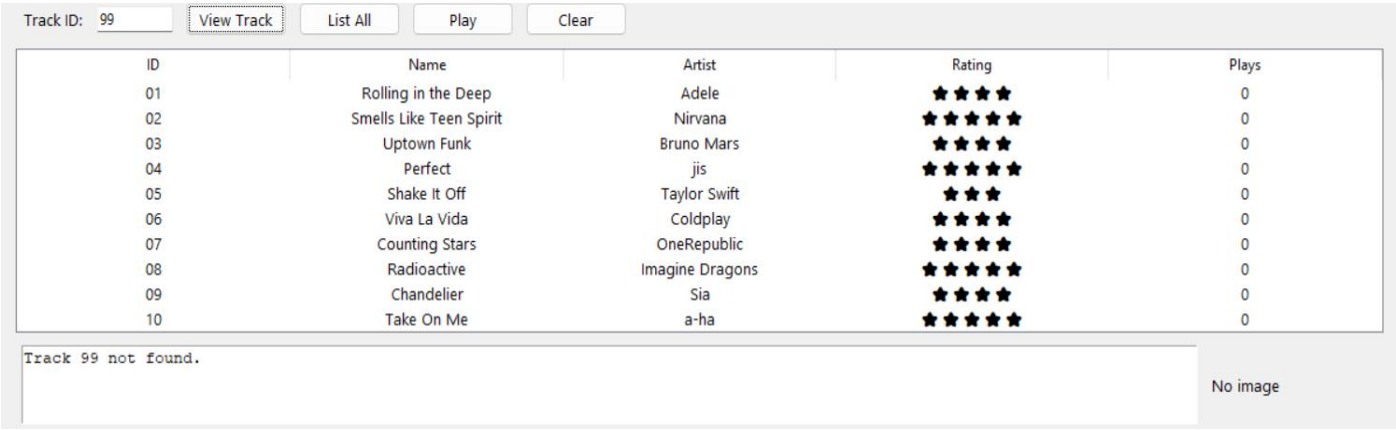
Evidence 1.3: Negative ID



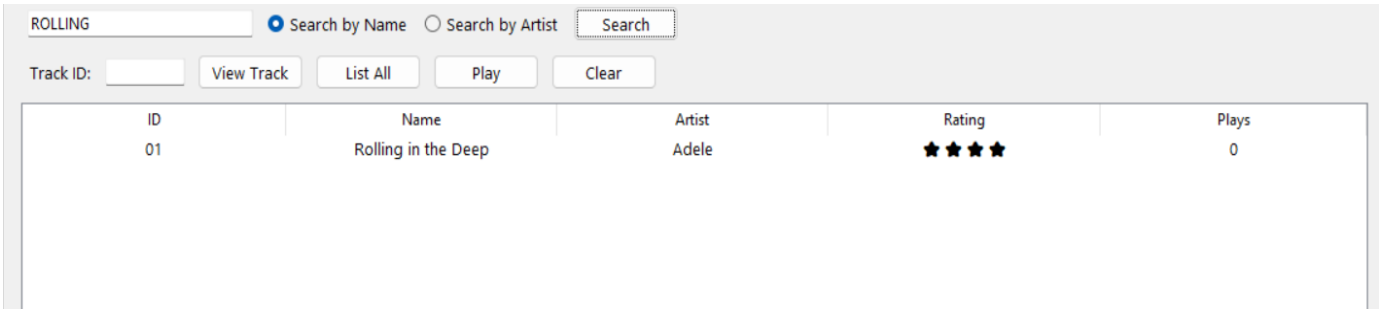
Evidence 1.4: ID = 0



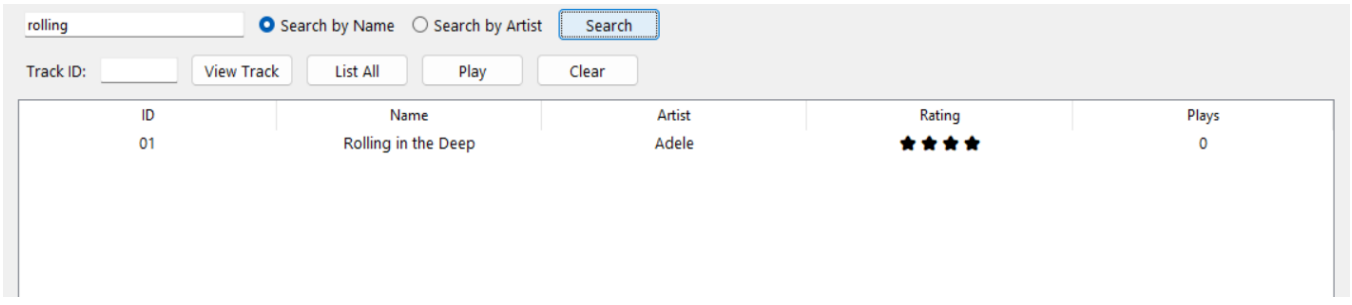
