INTERNATIONAL ISLAMIC UNIVERSITY ISLAMABAD

COURSE:-

Data Structure and Algorithm (DS&A)

Submitted By:

Reg no:- 4747, 4763, 4814, 4765-FOC/BSSE/F23 **(A)**

Submitted To:

Name:- Sir Shakeel Ahmed

Course Code:- CS 214

Department Of Software Engineering

Faculty of Computing and information technology

Music Player Manager

---

1. Introduction

1.1 Purpose

This SRS defines the requirements for the **Music Player Manager**, a C++-based application designed to allow users to manage, play, and organize their music collection. The system enables functionalities such as adding, deleting, and editing song details, playing songs, creating playlists, and searching tracks.

1.2 Scope

The Music Player Manager will be a standalone desktop application developed in C++. The system will:

- Support song storage and metadata management.
- Allow playback (simulated).
- Offer playlist creation and management.
- Provide a search functionality by song name, artist, or genre.
- Display song lists and playlists.
- Operate via a Command Line Interface (CLI).

1.3 Definitions, Acronyms, and Abbreviations

- **CLI**: Command Line Interface
- **ID3 Tags**: Metadata container used in MP3 files
- **SRS**: Software Requirements Specification

1.4 References

- IEEE SRS Standards
- C++ documentation (ISO C++17/20)

1.5 Overview

This document provides detailed functional and non-functional requirements, system features, user interfaces, and design constraints for the Music Player Manager.

---

2. Overall Description

2.1 Product Perspective

This is an independent desktop software that will interact with the file system to store and retrieve music data (but not actual playback of music files).

2.2 Product Functions

- Add/Edit/Delete Songs
- Display All Songs

- Create/Delete Playlists
- Add Songs to Playlist
- Remove Songs from Playlist
- Search by Song Name/Artist/Genre
- Simulated Play Song Function
- Save and Load data from files

2.3 User Classes and Characteristics

- **General User**: Can manage their songs and playlists.
- **Developer/Maintainer**: Can modify the codebase or update song data formats.

2.4 Operating Environment

- OS: Windows/Linux
- Compiler: g++, MinGW, or any standard C++ compiler
- Interface: Command Line

2.5 Design and Implementation Constraints

- Written in C++
- No GUI or multimedia library dependencies
- Use of standard file I/O for persistence
- Simulated song playing (e.g., displaying now playing)

2.6 Assumptions and Dependencies

- User has access to the file system.
- Songs are managed as data entries, not audio files.

---

3. Specific Requirements

3.1 Functional Requirements

FR1: Add New Song

- Input: Song title, artist, album, genre, duration
- Output: Song is stored in the list
- Trigger: User selects "Add Song"

-
  ```
  // Modified Song class with all fields
  class Song {
  public:
      int id;
      string title;
      string artist;
      string album;
      string genre;
      int duration;

      Song(int i=0, string t="", string a="", string al="", string g="",
  int d=0)
          : id(i), title(t), artist(a), album(al), genre(g), duration(d)
  {}
  };
  ```

```
// In main() menu (case 1)
case 1: {
    static int nextId = 1;  // Auto-generate IDs
    string t, a, al, g;
    int d;
    cout << "Title: ";
    getline(cin, t);
    cout << "Artist: ";
    getline(cin, a);
    cout << "Album: ";
    getline(cin, al);
    cout << "Genre: ";
    getline(cin, g);
    cout << "Duration (s): ";
    cin >> d;
    playlist.addSong(Song(nextId++, t, a, al, g, d));
    break;
}
```

FR2: Display Song List

- Input: Command
- Output: List of all songs with details
- Trigger: User selects "Display Songs"

```
// Modified Playlist::display()
void display() {
    DNode* current = head;
    while (current) {
        cout << "ID: " << current->data.id
             << "\nTitle: " << current->data.title
             << "\nArtist: " << current->data.artist
             << "\nAlbum: " << current->data.album
             << "\nGenre: " << current->data.genre
             << "\nDuration: " << current->data.duration << "s\n\n";
        current = current->next;
    }
}
```

