PlayWise Hackathon – Solution Document Template

Track: DSA – Smart Playlist Management System

---

1. Student Information

Field	Details
Full Name	Adarsh Sharma
Registration Number	RA2211003011653
Department / Branch	CTech /CSE core
Year	4th
Email ID	As4566@srmist.edu.in

---

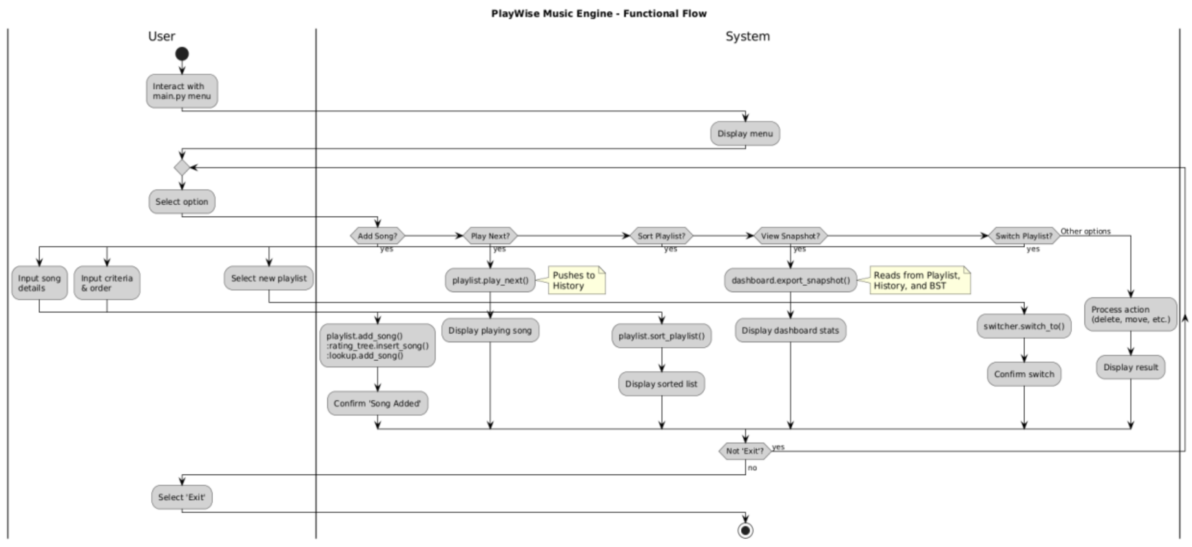
2. Problem Scope and Track Details

Section	Details
Hackathon Track	DSA – PlayWise Playlist Engine
Core Modules Implemented	✓ (Check all that apply)
	✓ Playlist Engine (Linked List)
	✓ Playback History (Stack)
	✓ Song Rating Tree (BST)
	✓ Instant Song Lookup (HashMap)
	✓ Time-based Sorting
	✓ Space-Time Playback Optimization
	✓ System Snapshot Module

