MUSICPLAYER

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology/Master of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

P.Ramya (AP23110011451)

Sk.Nelofer (AP23110011470)

G.Sai Divya(AP23110011467)

P.Mounika(AP23110011431)

Under the Guidance of

DR. KAVITHA RANI KARNENA

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

[November, 2024]

Certificate

Date: 16-Nov-22

This is to certify that the work present in this Project entitled "**MUSIC PLAYER**" has been carried out by **Ramya,Nelofer,Divya,Mounika** under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

(Signature)

Prof. / Dr. [Name]

Designation,

Affiliation.

Co-supervisor

(Signature)

Prof. / Dr. [Name]

Designation,

Affiliation.

Acknowledgements:

This program is a simple implementation of a music player using playlist system having regular as well as premium song. It uses essential Object-Oriented Programming (OOP) principles including:

• **Encapsulation:** Using getters and setters in the Song class we ensure that the song details should be protected and can be accessed safely for modification.
•**Inheritance:** The PremiumSong class inherits from the Song class. It allows us to extend the functionality of plain songs by adding additional properties specific to certain premium songs.
•**Polymorphism:** The displayDetails method in both Song and PremiumSong illustrates runtime polymorphism, where the appropriate method is called depending on whether the song is a regular or premium song.
•**Abstraction:** The program uses classes such as Song, PremiumSong, and Playlist to represent the concept of songs and playlists but this abstraction would make it easier for users to interact with the interface.
•**Dynamic Memory Management:** The method addSong in the Playlist class and the method addSongToPlaylist in the MusicPlayer class dynamically manage memory allocation by adding new songs to the playlist.
The code provides an interface in which users can interact with their playlists, add and remove tracks, search by track name and give ratings, and play all tracks from the active playlist.

Table of Contents

1.Certificate 2

2.Acknowledgements 3

3.Table of Contents 4

4.Abstract 5

5.Introduction 6

6.Methodology 7

7.Discussion 9

8.Concluding Remarks 18

9.Future Work 19

10. References 22

Abstract:

Here is a simulation of the code for a music player application that manages a playlist of songs. Examples of how the principles of Object-Oriented Programming (OOP) are applied would be demonstrating encapsulation, inheritance, and polymorphism in modeling songs and playlists. There are two kinds of songs: regular songs and premium songs, the latter having features not presented in a regular song. The basic modules of the program are

Song Class: A class for representing a song with a title, artist, duration and rating. It includes get and set methods for these attributes as well as a way of printing song information.
Premium Song Class : A subclass for Song - adds an extra feature for premium songs, demonstrating inheritance and method overriding.
class Playlist class Containing a list of songs and methods to add, remove, search, and print out the songs in the playlist. It uses polymorphism by using the vector of Song to store both normal and premium songs.
class MusicPlayer Class Interface for managing a playlist: adding songs, playing all songs, and searching for songs to rate.
The user interacts with the system by a simple text-based menu, which can add songs to a playlist, display all songs, play the songs, search for songs to rate, and remove songs from the playlist. This allows for interactive addition of songs, rating, and display of details regarding the playlist, thus giving a basic and functional music management experience.

This application could be extended in a number of ways-from including more song attributes (e.g., genre) to supporting playlists in different formats (e.g., XML or JSON) or even allowing actual audio files to play back.

INTRODUCTION:

A**"Music Player Application"** is a digital tool used in the support of managing and having fun with collections of music. The growing popularity of digital music platforms has provided for such applications, where users may list their favorite songs in playlists, play their music, and even rate tracks. This application simulates a basic music player, so it will allow users to create playlists, add songs to them, and manage the songs by rating or removing them.

It is based on the **Object-Oriented Programming (OOP)** principles, where the code is organized along the concepts in the real world. The major objects are here **"Song"**, **"PremiumSong"**, **"Playlist"**, and **"MusicPlayer"**. A **"Song"** is quite simple: one sound track - it's having some attributes like title, artist, duration, rating. A **"PremiumSong"** is a fancy version of the song, so while extending the basic attributes of the song with some added functionalities. The Playlist is a collection of songs. It should support addition, removal, and display of songs. MusicPlayer is a central interface from which the user manipulates the playlist adding songs, playing them, and grading tracks.

It makes an application that will manage songs and playlists in a flexible and organized way through the use of OOP concepts such as inheritance -where PremiumSong inherits from Song; **"polymorphism"** -where different types of songs are treated the same; and **"encapsulation"**-that just protects the song details while allowing controlled access-when needed. This structure also enables easy extension of the application in the future by adding new features like support for other formats or improving the user interface.