Evidence 1.5: Non-existent ID



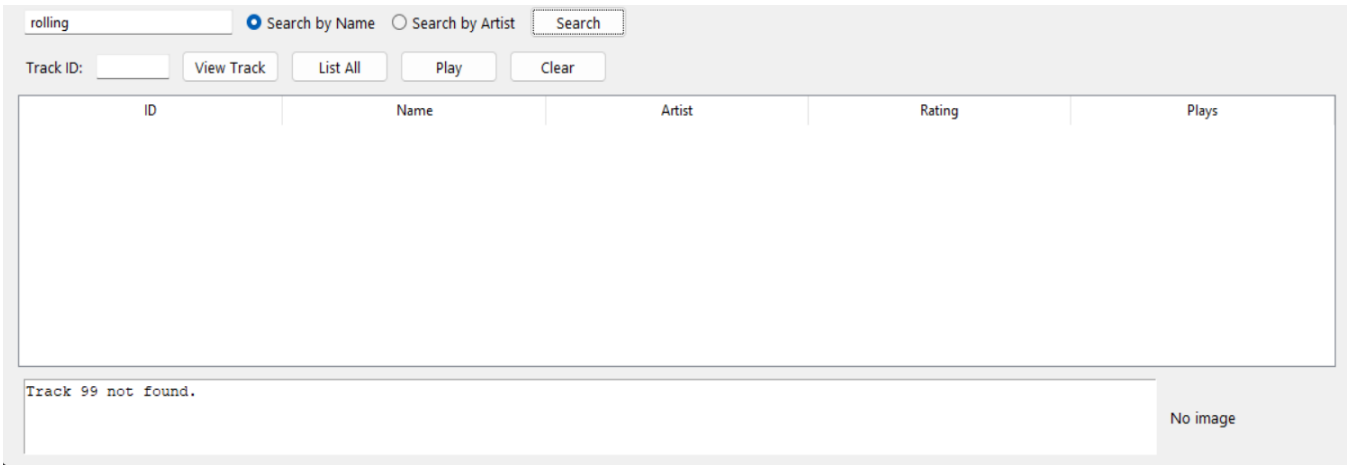
Evidence 2.1: Uppercase name



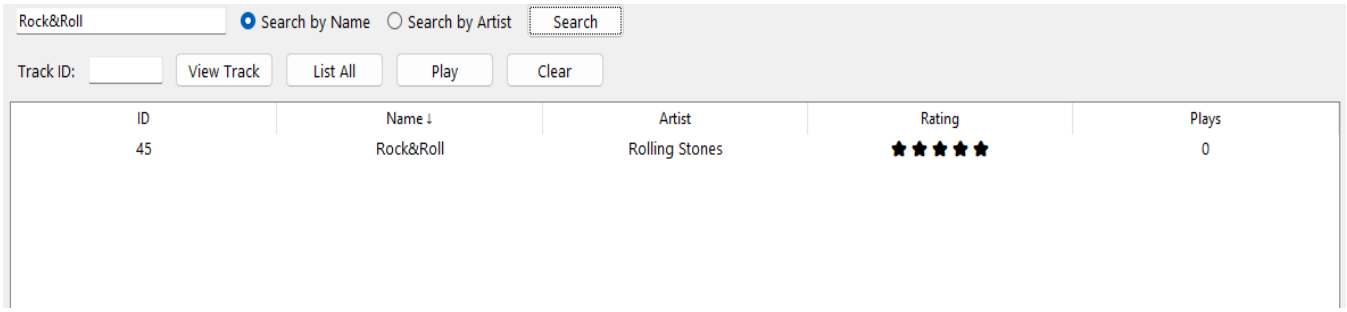
Evidence 2.2: Lowercase name



