Music Streaming System — Final Project Report

Introduction

This project is called **Music Streaming System**.
The main goal of the system is to simulate a simple music-streaming platform where users can listen to tracks, receive notifications about new releases, and get music recommendations.

In this project, we implemented **six design patterns** that we studied in class:
Builder, Factory Method, Strategy, Observer, Decorator, and Facade.
Each pattern solves a specific problem in the system and makes the code easier to extend, reuse, and maintain.

The project includes the following features:

- creating users (Free or Premium),
- searching and playing tracks,
- premium audio enhancements,
- recommendations,
- notifications about new releases.

Body Part

1. Builder Pattern (Track and Playlist)

We used the **Builder** pattern in the Track class (and optionally in Playlist).
The purpose is to create objects step by step without using long constructors.

Example usage:

```
data.add(new Track.Builder()
        .id( v: "t1")
        .title( v: "Believer")
        .artist(a)
        .genre( v: "alt-rock")
        .seconds( v: 204)
        .build());
```

This pattern makes it easy to create flexible and readable objects.

2. Factory Method (UserFactory)

The **Factory Method** pattern is used in DefaultUserFactory to create different types of users.

- FreeUser
- PremiumUser

The factory decides which class to return depending on the selected user type:

```
User free = app.signup( name: "Inayatulla", UserTier.FREE);
User prem = app.signup( name: "Sultan", UserTier.PREMIUM);
```

This helps avoid direct object creation in the main system, improving modularity.

---

3. Strategy Pattern (Recommendations)

The **Strategy** pattern is used for music recommendations.
We implemented multiple strategies:

- ByGenreStrategy
- ByHistoryStrategy

The RecommendationContext chooses which strategy to use at runtime:

```
app.recommend(prem, new ByGenreStrategy( genre: "alt-rock", repo))
        .forEach( Track track -> System.out.println("- " + track.getTitle()));
```

This pattern makes it easy to add new recommendation algorithms without changing existing code.

---

4. Observer Pattern (Notifications)

The **Observer** pattern is implemented using EventBus, Subscriber, and ReleaseEvent.

All users subscribe to the EventBus, and when a new track is released, they receive a notification automatically:

```
Publishing ReleaseEvent("Whatever It Takes")
[Notify] to Sultan: New release: Whatever It Takes
[Notify] to Inayatulla: New release: Whatever It Takes
```

This pattern helps separate event publishing from event handling.

---

5. Decorator Pattern (Premium Streaming Features)

The **Decorator** pattern is used for audio streaming.

Base stream:

- BasicStream

Premium decorators:

- LyricsDecorator
- HdAudioDecorator

When a Premium user plays music:

```
-- Play as PREMIUM (Decorators added: Lyrics + HD) --
Stream info: Basic stream + Lyrics + HD
Output: ♪ chunk from Believer [Lyrics overlay] [HD]
```

Instead of creating many subclasses, the Decorator allows adding features dynamically.

---

6. Facade Pattern (AppFacade)

The **Facade** pattern is implemented in the AppFacade class.
It provides a simple interface for the whole system:

- signup()
- search()
- play()
- recommend()
- publishNewRelease()

The Facade hides all internal complexities of authentication, catalogs, repositories, decorators and observers.

System Demonstration (Output)

Below is an example of the system output:

```
=== MUSIC STREAMING SYSTEM DEMO ===

Created user: Inayatulla (FREE)
Created user: Sultan (PREMIUM)

Track created with Builder: Believer (Imagine Dragons, alt-rock)

Searching for track "Believer" through AppFacade...

-- Play as FREE --
Stream info: Basic stream
Output: ♪ chunk from Believer

Stream info: Basic stream + Lyrics + HD
Output: ♪ chunk from Believer [Lyrics overlay] [HD]

Using strategy: ByGenreStrategy ("alt-rock")
- Believer
- Thunder

Publishing ReleaseEvent("Whatever It Takes")
[Notify] to Sultan: New release: Whatever It Takes
[Notify] to Inayatulla: New release: Whatever It Takes

=== END OF DEMO ===

Process finished with exit code 0
```

This output demonstrates the work of:

- Decorator (premium audio),
- Strategy (recommendations),
- Observer (notifications),
- Facade (simple system access).

Conclusion

In this project, we successfully designed and implemented a Music Streaming System using six design patterns.
Each pattern helped us solve specific structural and behavioral problems:

- Builder made object creation flexible,
- Factory Method separated user creation,

- Strategy allowed dynamic recommendation logic,
- Observer enabled real-time notifications,
- Decorator added premium features without subclassing,
- Facade unified the system into a simple interface.

The final system is modular, extensible, and easy to maintain.