METHODOLOGY:

The methodology for the design and implementation of the Music Player Application revolves around using **Object-Oriented Programming (OOP)** principles to ensure a modular, reusable, and maintainable system. The application was structured to handle the core functionalities of managing songs, playlists, and the music player, with clear separation of concerns between each component.

First, the core entities of the application were identified. These include the **"Song"** object, which holds information about a song such as its title, artist, duration, and rating. The **"PremiumSong"** class extends the **"Song"** class by adding additional features specific to premium songs, demonstrating the concept of inheritance. The **"Playlist"** class manages a collection of songs and provides methods to add, remove, and search for songs within the playlist. Finally, the **"MusicPlayer"** class is the interface that allows users to interact with the system, enabling them to add songs to the playlist, play all songs, search and rate songs, or remove them.

The design of the application uses **"encapsulation"** to protect the internal data of each class, providing controlled access through getter and setter methods. For example, the **"Song"** class provides methods like **"getTitle()"** and **"getRating()"**,

allowing the internal state of the object to be safely accessed and modified. The **"PremiumSong"** class builds on the **"Song"** class, inheriting its properties and adding new ones, such as an additional feature for premium songs. This use of inheritance allows the code to be more flexible and reduces redundancy.

"Polymorphism" was employed to allow the system to handle both regular and premium songs in a uniform way. For example, the method **"displayDetails()"** is implemented in both the **"Song"** and **"PremiumSong"** classes, allowing the correct method to be called based on the type of song at runtime. This ensures that premium songs can display their additional features, while regular songs display only their basic information.

The **"Playlist"** class manages a collection of songs using a vector of **"Song"** pointers, which allows it to store both regular and premium songs. The vector is dynamically resized as songs are added or removed, and the **"Playlist"** class provides methods for adding, removing, and displaying songs, as well as searching for songs by title or artist.

The **"MusicPlayer"** class acts as the central controller for user interaction. It allows the user to add songs to a playlist, search for and rate songs, play all songs in the current playlist, and remove songs. User input is handled through the standard C++ input/output functions, where song details such as title, artist, duration, and type (regular or premium) are captured via **"cin"** and **"getline()"**. The MusicPlayer then performs the corresponding actions based on the user's choice from the menu.

The program also handles basic error conditions. For instance, when setting a rating, the system checks that the rating is between 0 and 5, ensuring that invalid ratings cannot be applied. Additionally, when searching for a song or removing a song, the system checks whether the song exists in the playlist and provides appropriate feedback to the user.

Memory management is handled through dynamic memory allocation when new songs are added to the playlist. Each song is created using the **"new"** keyword and stored as a pointer in the playlist. In a more complete application, it would be necessary to implement memory deallocation **(e.g., using `delete`)** to avoid memory leaks, though this feature is not included in the current version.

The user interacts with the system through a menu-driven interface. The menu offers choices such as adding songs to the playlist, displaying songs, playing all songs, searching and rating songs, and removing songs. The menu is displayed in a loop, allowing the user to continue interacting with the system until they choose to exit.

In summary, the methodology used to develop the Music Player Application involves identifying key components like songs, playlists, and the music player, and organizing them into classes using OOP principles. This approach allows for easy expansion of functionality and ensures that the code is both maintainable and flexible

DISCUSSIONS:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Song {
private:
    string title;
    string artist;
    double duration;
    int rating;

public:
    Song(string t, string a, double d) : title(t), artist(a), duration(d), rating(0)
    {} string getTitle() const { return title; } string getArtist() const { return
```

```
    artist; } double getDuration() const { return duration; } int getRating()
    const { return rating; } void setRating(int r) {
        if (r >= 0 && r <= 5) {
            rating = r;
        } else { cout << "Invalid rating! Please provide a value between 0 and
            5.\n";
        }
    }
    virtual void displayDetails() const { cout <<
        "Title: " << title << ", Artist: " << artist
            << ", Duration: " << duration << " mins, Rating: " << rating << "/5\n";
    }
};

class PremiumSong : public Song {
private: string
    feature;

public:
    PremiumSong(string t, string a, double d, string f)
        : Song(t, a, d), feature(f) {} void
    displayDetails() const override {
    Song::displayDetails();
        cout << "Additional Feature: " << feature << "\n";
    }
};
```