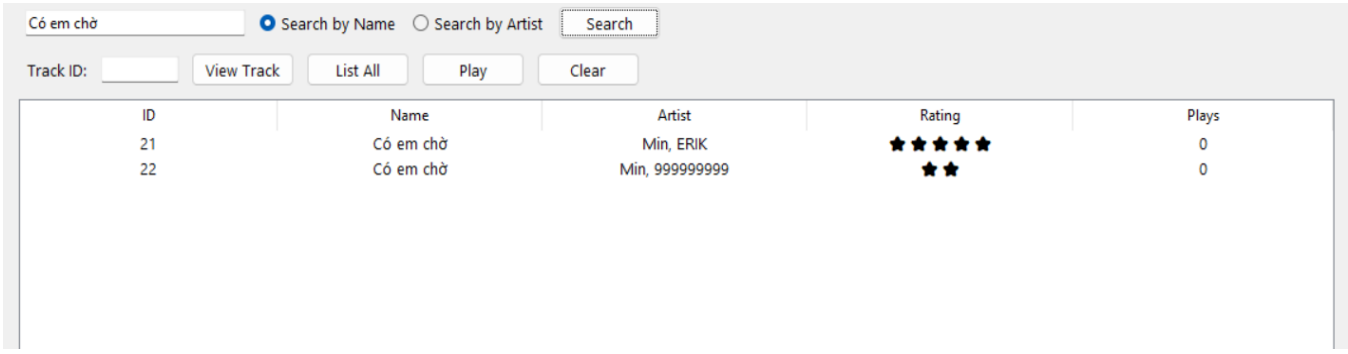
Evidence 2.3: Name with spaces



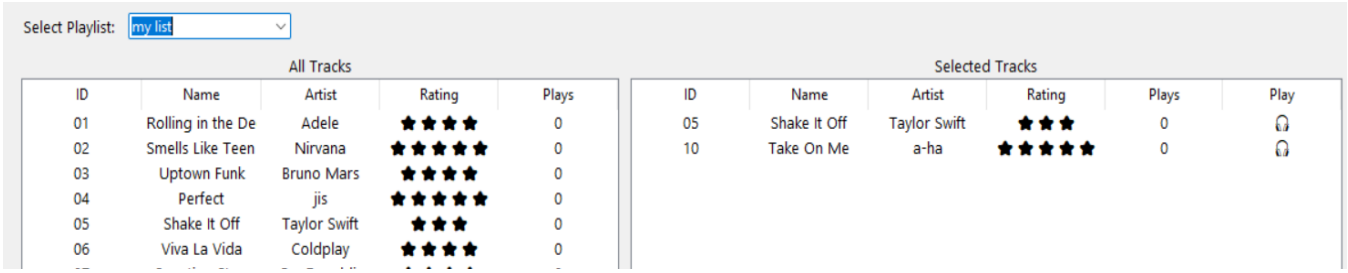
Evidence 2.4: Name with special characters



