Music Streaming System Report

Gabrielle McCrae

7/26/2025

CEN 3062 - Intro To Software Design

Music Streaming System - Design Patterns Analysis Report

Project Overview

System: Music Streaming Application
Language: Python
Patterns Implemented: Singleton, Strategy, Decorator

This report analyzes the implementation of three design patterns in a music streaming system that manages music playback, supports multiple audio formats, and provides enhanced playlist functionality.

Pattern Analysis

1. Singleton Pattern (Creational)

What problem does the pattern solve in the system?

The Singleton pattern solves the problem of ensuring only one music player instance exists throughout the application. In a music streaming system, having multiple player instances could lead to:

- Conflicting audio playback (multiple songs playing simultaneously)
- Inconsistent player state across different parts of the application
- Resource waste with multiple audio processing instances
- Synchronization issues between different components

What is the basic idea behind the pattern?

The singleton pattern restricts instantiation of a class to a single object and provides global access to that instance. It ensures that:

- Only one instance of the class can be created
- The instance is globally accessible
- Lazy initialization is possible (created only when first needed)
- Thread-safe creation in multi-threaded environments

How does the pattern work in the implementation?

```
class MusicPlayer:
    """
    Singleton Pattern Implementation for Music Player
    Ensures only one music player instance exists throughout the
application
    """
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super(MusicPlayer, cls).__new__(cls)
                    cls._instance._initialized = False
        return cls._instance
```

The implementation uses:

- Class variable `_instance`: Stores the single instance
- Threading lock: Ensures thread-safe creation
- Double-check locking: Prevents multiple instances in concurrent scenarios
- Initialization flag: Prevents re-initialization of the same instance

Main Advantage

- Controlled access: Guarantees single point of control for music playback
- Global state consistency: All components share the same player state
- Resource efficiency: Only one audio processing instance exists
- Thread safety: Prevents race conditions in multi-threaded applications

Main Disadvantage

- Global state dependency: Creates tight coupling between components
- Testing difficulty: Hard to mock or create isolated test instances
- Scalability limitations: Cannot have multiple players for different contexts
- Hidden dependencies: Components implicitly depend on the singleton instance

2. Strategy Pattern (Behavioral)

What problem does the patterns solve in the system?

The Strategy pattern solves the problem of handling different audio format playbook algorithms. Music streaming systems need to:

- Support multiple audio formats(MP3, FLAC, streaming)
- Switch between different processing algorithms at runtime
- Add new formats without modifying existing code
- Optimize playbook based on format characteristics

What is the basic idea behind the pattern?

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows:

- Algorithm selection at runtime
- Independence of algorithms from clients that use them
- Easy addition of new strategies without modifying existing code
- Elimination of conditional statements for algorithm selection

How does the pattern work in the implementation

```
class PlaybackStrategy(ABC):
    """
    Strategy Pattern Interface for different audio format playback
    Allows switching between different audio processing algorithms
    """

    @abstractmethod
    def play(self, song):
        pass

    @abstractmethod
    def get_format_info(self):
        pass

class MP3Strategy(PlaybackStrategy):
    """Strategy for MP3 audio format"""

    def play(self, song):
        print(f"🎵 Playing MP3: {song.title} by {song.artist}")
        print("   Using MP3 decoder with lossy compression")
```

The implementation includes:

- Strategy interface: `PlaybackStrategy` defines common methods

- Concrete strategies: `MP3Strategy`, `FLACStrategy`, `StreamingStrategy`
- Context: `MusicPlayer` uses strategies through the interface
- Runtime switching: Strategies can be changed during execution

Main Advantage:

- Runtime flexibility: Can switch audio processing algorithms during execution
- Open/Closed principle: Easy to add new formats without modifying existing code
- Code reusability: Strategies can be reused across different contexts
- Separation of concerns: Each format has its own dedicated implementation

Main Disadvantage:

- Increased complexity: More classes and interfaces to maintain
- Strategy proliferation: Can lead to many small strategy classes
- Client awareness: Clients must understand different strategies to choose appropriately
- Communication overhead: Strategies might need more context than provided by the interface

3. Decorator Pattern (Structural)

What problem does the pattern solve in the system?