```
class Playlist {
private:
    string name;
    vector<Song *> songs;

public:
    Playlist(string n) : name(n) {}
    void addSong(Song *song) {
        songs.push_back(song);
     cout << "Song \"" << song->getTitle() << "\" added to playlist \"" << name <<
"\".\n";
    }
    void removeSong(const string &title) { auto it =
        remove_if(songs.begin(), songs.end(), [&title](Song *s) { return
        s->getTitle() == title;
        });
        if (it != songs.end()) {
            cout << "Song \"" << (*it)->getTitle() << "\" removed from playlist \"" <<
name << "\".\n";
            songs.erase(it, songs.end());
        } else { cout << "Song \"" << title << "\" not found in
            playlist.\n";
        }
    }
    Song *searchSong(const string &query) const { for (auto *song :
        songs) { if (song->getTitle() == query || song->getArtist() ==
        query) { return song;
            }
```

```
        }
        return nullptr;
    }
    void displaySongs() const { cout <<
        "\nPlaylist: " << name << "\n";
        if (songs.empty()) { cout << "No songs
            in the playlist.\n";
            return;
        }
        for (auto *song : songs) {
            song->displayDetails();
        }
    }
    const vector<Song *> &getSongs() const { return songs; }
    string getName() const { return name; }
};

class MusicPlayer {
private:
    Playlist *currentPlaylist;

public:
    MusicPlayer() : currentPlaylist(nullptr) {}
    void setPlaylist(Playlist *playlist) {
    currentPlaylist = playlist;
        cout << "Playlist \"" << playlist->getName() << "\" is now active.\n";
    }
    void playAll() const {
```

```
    if (!currentPlaylist) {
        cout << "No playlist set.\n";
        return;
    }
    cout << "Playing all songs in playlist \"" << currentPlaylist->getName() <<
"\":\n"; for (auto *song : currentPlaylist->getSongs()) { cout << "Playing: " <<
    song->getTitle() << " by " << song->getArtist() << "\n";
    }
  }
  void addSongToPlaylist(Playlist &playlist) {
    string title, artist, type;
    double duration;
    cout << "Enter song title: ";
    cin.ignore();
    getline(cin, title);
    cout << "Enter artist name: ";
    getline(cin, artist);
    cout << "Enter duration (in minutes): "; cin >>
    duration; cout << "Enter song type ('regular' or
    'premium'): "; cin >> type;

    if (type == "premium") {
      string feature; cout << "Enter
      additional feature: ";
      cin.ignore(); getline(cin, feature); playlist.addSong(new
      PremiumSong(title, artist, duration, feature));
    } else { playlist.addSong(new Song(title, artist,
      duration));
```

```
        }
    }
    void searchAndRateSong(Playlist &playlist) {
        string title;
        cout << "Enter song title to search and rate: ";
        cin.ignore();
        getline(cin, title);
        Song *song =
        playlist.searchSong(title); if (song) {
        song->displayDetails(); int rating;

            cout << "Enter rating (0 to 5): ";
            cin >> rating; song-
            >setRating(rating);
        } else { cout << "Song \"" << title << "\" not found in
            playlist.\n";
        }
    }
};

int main() {
    MusicPlayer player; Playlist
    myPlaylist("My Playlist"); int
    choice;

    do { cout << "\n** Music Player Menu **\n";
        cout << "1. Add Song to Playlist\n"; cout
        << "2. Display All Songs in Playlist\n";
```

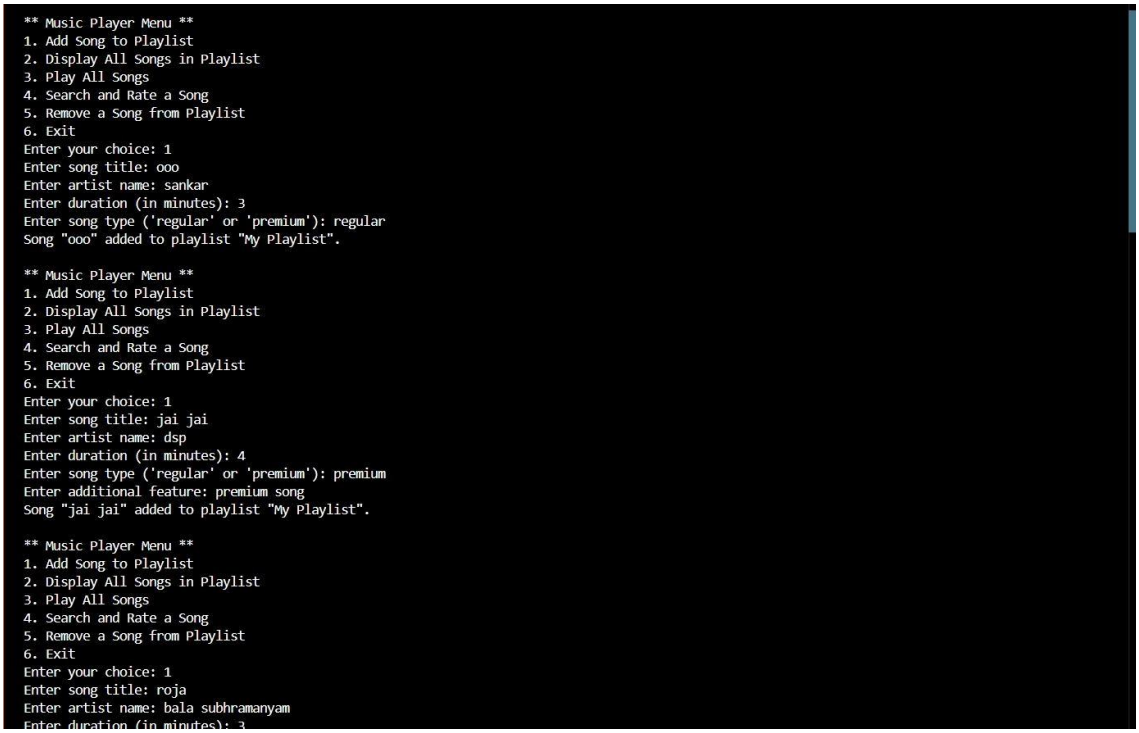
```
cout << "3. Play All Songs\n"; cout << "4.
Search and Rate a Song\n"; cout << "5.
Remove a Song from Playlist\n"; cout <<
"6. Exit\n"; cout << "Enter your choice: ";
cin >> choice;

