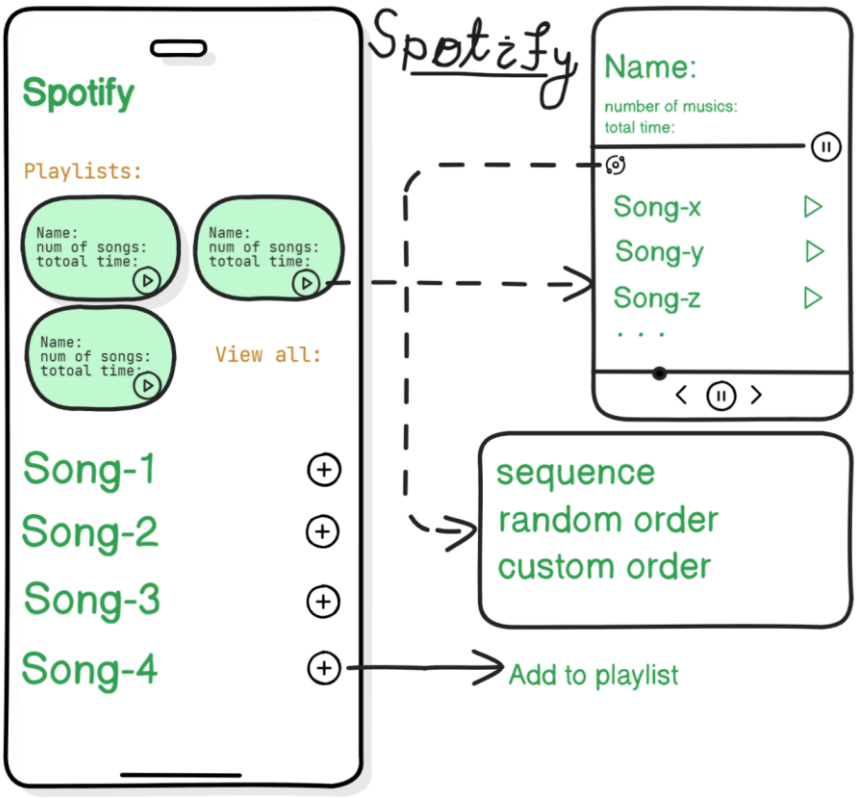
Build Spotify, Music Player App

Problem statement:

- Functional requirement:
  - User can play and pause songs.
  - User can create playlist, add songs to playlist and play entire playlist (sequential or random or custom order).
  - App should support multiple output device (Bluetooth speaker, Wired speaker)
- Non-Functional requirement:
  - Entire design should be easily scalable.
  - New feature (new output device, new order to play music from playlist, etc.) can be easily implemented.

Happy Flow:



Design Patterns in Spotify Music Player Clone:

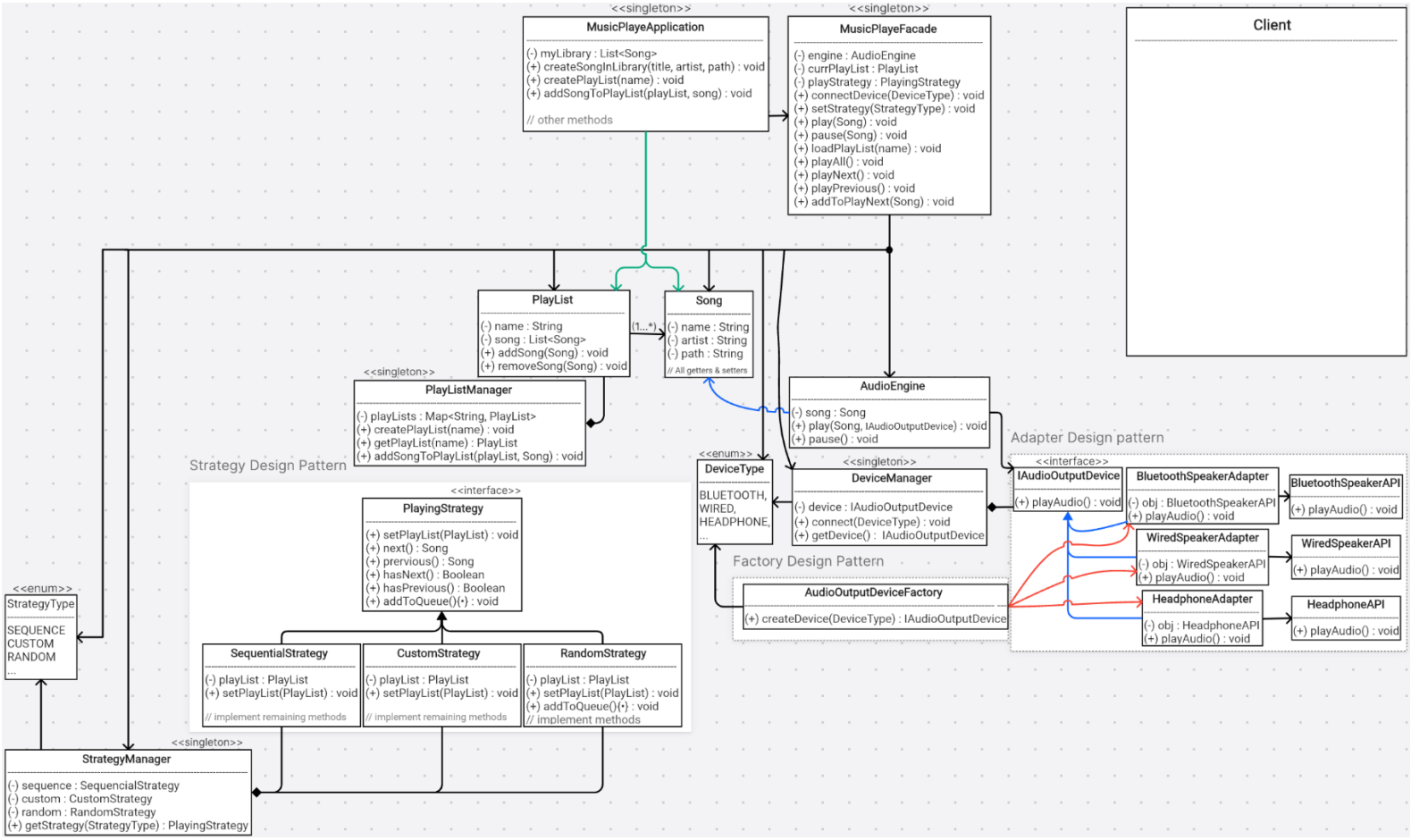
1. Strategy Design Pattern — Playlist Playback Modes
To support multiple playback modes (*Sequential*, *Custom*, *Random*), we use the Strategy Design Pattern. This allows dynamic switching of playback strategies at runtime without modifying the core player logic, ensuring scalability and adherence to SOLID principles.