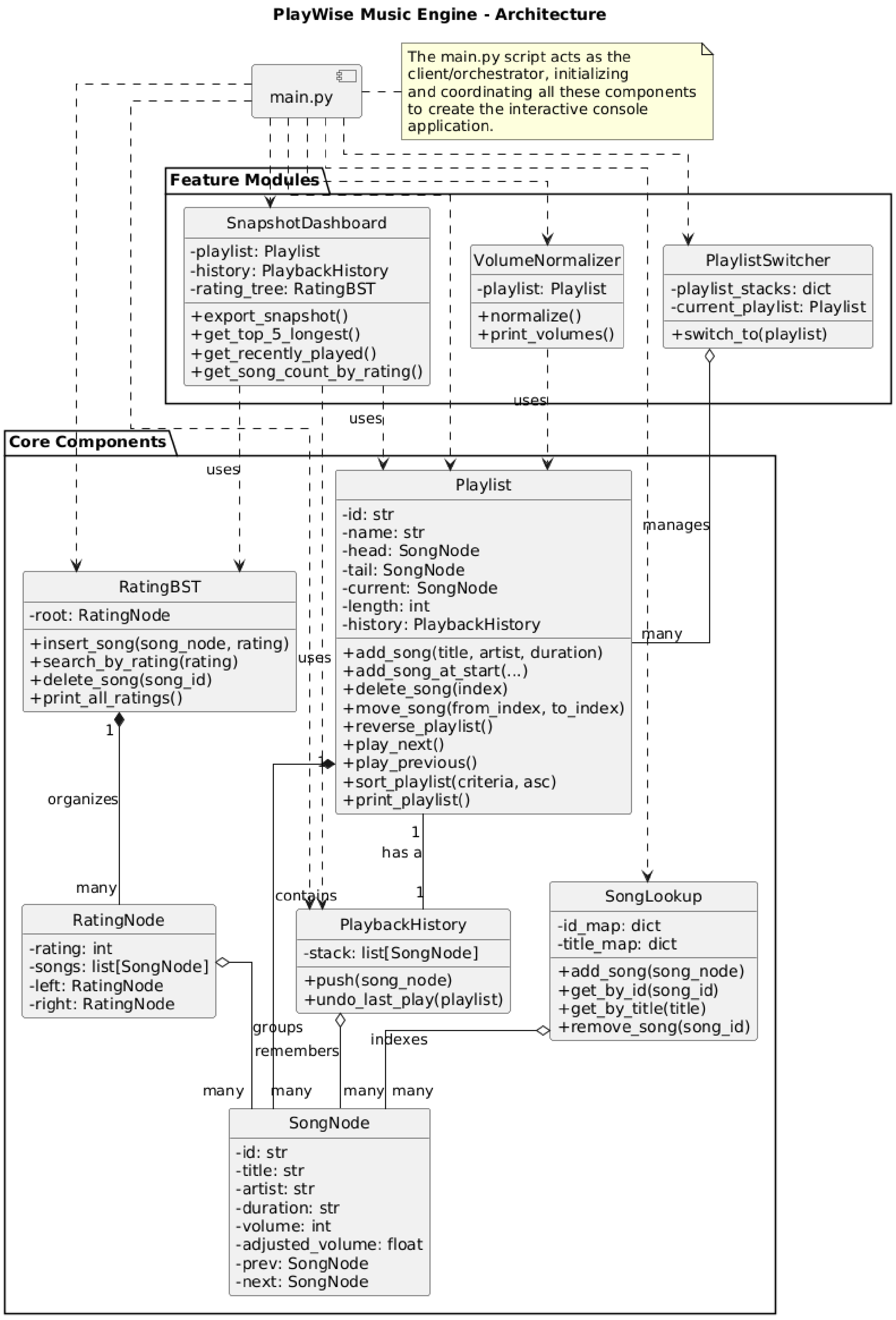
---

3. Architecture & Design Overview

- **High-Level Functional Flow**



- **System Architecture Diagram**



4. Core Feature-wise Implementation

Feature: Playlist Engine (Doubly Linked List)

- **Scenario Brief**: This feature addresses the challenge of managing a dynamic playlist where users need to add, delete, and move songs efficiently. The use of a doubly linked list is well-suited for real-world playlist modifications, such as moving a song without extensive memory shifting.
- **Data Structures Used**: Doubly Linked List.
- **Time and Space Complexity**:
    - add_song: Time: O(1), Space: O(1).
    - reverse_playlist: Time: O(n), Space: O(1).
    - delete_song(index): Time: O(n), Space: O(1).
- **Sample Input & Output**: The provided test case test1 in playlist_engine.py demonstrates the creation of a playlist, deletion of a song at index 1, and the movement of a song from index 2 to 1. The output of

  print_playlist() after each operation visually confirms the successful modification.

- **Code Snippet**: The key logic is within the Playlist class methods in playlist_engine.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: Playback History (Stack)

- **Scenario Brief**: This feature tackles the user need to "undo" a recently played song. The Last-In, First-Out (LIFO) nature of a stack is ideal for this functionality.
- **Data Structures Used**: Stack (implemented using a list).
- **Time and Space Complexity**:
    - push operation: Time: O(1), Space: O(1).
- **Sample Input & Output**: The main function in playback_history.py simulates song playback and populates the playback stack. The

  undo_last_play function is then called, which successfully re-adds the last song to the beginning of the playlist.