switch (choice) { case 1:
   player.addSongToPlaylist(myPlaylist);
   break;
   case 2:
   myPlaylist.displaySongs();
   break; case 3:
      player.setPlaylist(&myPlaylist);
   player.playAll(); break; case 4:
   player.searchAndRateSong(myPlaylist);
   break; case 5: {
      string title;
      cout << "Enter the title of the song to remove: ";
      cin.ignore();
      getline(cin, title);
      myPlaylist.removeSong(title);
      break;
   }
   case 6: cout << "Exiting Music Player.
      Goodbye!\n";
      break;
   default:
      cout << "Invalid choice! Please try again.\n";
```

```
    }
  } while (choice != 6);

  return 0;
}
```

output:



```
Enter song title: roja
Enter artist name: bala subhramanyam
Enter duration (in minutes): 3
Enter song type ('regular' or 'premium'): regular
Song "roja" added to playlist "My Playlist".

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
Enter your choice: 2

Playlist: My Playlist
Title: ooo, Artist: sankar, Duration: 3 mins, Rating: 0/5
Title: jai jai, Artist: dsp, Duration: 4 mins, Rating: 0/5
Additional Feature: premium song
Title: roja, Artist: bala subhramanyam, Duration: 3 mins, Rating: 0/5

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
Enter your choice: 4
Enter song title to search and rate: ooo
Title: ooo, Artist: sankar, Duration: 3 mins, Rating: 0/5
Enter rating (0 to 5): 4

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
Enter your choice: 4
```

```
Enter your choice: 4
Enter song title to search and rate: jai jai
Title: jai jai, Artist: dsp, Duration: 4 mins, Rating: 0/5
Additional Feature: premium song
Enter rating (0 to 5): 3

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
Enter your choice: 2

Playlist: My Playlist
Title: ooo, Artist: sankar, Duration: 3 mins, Rating: 4/5
Title: jai jai, Artist: dsp, Duration: 4 mins, Rating: 3/5
Additional Feature: premium song
Title: roja, Artist: bala subhramanyam, Duration: 3 mins, Rating: 0/5

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
Enter your choice: 5
Enter the title of the song to remove: roja
Song "roja" removed from playlist "My Playlist".