The Decorator pattern solves the problem of adding functionality to playlists without modifying their structure. Music streaming systems need to:

- Add features like shuffle, repeat, and analytics to playlists
- Combine multiple enhancements (shuffle + repeat + analytics)
- Maintain the original playlist interface
- Allow dynamic addition and removal of features

What is the basic idea behind the pattern?

The Decorator pattern allows behavior to be added to objects dynamically without altering their structure. It:

- Provides a flexible alternative to subclassing
- Allows multiple decorators to be combined
- Maintains the original object's interface
- Enables runtime composition of behaviors

How does the pattern work in the implementation?

```
class Playlist(ABC):
    @abstractmethod
    def play_all(self):
        pass
```

```python
class PlaylistDecorator(Playlist):
    """Base decorator for playlists"""

    def __init__(self, playlist: Playlist):
        self._playlist = playlist
```

```python
class ShuffleDecorator(PlaylistDecorator):
    def play_all(self):
        print("🔀 Shuffle mode enabled!")
        songs = self.get_songs()
        self.random.shuffle(songs)
```

The implementation features:

- Component interface: `Playlist` defines the basic interface
- Concrete component: `BasicPlaylist` provides basic functionality
- Base decorator: `PlaylistDecorator` maintains the interface
- Concrete decorators: `ShuffleDecorator`, `RepeatDecorator`, `AnalyticsDecorator`
- composition : Decorators can be nested for combined functionality

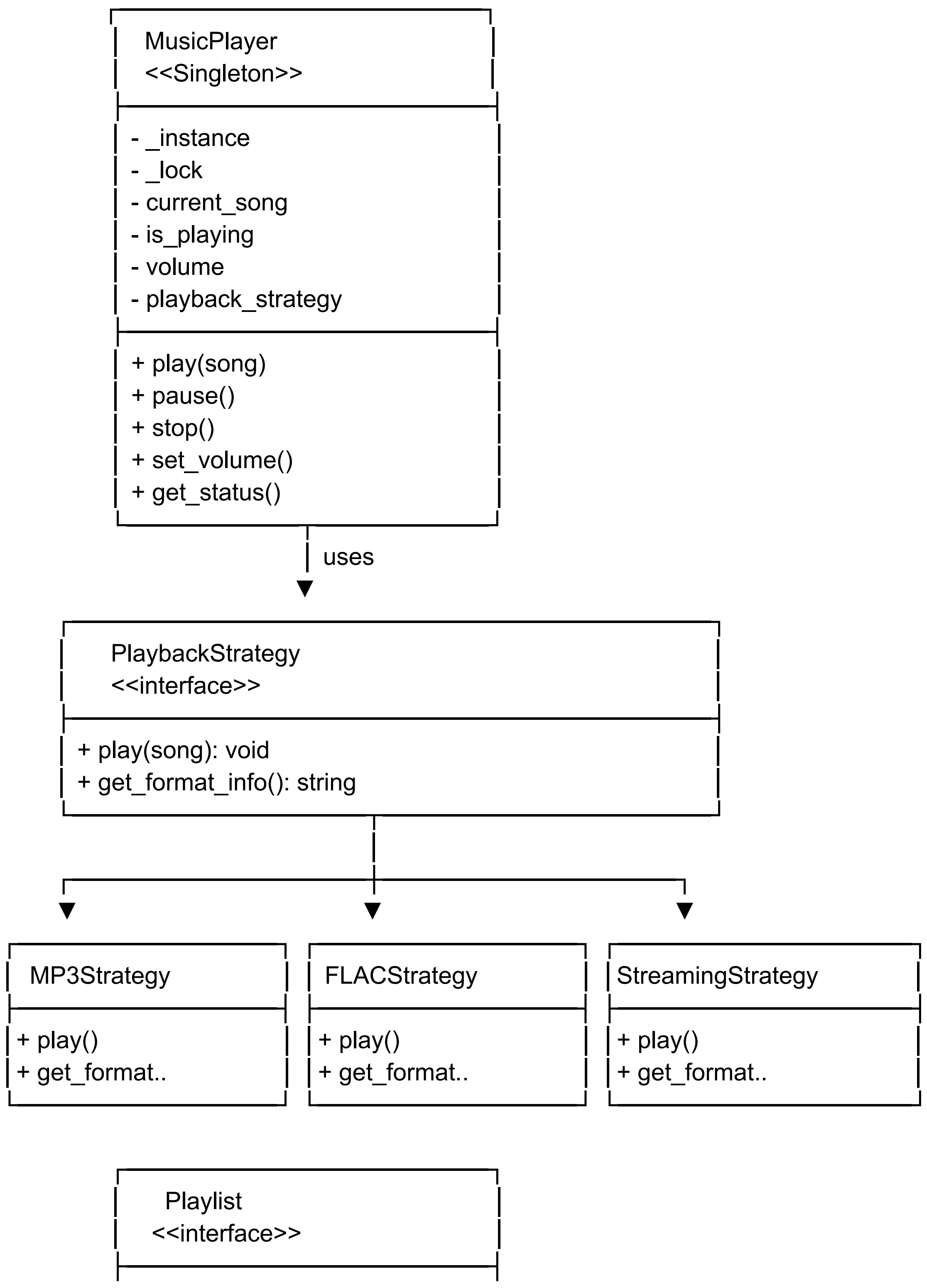
Main Advantage:

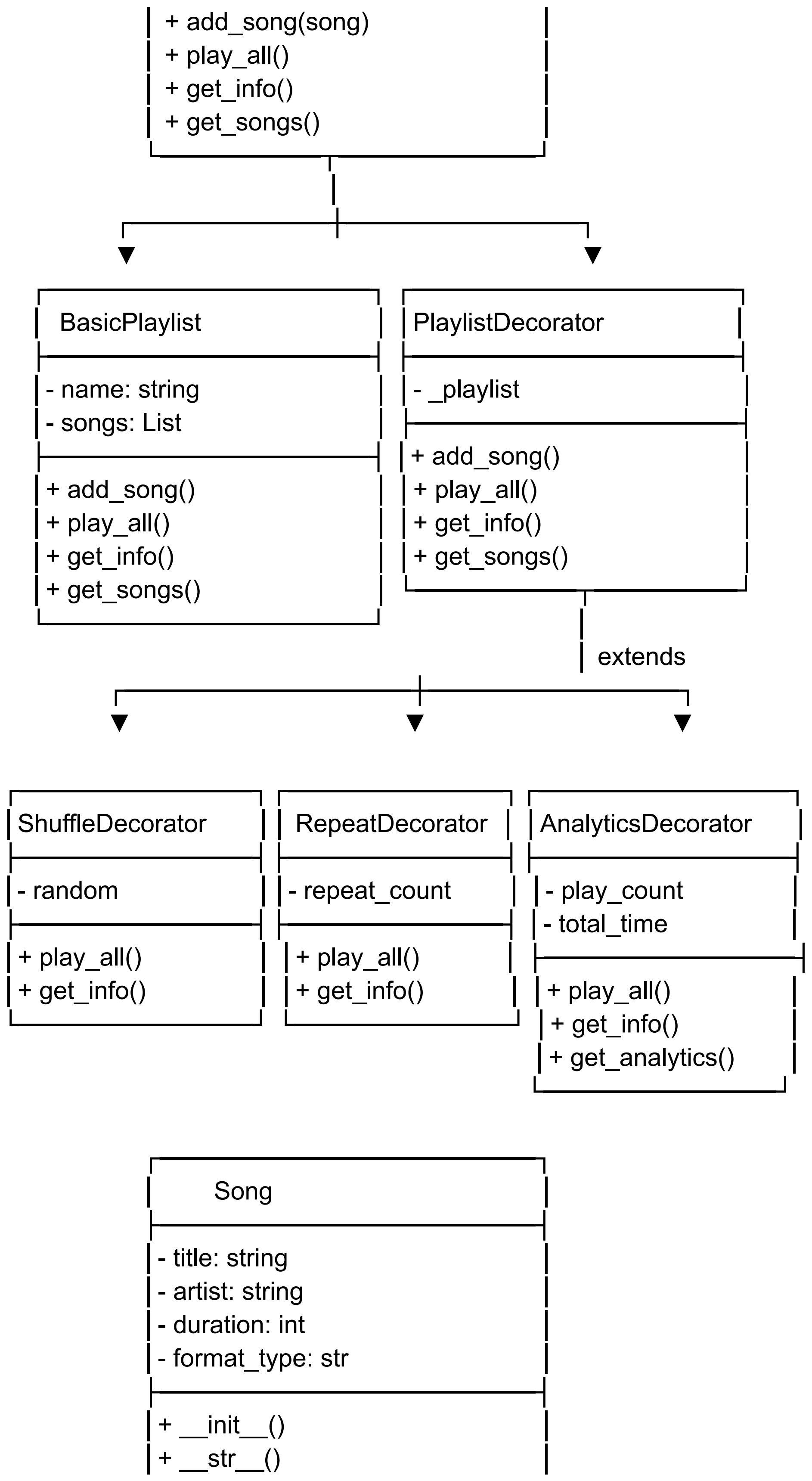
- Dynamic enhancement: Features can be added at runtime without modifying original classes
- Flexible composition: Multiple decorators can be combined in any order
- Single Responsibility: Each decorator handles one specific enhancement
- Interface preservation: Enhanced objects maintain the same interface as originals