2. Adapter Design Pattern — Audio Device Integration
Different audio output devices (*Bluetooth*, *Wired*, *Headphones*) expose varied APIs. The Adapter Design Pattern is applied to unify these APIs into a common interface, enabling seamless device switching and easier integration of future devices.

3. Factory Design Pattern — Object Creation
Entities like *Songs*, *Playlists*, and *Audio Devices* are created using the Factory Design Pattern. This abstracts object creation logic, promoting loose coupling and making the system extensible for adding new entity types.

4. Singleton Design Pattern — Service Managers
Core managers such as *PlaylistManager*, *DeviceManager*, and *StrategyManager* are implemented as Singletons to ensure centralized control, consistency, and efficient resource management across the application.

UML:



"Here's the ***Low-Level Design*** for our Spotify-like **music player application**. The idea is to build a system that's ***scalable***, ***maintainable***, and ***flexible*** — ready to support **multiple playback strategies**, **different audio output devices**, and **central service management**. We're using **four key design patterns**: ***Strategy***, ***Adapter***, ***Factory***, and ***Singleton***."

Core Components Overview

1. MusicPlayerFacade (Facade Pattern)

- *Purpose:* Acts as the **single entry point** for the rest of the application to **interact** with our **playback system**.
- ***Responsibilities:***
  - ***Load*** a ***playlist*** into the player.
  - ***Switch*** between ***playback*** strategies (**Sequential**, **Custom**, **Random**).
  - Control playback: **Play**, **Pause**, **Next**, **Previous**.
  - Manage the currently selected **audio output device**.
- ***Why Facade?*** Because we're **hiding** all the **complexity** of strategies, device management, and the audio engine behind a ***simple***, ***unified interface***.

2. Strategies for Playlist Playback (Strategy Pattern)

- We have an ***interface*** IPlayingStrategy with implementations:
  - ***SequentialStrategy*** — plays songs in **order**.
  - ***CustomStrategy*** — allows a **user-defined order**.
  - ***RandomStrategy*** — **shuffles** the playlist.
- The **StrategyManager** or a ***Factory*** is responsible for providing the right strategy based on user selection.
- Benefit: If tomorrow we want "Play by Genre" or "AI Suggested Order," we just add a new strategy — no changes to the existing player logic.

3. Audio Output Device Abstraction (Adapter Pattern)

- Every device type (***Bluetooth***, ***Wired***, ***Headphones***) comes with its **own API** and **data format**.
- We define a **common interface** *IAudioOutputDevice* with methods like ***playAudio()*** and ***pauseAudio()***.
- For each device type, we have an **Adapter**:
  - *BluetoothSpeakerAdapter*
  - *WiredSpeakerAdapter*
  - *HeadphoneAdapter*
- Each adapter wraps the native device API (***BluetoothSpeakerAPI***, ***WiredSpeakerAPI***, ***HeadphoneAPI***) and **converts** our **calls** into the **device's specific *API*** calls.
- This way, our **AudioEngine** doesn't care what device it's using — it just calls the interface.

4. Factory for Object Creation (Factory Pattern)

- For creating ***entities*** like:
  - *Songs*
  - *Playlists*
  - *Output Devices*
- The Factory **hides** creation ***complexity*** and ensures the right subclass or adapter is returned for a given input.
- For example, the **DeviceFactory** takes a ***DeviceType*** and returns the correct ***IAudioOutputDevice*** implementation.

5. Centralized Managers (Singleton Pattern)

- To **avoid *inconsistent states***, certain services exist only once:
  - ***PlaylistManager*** — Manages **creation**, **update**, **retrieval** of ***playlists***.
  - ***DeviceManager*** — Tracks **available** devices and **current *selection***.
  - ***StrategyManager*** — **Provides** available ***strategies***.
- Singleton ensures ***only one instance*** per ***manager***, accessible **globally**.

Why This Design Works

- **Scalable**: New devices or strategies don't require changes to core classes.
- **Maintainable**: Changes are ***isolated*** to their respective modules.
- **Flexible**: Swapping devices or playback modes is instant and independent.
- **Testable**: Each part can be ***unit tested*** in isolation.

Code Link: *https://github.com/sibasundarj8/-System-Design-/tree/main/Projects/SoundBloom%20Music%20player%20App*