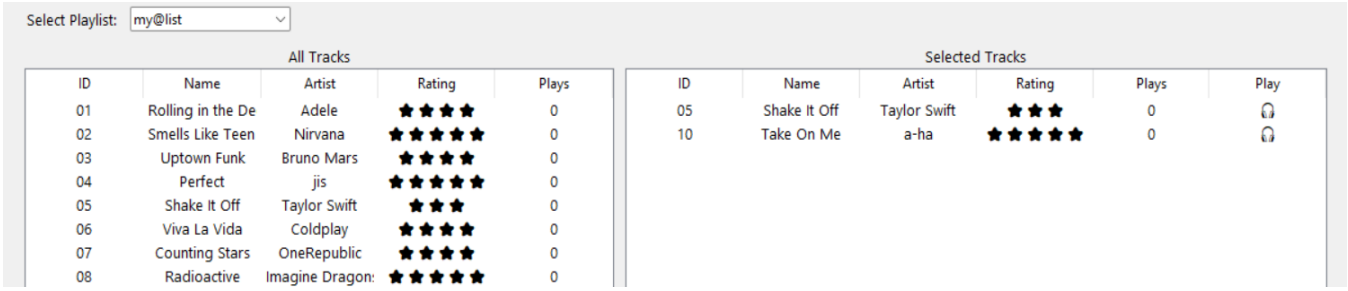
Evidence 2.5: Name with accents



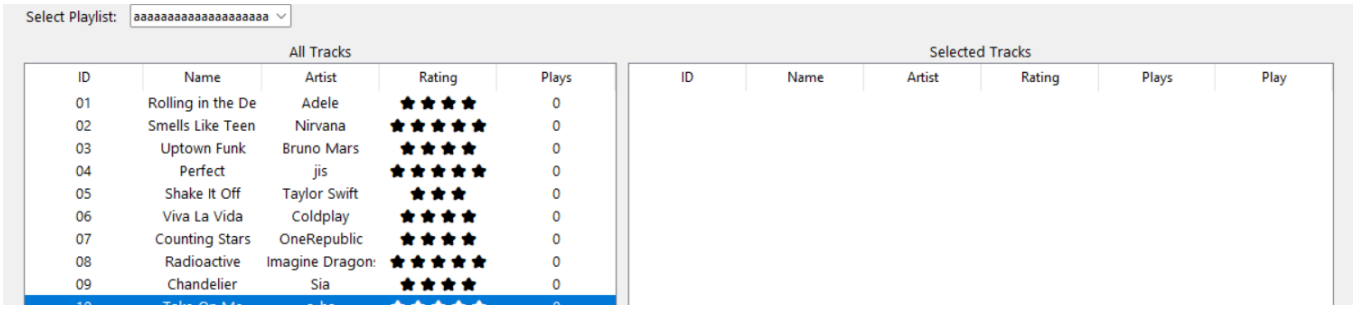
Evidence 3.2: Name with space



Evidence 3.3: Name with special characters

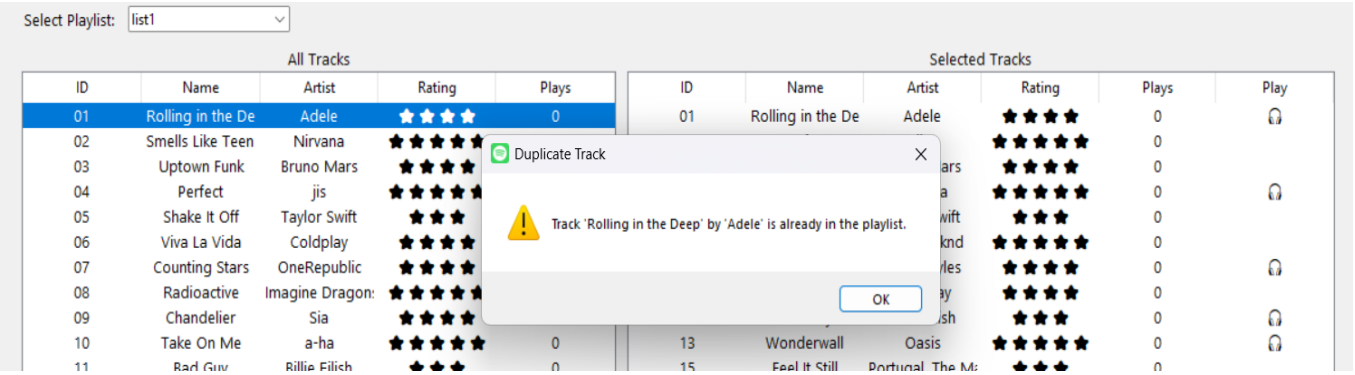