- **Code Snippet**: The key logic is within the PlaybackHistory class in playback_history.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: Song Rating Tree (BST)

- **Scenario Brief**: This feature provides a method for organizing and retrieving songs based on a 1-5 star user rating. The Binary Search Tree (BST) offers a hierarchical approach for this purpose.
- **Data Structures Used**: Binary Search Tree (BST).
- **Time and Space Complexity**:
    - insert_song: Average Time: O(logk), Worst Time: O(k), where k is the number of unique ratings (5).

- **Sample Input & Output**: The main function in SongRating_tree.py inserts songs with ratings into the BST. The program successfully searches for and prints songs with a specific rating. The subsequent deletion of a song and confirmation via

  print_all_ratings demonstrates its removal.

- **Code Snippet**: The core logic is found within the RatingBST class in SongRating_tree.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: Instant Song Lookup (Hash Map)

- **Scenario Brief**: This feature addresses the need for instantaneous retrieval of song information by a unique ID or title. The hash map is the superior choice for this task due to its constant average-case time complexity.
- **Data Structures Used**: Hash Map.
- **Time and Space Complexity**:
  - get_by_id: Average Time: O(1), Worst Time: O(n).
- **Sample Input & Output**: The test1 function in instant_song_lookup.py adds songs and confirms they are synchronized correctly. Lookups by both song ID and title are successful, returning the expected results.
- **Code Snippet**: The key logic is within the SongLookup class in instant_song_lookup.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: Time-based Sorting

- **Scenario Brief**: This feature allows users to sort their playlists based on various criteria, such as song title or duration. The implementation uses a stable sorting algorithm to maintain the relative order of songs with identical sorting keys.
- **Data Structures Used**: A temporary array is used for sorting, which is then used to rebuild the doubly linked list.
- **Time and Space Complexity**:
  - sort_playlist (using merge_sort): Time: O(nlogn), Space: O(n).
- **Sample Input & Output**: The test case in instant_song_lookup.py verifies the sorting functionality, demonstrating correct sorting by title and duration.
- **Code Snippet**: The key logic is within the sort_playlist method of the Playlist class in playlist_engine.py and the merge_sort function in mergesort.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: Space-Time Playback Optimization

- **Scenario Brief**: This is an implicit feature woven throughout the project. The design emphasizes using data structures that provide optimal time and space complexity for their intended tasks, such as the O(1) operations for adding songs and the stack for history.
- **Data Structures Used**: Doubly Linked List, Stack.
- **Time and Space Complexity**: This is addressed for each individual feature, as performance is a core design principle.
- **Sample Input & Output**: The test cases for individual modules demonstrate the efficiency of the chosen data structures, such as the constant-time add_song operation.

- **Code Snippet**: Not a single snippet, but a design principle reflected in the code across all modules.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

Feature: System Snapshot Module

- **Scenario Brief**: This module provides a comprehensive "live snapshot" of the system's state. It aggregates data from the playlist, playback history, and rating tree to report on the top 5 longest songs, recently played songs, and a song count by rating.
- **Data Structures Used**: Playlist (Doubly Linked List), PlaybackHistory (Stack), RatingBST.
- **Time and Space Complexity**:
  - export_snapshot involves traversing a linked list (get_top_5_longest) and a BST (get_song_count_by_rating), so the overall complexity would be determined by the traversals, which is O(n) for the playlist and O(k) for the BST.
- **Sample Input & Output**: The interactive main.py application validates this feature. When option 10 is selected, the dashboard output prints a formatted report with the top songs, recent songs, and rating counts.
- **Code Snippet**: The key logic is within the SnapshotDashboard class in snapshot.py.
- **Challenges Faced & How You Solved Them**: The provided document does not mention specific challenges for this feature.