** Music Player Menu **
1. Add Song to Playlist
2. Display All Songs in Playlist
3. Play All Songs
4. Search and Rate a Song
5. Remove a Song from Playlist
6. Exit
```

CONCLUDING REMARKS:

The code you've written is a solid, straightforward implementation of a basic music player. It uses object-oriented programming concepts like inheritance, polymorphism, and encapsulation really well. For example, the `Song` class is the foundation, holding the essential details like the song title, artist, duration, and rating. The `PremiumSong` class builds on this by adding an extra feature and customizing how song details are displayed, which shows how inheritance and method overriding work together.

The **"Playlist"** class stores a list of songs, both regular and premium, and the **"MusicPlayer"** class lets the user interact with the playlist. You can add songs, remove them, search for specific songs, rate them, and play the entire playlist. The user interface is simple and clear, guiding users through each action with easy-to-follow prompts.

That said, there are a couple of areas where the code could be improved. One important thing to consider is memory management. The code uses raw pointers to store songs in the playlist, which means there's a risk of memory leaks if songs aren't properly deleted. Switching to smart pointers like **"std::unique_ptr"** would handle this automatically and make the code safer. Also, adding more input validation would improve the user experience. For example, you could check if users enter a valid song duration or ensure that the song title isn't empty before accepting it.

You could also add extra features, like the ability to shuffle the playlist, save and load playlists from a file, or even store additional details about the songs, like their album or genre. Overall, though, the code does a great job of demonstrating how object-oriented principles can be applied in a practical, real-world scenario. It's a

good foundation that can be expanded with more features and improvements over time.

FUTURE WORK:

For future work, there are several enhancements and features that could be added to improve the functionality and user experience of the music player application. Here are some suggestions:

1. "Memory Management Improvements":

- As mentioned earlier, the current implementation uses raw pointers for storing songs in the playlist, which can lead to memory leaks. Moving to **"smart pointers" (such as `std::unique_ptr` or `std::shared_ptr`)** would automatically manage memory, ensuring proper cleanup of resources when songs are removed or when the program ends.

2. "Persisting Playlists":

- The ability to "**save and load playlists"** would be a significant improvement. Users could save their playlists to a file **(e.g., in JSON or XML format)** and load them back into the application later. This would allow users to maintain their playlists across sessions and share them with others.

3. "Enhanced Input Validation":

- The current program could benefit from more **"robust input validation"**. For example, checking that the song title is not empty, ensuring that the song duration is a positive number, and validating that ratings are within the acceptable range would improve the user experience. Additionally, the program could provide more user-friendly error messages when invalid inputs are detected.

4. "Playlist Management":

- More advanced playlist management features could be added, such as the ability to **"shuffle"** the playlist, **"sort"** songs by different criteria (e.g., title, artist, rating, or duration), and **"create multiple playlists".** Users could have the option to create, modify, and switch between different playlists.

5. “User Preferences”:

- The program could allow users to customize their experience. For example, users could set **“personalized settings”** like default sorting options for songs, playback preferences, or even theme options for the interface (if the application is expanded to have a GUI).

6. “Song Search and Filtering”:

- Currently, the program only allows searching by title or artist. Adding the ability to **“filter songs by additional attributes”** such as rating, duration, or even song genre, would make it easier for users to find and organize their music.

7. “Improved Playback Features”:

- While the application currently "plays" all songs in a playlist, adding more sophisticated playback features would enhance the experience. Features such as **“pause, skip, repeat”,** and the ability to display song progress or time elapsed would make the music player feel more interactive. Integration with real audio files **(e.g., MP3 or WAV)** could be explored to allow for actual playback.

8. “User Ratings and Reviews”:

- In addition to just rating songs, users could be allowed to **“write reviews”** or add comments to songs. This would help create a more interactive and personalized music experience, and could even integrate with an online database of songs to share user-generated content.

9. “Integration with External APIs”:

- The application could be expanded to integrate with **“external music APIs” (e.g., Spotify, Apple Music, or Last.fm)** to automatically fetch song information like album art, track length, and genre. This would make the application feel more modern and connected to real-world music services.

10. “Graphical User Interface (GUI)”:

- While the current implementation is text-based, adding a **GUI** (using libraries like Qt or SFML) could greatly enhance user interaction. A graphical

interface would allow users to drag and drop songs into playlists, view song details more clearly, and interact with the music player in a more intuitive way.

11. "Cross-Platform Support":

- If the application is developed further, ensuring it is **cross-platform** (e.g., working on Windows, macOS, and Linux) would broaden its reach and make it more accessible. Using a framework like Qt or a web-based application could help achieve this goal.

12. "Mobile Application":

- Taking this concept to mobile devices (iOS/Android) could also be an interesting direction. Users would be able to manage their music playlists on-the-go, and the app could take advantage of mobile features like offline music storage and notifications.

These are just a few of the directions in which this music player application could evolve. By incorporating these ideas, the program could grow into a fully-featured, user-friendly music management tool, potentially evolving into something more than just a basic music player—creating a richer and more personalized experience for the user.

References:

1. Thinking in C++, Bruce, Eckel, Pearson, Second edition, Volume 1, 2002.

2. Object-oriented programming in C++, Robert Lafore, Course Sams Publishing, Fourth edition,2001.

3. Lischner, Ray. STL Pocket Reference: Containers, Iterators, and Algorithms. " O'Reilly Media,Inc.", 2003