Evidence 3.4: Overly long name

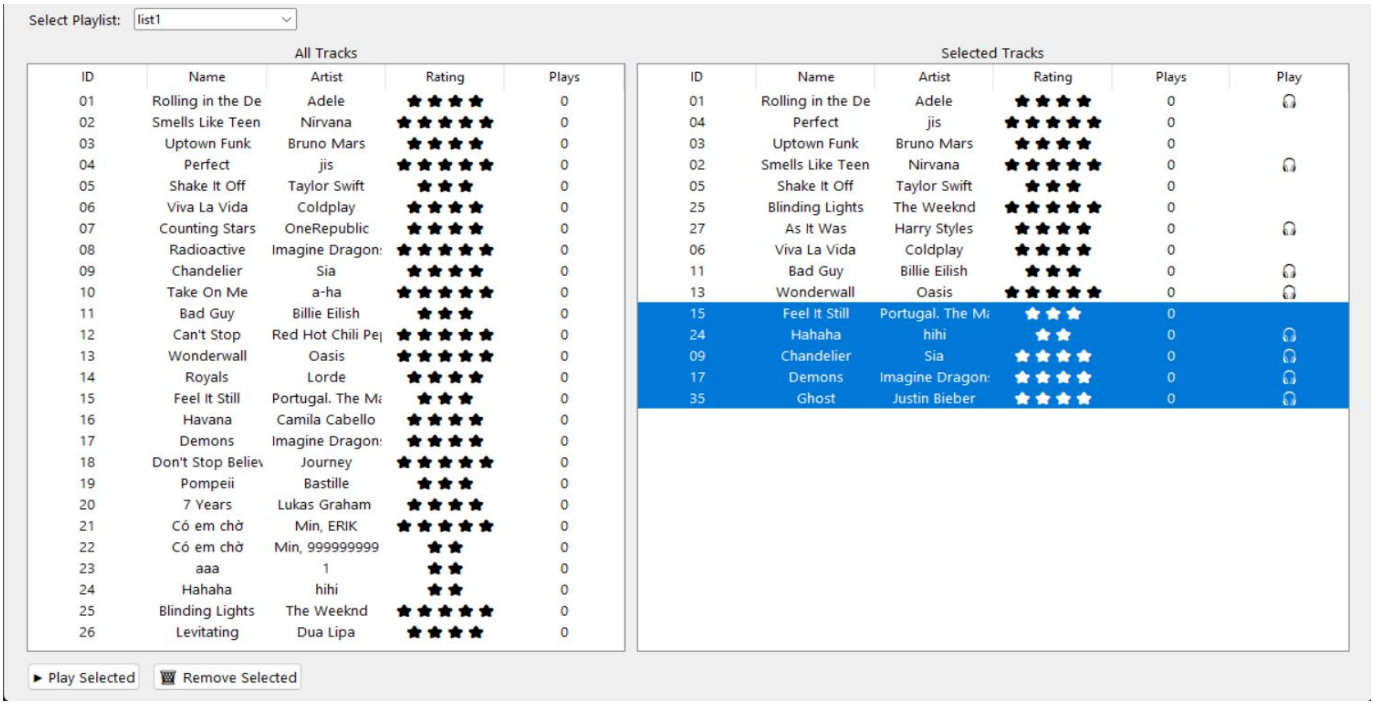


Evidence 3.5: Play selected songs

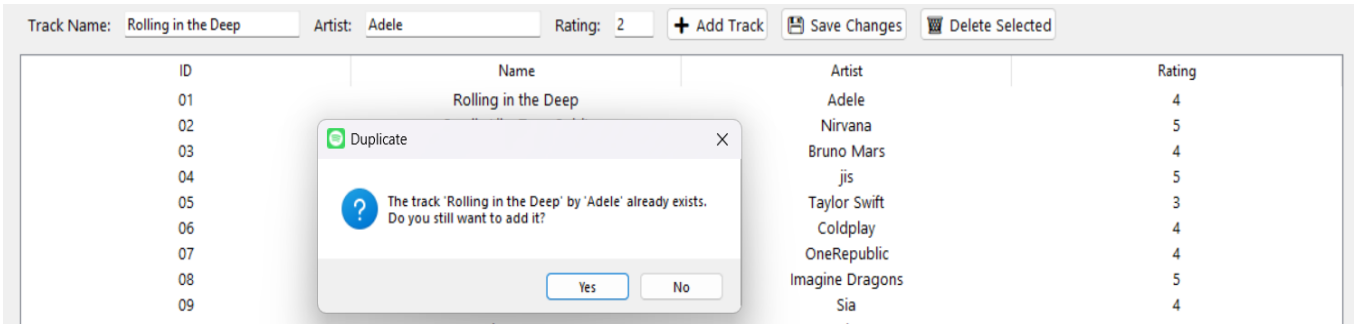
Evidence 3.7: Add the same song to the list