FR3: Edit Song

- Input: Song ID, new metadata
- Output: Song details updated
- Trigger: User selects "Edit Song"

```
// In Playlist class
bool editSong(int id, const Song& newData) {
    DNode* current = head;
    while (current) {
        if (current->data.id == id) {
            current->data = newData;
            return true;
        }
        current = current->next;
```

```cpp
        }
        return false;
    }

    // New menu option
    case 15: {
        int id;
        cout << "Song ID to edit: ";
        cin >> id;
        cin.ignore();

        string t, a, al, g;
        int d;
        cout << "New Title: ";
        getline(cin, t);
        cout << "New Artist: ";
        getline(cin, a);
        cout << "New Album: ";
        getline(cin, al);
        cout << "New Genre: ";
        getline(cin, g);
        cout << "New Duration (s): ";
        cin >> d;

        if (playlist.editSong(id, Song(id, t, a, al, g, d))) {
            cout << "Song updated!\n";
        } else {
            cout << "Song not found!\n";
        }
        break;
    }
```

FR4: Delete Song

- Input: Song ID
- Output: Song is removed
- Trigger: User selects "Delete Song"

```cpp
    // Modified Playlist::removeSong()
    bool removeSong(int id) {
        DNode* current = head;
        while (current) {
            if (current->data.id == id) {
                if (current->prev) current->prev->next = current->next;
                else head = current->next;
                if (current->next) current->next->prev = current->prev;
                else tail = current->prev;
                delete current;
                size--;
                return true;
            }
            current = current->next;
        }
        return false;
    }
```

```
// Updated menu case 2
case 2: {
    int id;
    cout << "Song ID to remove: ";
    cin >> id;
    if (playlist.removeSong(id)) cout << "Song removed.\n";
    else cout << "Song not found.\n";
    break;
}
```

FR5: Create Playlist

- Input: Playlist name
- Output: Playlist created
- Trigger: User selects "Create Playlist"

```
// PlaylistManager class
class PlaylistManager {
private:
    map<string, Playlist> playlists;

public:
    void createPlaylist(const string& name) {
        playlists[name] = Playlist();
    }
};

// New menu option
case 16: {
    string name;
    cout << "Playlist name: ";
    getline(cin, name);
    playlistManager.createPlaylist(name);
    cout << "Playlist created!\n";
    break;
}
```

FR6: Add Song to Playlist

- Input: Playlist name, Song ID
- Output: Song added to playlist
- Trigger: User selects "Add to Playlist"

```
// In PlaylistManager class
bool addToPlaylist(const string& playlistName, int songId, Playlist&
mainPlaylist) {
    if (playlists.find(playlistName) == playlists.end()) return false;

    DNode* current = mainPlaylist.getHead();
    while (current) {
        if (current->data.id == songId) {
            playlists[playlistName].addSong(current->data);
            return true;
        }
        current = current->next;
    }
    return false;
}

// New menu option
case 17: {
    string plName;
    int songId;
    cout << "Playlist name: ";
    getline(cin, plName);
    cout << "Song ID to add: ";
    cin >> songId;

    if (playlistManager.addToPlaylist(plName, songId, playlist)) {
        cout << "Song added to playlist!\n";
    } else {
        cout << "Failed to add song!\n";
    }
    break;
}
```

FR7: Search Song

- Input: Search keyword
- Output: Matching songs
- Trigger: User selects "Search"

```
    // Enhanced search function
    void searchSongs(Playlist& p, string keyword) {
        DNode* current = p.getHead();
        bool found = false;

        while (current) {
            if (current->data.title.find(keyword) != string::npos ||
                current->data.artist.find(keyword) != string::npos ||
                current->data.album.find(keyword) != string::npos ||
                current->data.genre.find(keyword) != string::npos) {
                cout << "ID: " << current->data.id << " | Title: " <<
    current->data.title << "\n";
                found = true;
                }
            current = current->next;
        }
        if (!found) cout << "No matches found!\n";
    }

    // New menu option
    case 18: {
        string keyword;
        cout << "Search keyword: ";
        getline(cin, keyword);
        searchSongs(playlist, keyword);
        break;
    }
```