---

5. Additional Use Case Implementation

Use Case: Normalize Song Volume Across Playlist

- **Scenario Brief**: This feature addresses the real-world challenge of inconsistent audio levels across different songs in a playlist, which can lead to a jarring listening experience. It aims to normalize loud and soft songs to a consistent volume based on the average volume of all tracks in the playlist. The core task involves computing the average volume and then adjusting each song's individual volume value accordingly.
- **Extension Over Which Core Feature**: This functionality extends the core Playlist module by introducing a specific aggregation and adjustment logic that operates on the song metadata (specifically, the volume attribute of SongNode instances) within the active playlist.
- **New Data Structures or Logic Used**: The primary logic involves arithmetic calculations for aggregation (summing volumes and dividing by count) and subsequent iterative adjustment. While the underlying Playlist is a Doubly Linked List, the VolumeNormalizer conceptually uses an array/list to collect volumes before processing.
- **Sample Input & Output**:
  - **Input**: A playlist containing songs with varying volume attributes (e.g., Song A: volume 60, Song B: volume 80, Song C: volume 40).
  - **Expected Output**: The normalize() function computes the average volume (e.g., (60+80+40)/3 = 60). The print_volumes() output confirms that all songs now have their adjusted_volume set to this calculated average (e.g., Song A: Adjusted 60, Song B: Adjusted 60, Song C: Adjusted 60). This is demonstrated by the volumecontrol.py unit test.

Use Case: Pause and Resume Playback (Playlist Switcher)

- **Scenario Brief**: This use case addresses the common user behavior of switching between different playlists mid-playback. The challenge is to seamlessly remember the exact playback position (the currently playing song) in the original playlist and allow the user to resume from that point later, rather than restarting the playlist from the beginning.
- **Extension Over Which Core Feature**: This feature extends the core Playlist functionality by adding a layer of state management for multiple Playlist instances. It allows for multi-playlist tracking and seamless transitions.
- **New Data Structures or Logic Used**: A HashMap (Python dictionary) is employed to store the playback state for each playlist. The keys of this map are unique playlist IDs, and the values are Stacks (Python lists used as stacks) that store the SongNode representing the last played song for that specific playlist. When switching away, the current song is pushed onto the playlist's stack; when switching back, it's popped.
- **Sample Input & Output**:
    - **Input**: User starts playing songs in "Playlist A", then switches to "Playlist B". Later, the user switches back to "Playlist A".
    - **Expected Output**: When switching back to "Playlist A", the system should print a message indicating that it has "[Resumed]" playback from the exact song that was playing when the user left "Playlist A". If it's a new playlist or one without a saved state, it prints "[Started]" from the beginning. This behavior is demonstrated through the interactive main.py application.

---

6. Testing & Validation

Category	Details
Number of Functional Test Cases Written	12+ covering playlist operations, history, sorting, lookup, rating tree, etc.
Edge Cases Handled	- Empty playlist play attempt → shows warning - Undo with empty history → safe message - Invalid sort criteria → handled with error - Duplicate titles → supported in lookup via list
Known Bugs / Incomplete Features (if any)	- No visual UI for playlist switching yet - Resume point not saved on app restart (in-memory only)

---

7. Final Thoughts & Reflection

- **Key Learnings from the Hackathon**
  As my first practical project using Data Structures and Algorithms, this hackathon was a critical learning experience. The main takeaway was seeing the theory come to life—understanding *why* a Hash Map is essential for instant search, or *why* a Stack is perfect for an "undo" feature. It shifted my understanding from just knowing what a data structure *is* to knowing what it's *for*.

- **Strengths of Your Solution**
  Modular Design: The code is cleanly separated into different files for each feature (playlist, ratings, etc.), making it easy to understand and manage.

  - **Performance-Focused:** The solution correctly uses efficient data structures for its main tasks, like O(1) lookups and O(1) additions to the playlist.
  - **Complete System:** It successfully integrates all required components into a single, working command-line application.

- **Areas for Improvement**
  With more time, the clear next steps would be:

  - **Add a User Interface (GUI):** Replace the command-line interface with a visual one to make the application more user-friendly.
  - **Implement Data Persistence:** Add a way to save playlists (e.g., to a file or a simple database) so the data isn't lost when the program closes.

- **Relevance to Your Career Goals**
  This project was a vital first step toward my goal of becoming a Software Development Engineer (SDE). It provided hands-on experience in turning requirements into a functional backend system, applying algorithmic thinking to solve real problems, and building a complete application from the ground up—all essential skills for a career in software development.