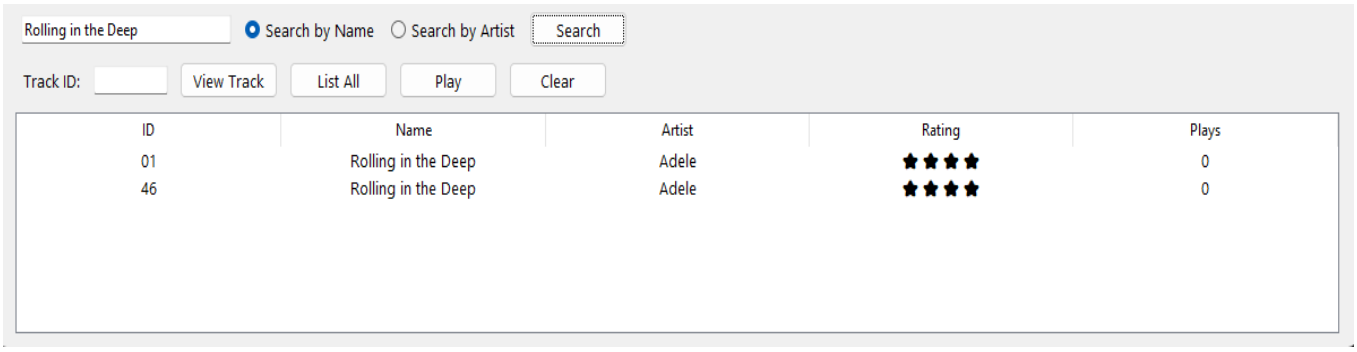


Evidence 3.8: Delete multiple songs in the list

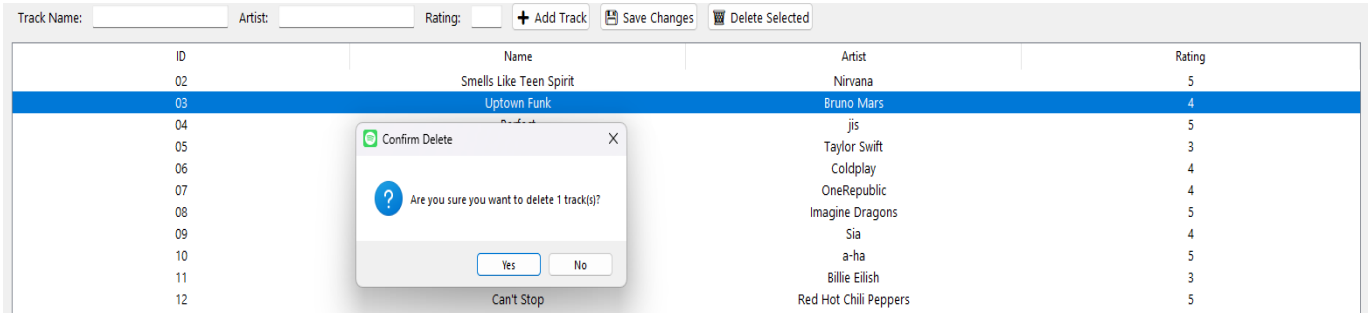


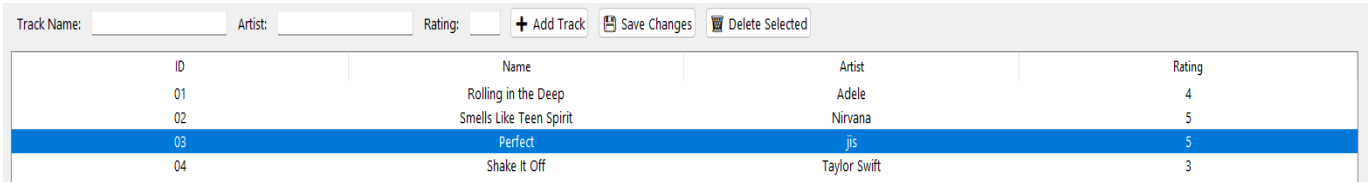
Evidence 4.1: Add duplicate song



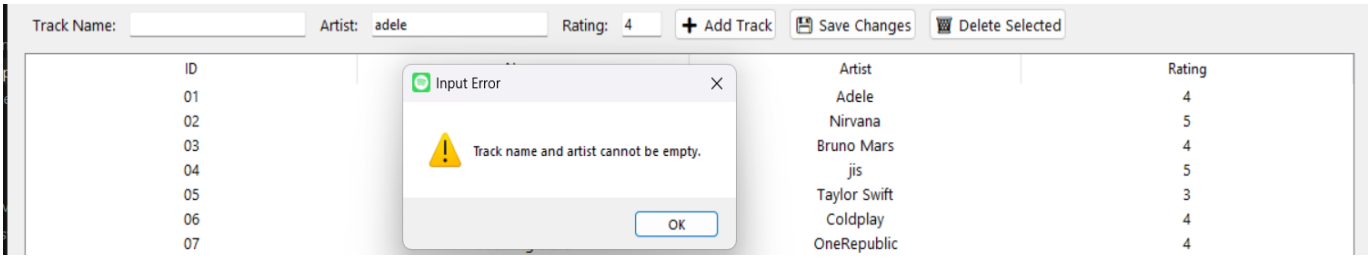