FR8: Play Song

- Input: Song ID
- Output: "Now Playing: <Song Title>"
- Trigger: User selects "Play Song"

```
// Modified play functionality
case 4: {
    int songId;
    cout << "Song ID to play: ";
    cin >> songId;

    DNode* current = playlist.getHead();
    while (current) {
        if (current->data.id == songId) {
            playQueue.addFront(current->data);
            Song s = playQueue.removeFront();
            recent.push(s);
            cout << "Now Playing: " << s.title << "\n";
            break;
        }
        current = current->next;
    }
    break;
}
```

FR9: Save/Load Data

- Input: Auto-triggered or manual
- Output: Songs/playlists saved to file and loaded on restart

```
// Save function
void saveData(Playlist& p, const string& filename = "data.txt") {
    ofstream file(filename);
    DNode* current = p.getHead();
    while (current) {
        file << current->data.id << ","
             << current->data.title << ","
             << current->data.artist << ","
             << current->data.album << ","
             << current->data.genre << ","
             << current->data.duration << "\n";
        current = current->next;
    }
}

// Load function
void loadData(Playlist& p, const string& filename = "data.txt") {
    ifstream file(filename);
    string line;
    while (getline(file, line)) {
        stringstream ss(line);
        string token;

        Song s;
        getline(ss, token, ','); s.id = stoi(token);
        getline(ss, s.title, ',');
        getline(ss, s.artist, ',');
        getline(ss, s.album, ',');
        getline(ss, s.genre, ',');
        getline(ss, token, ','); s.duration = stoi(token);

        p.addSong(s);
    }
}

// In main() startup:
loadData(playlist);

// New save menu option
case 19: {
    saveData(playlist);
    cout << "Data saved!\n";
    break;
}
```

---

4. External Interface Requirements

4.1 User Interfaces

- Text-based command-line menu with options for all features.

4.2 Hardware Interfaces

- Basic disk I/O supported by OS.

4.3 Software Interfaces

- File system for storing song and playlist data.

---

5. Non-functional Requirements

5.1 Performance Requirements

- Instant response for actions (within 1 second)
- Handle up to 1000 songs without lag

5.2 Safety Requirements

- Proper file handling to prevent data loss

5.3 Security Requirements

- No security implementation needed for standalone desktop use

5.4 Software Quality Attributes

- **Usability**: Easy CLI navigation
- **Maintainability**: Modular C++ code structure
- **Portability**: Compatible with major OS environments

6. Appendices

A. Data Structures (example)

cpp

CopyEdit

```
struct Song {
    int id;
    std::string title;
    std::string artist;
    std::string album;
    std::string genre;
    float duration;
};
```