Main Disadvantage:

- Complexity Increase: Creates many small objects and can be hard to debug
- Identity issues: Decorated objects are not identical to original objects
- Order dependency: The order of decorator application can affect behavior
- Interface Limitations: All decorators must conform to the same interface

UML Class Diagram




+ add_song(song)
+ play_all()
+ get_info()
+ get_songs()
BasicPlaylist
- name: string
- songs: List
+ add_song()
+ play_all()
+ get_info()
+ get_songs()
PlaylistDecorator
- _playlist
+ add_song()
+ play_all()
+ get_info()
+ get_songs()
extends
ShuffleDecorator
- random
+ play_all()
+ get_info()
RepeatDecorator
- repeat_count
+ play_all()
+ get_info()
AnalyticsDecorator
- play_count
- total_time
+ play_all()
+ get_info()
+ get_analytics()
Song
- title: string
- artist: string
- duration: int
- format_type: str
+ __init__()
+ __str__()

Relationships:

- MusicPlayer uses PlaybackStrategy (Strategy Pattern)
- PlaybackStrategy is implemented by concrete strategies (Strategy Pattern)
- PlaylistDecorator decorates Playlist (Decorator Pattern)
- BasicPlaylist implement Playlist interface
- Concrete Decorators extend PlaylistDecorator
- MusicPlayer is a singleton (Singleton Pattern)

Pattern Integration

The three patterns work together seamlessly:

1. Singleton MusicPlayer provides a central control point for music playback
2. Strategy Pattern allows the player to handle different audio formats dynamically
3. Decorator Pattern enables rich playlist functionality that can be combined flexibly

This combination demonstrates how design patterns can be layered to create robust, maintainable software systems.

Conclusion

This music streaming system successfully demonstrates how three different design patterns solve distinct problems:

- Singleton ensures controlled resource management
- Strategy provides algorithmic flexibility
- Decorator enables dynamic feature enhancement

The implementation showcases real-world applicability of design patterns and their ability to create maintainable, extensible software architectures.