Evidence 4.2: Remove song

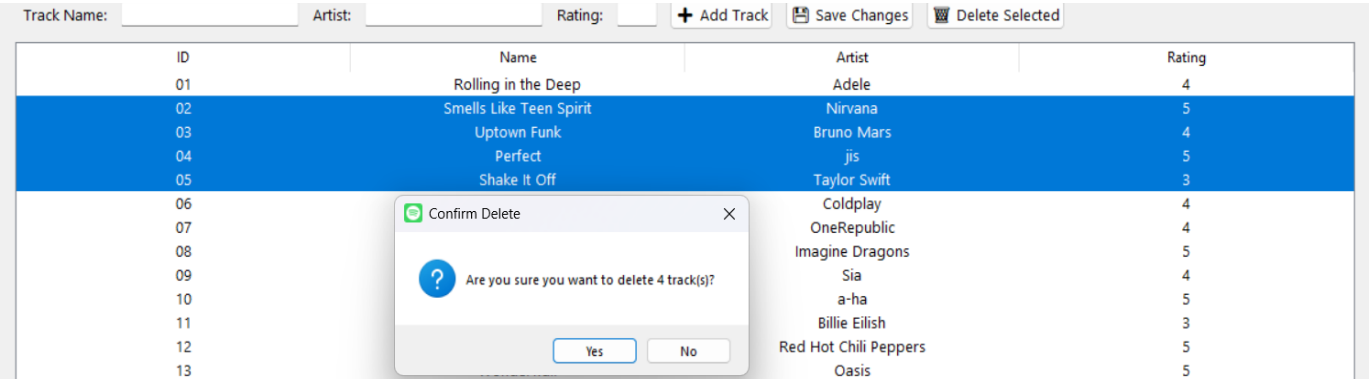


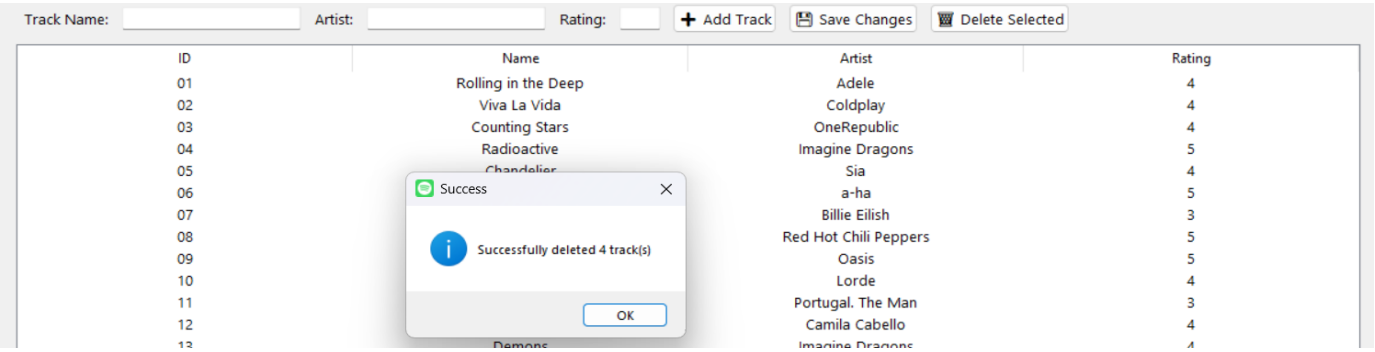


Evidence 4.3: Empty name

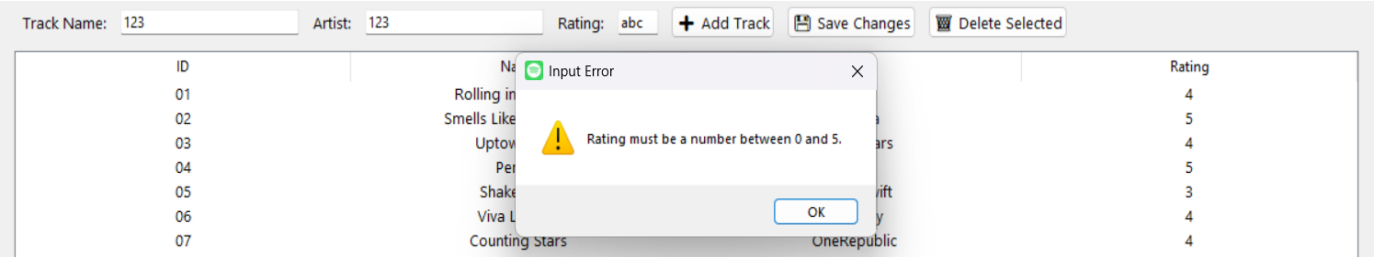


Evidence 4.4: Delete multiple songs

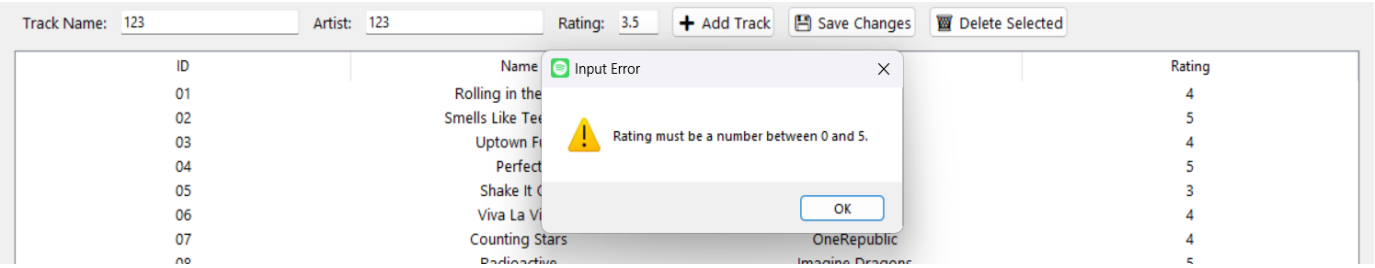