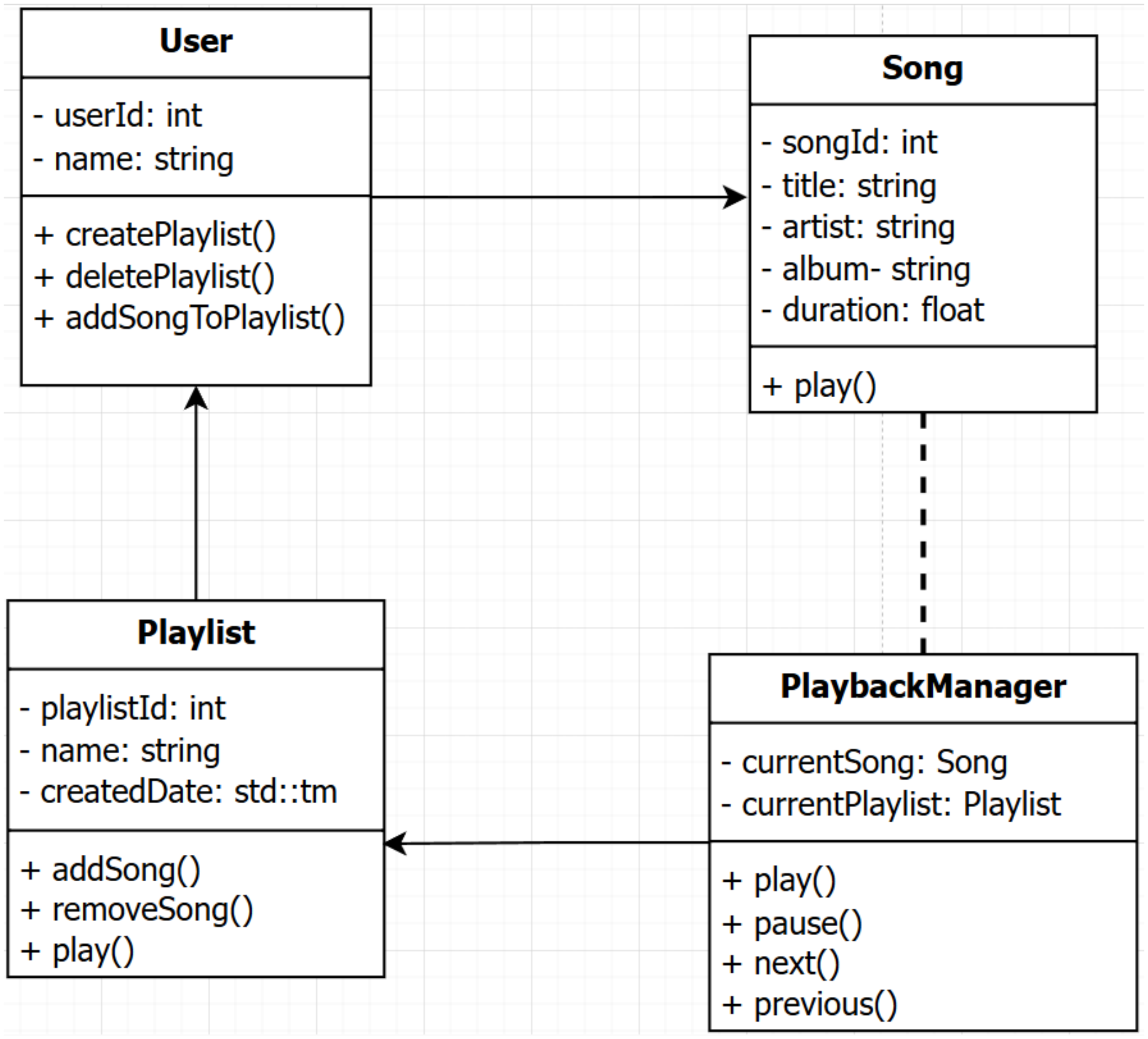
B. Sample Main Menu

markdown

CopyEdit

```
1. Add Song
2. Edit Song
3. Delete Song
4. View All Songs
5. Search Songs
6. Create Playlist
7. Add Song to Playlist
8. View Playlist
9. Play Song
10. Save & Exit
```

UML Class Diagram:


User
- userId: int
- name: string
+ createPlaylist()
+ deletePlaylist()
+ addSongToPlaylist()
Song
- songId: int
- title: string
- artist: string
- album- string
- duration: float
+ play()
Playlist
- playlistId: int
- name: string
- createdDate: std::tm
+ addSong()
+ removeSong()
+ play()
PlaybackManager
- currentSong: Song
- currentPlaylist: Playlist
+ play()
+ pause()
+ next()
+ previous()