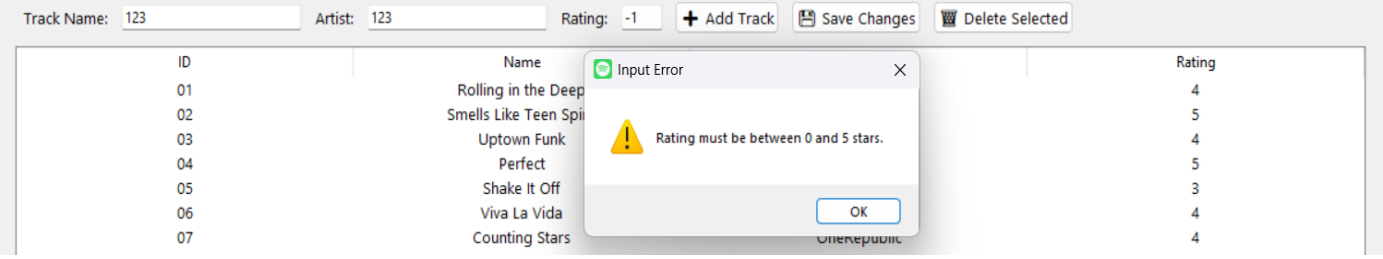
Evidence 5.1: Alphabetic rating



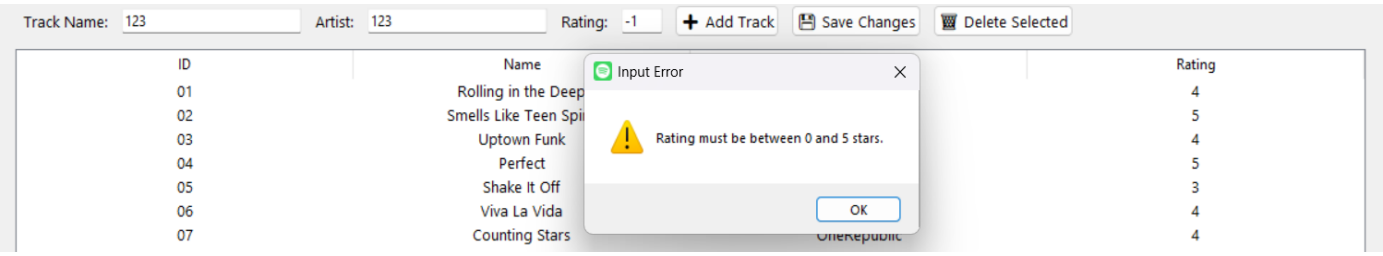
Evidence 5.2: Decimal rating



Evidence 5.3: Negative Rating



Evidence 5.4: Rating > 5



Evidence 5.5: Empty rating