Software Engineering: Project - 2

Team - 19

Bonus Task: Non-collaborative Playlists

Existing System

Users can create playlists by clicking the “ADD TO PLAYLIST” button which displays the option to either create a playlist or add songs to already existing playlists, but the public playlists are Collaborative.

Modified System

Now users can create non – collaborative public playlists as well. While making a playlist, they can choose the option "Read-Only Public". Implemented the above feature using “DECORATOR PATTERN”.

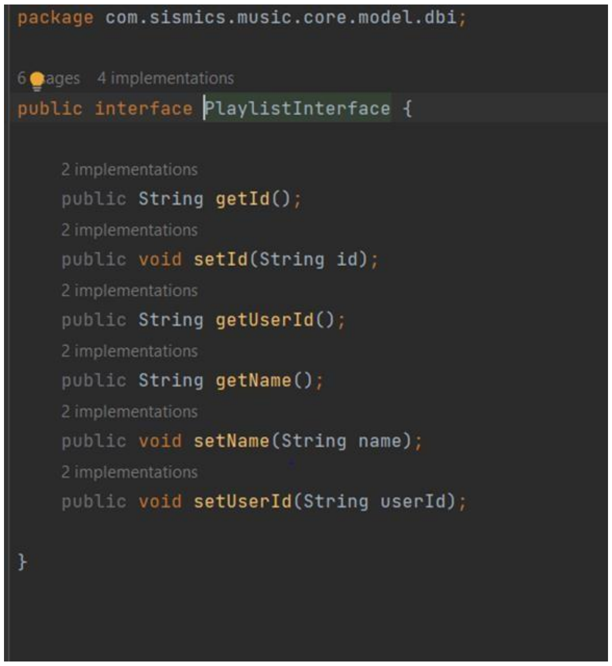
Decorator pattern

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Benefits of using Decorator pattern:

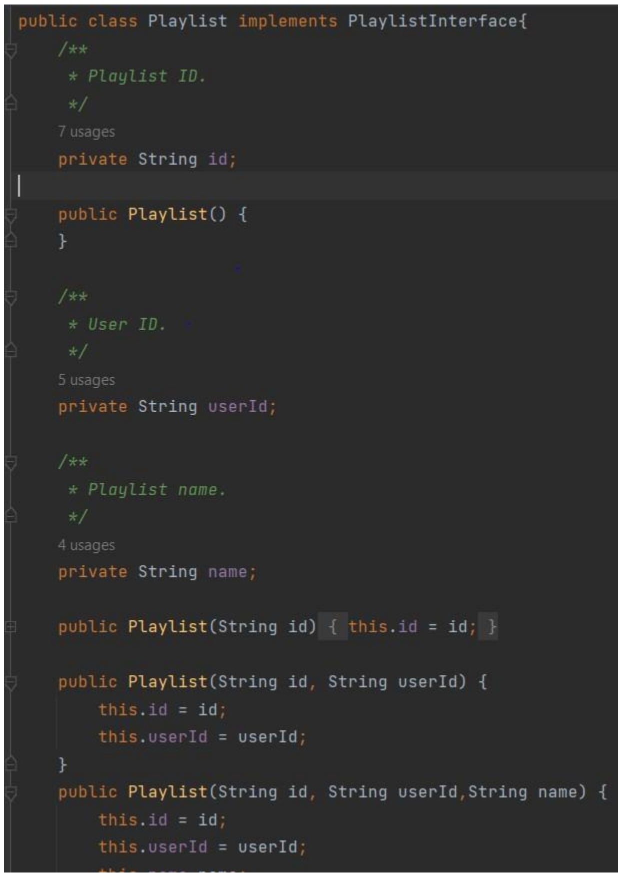
- **Enhances code reusability**: The Decorator pattern allows you to add new functionality to an existing object without modifying its original code. This helps in keeping the code modular and makes it easier to reuse existing code.
- **Provides flexibility**: With the Decorator pattern, you can add or remove behavior from an object at runtime. This flexibility allows you to create complex behavior by combining multiple decorators.
- **Simplifies code maintenance:** When you use the Decorator pattern, you can add or remove functionality by simply adding or removing decorators. This simplifies code maintenance and reduces the chances of introducing bugs.
- **Allows incremental feature development**: The Decorator pattern allows you to add new features to an application incrementally without impacting the existing functionality. This helps in avoiding large code refactoring efforts and reduces the risk of introducing new bugs.
- **Supports open/closed principle**: The Decorator pattern follows the open/closed principle, which states that software entities should be open for extension but closed for modification. With the Decorator pattern, you can extend the functionality of an object without modifying its original code, thus adhering to the open/closed principle.

Code Changes



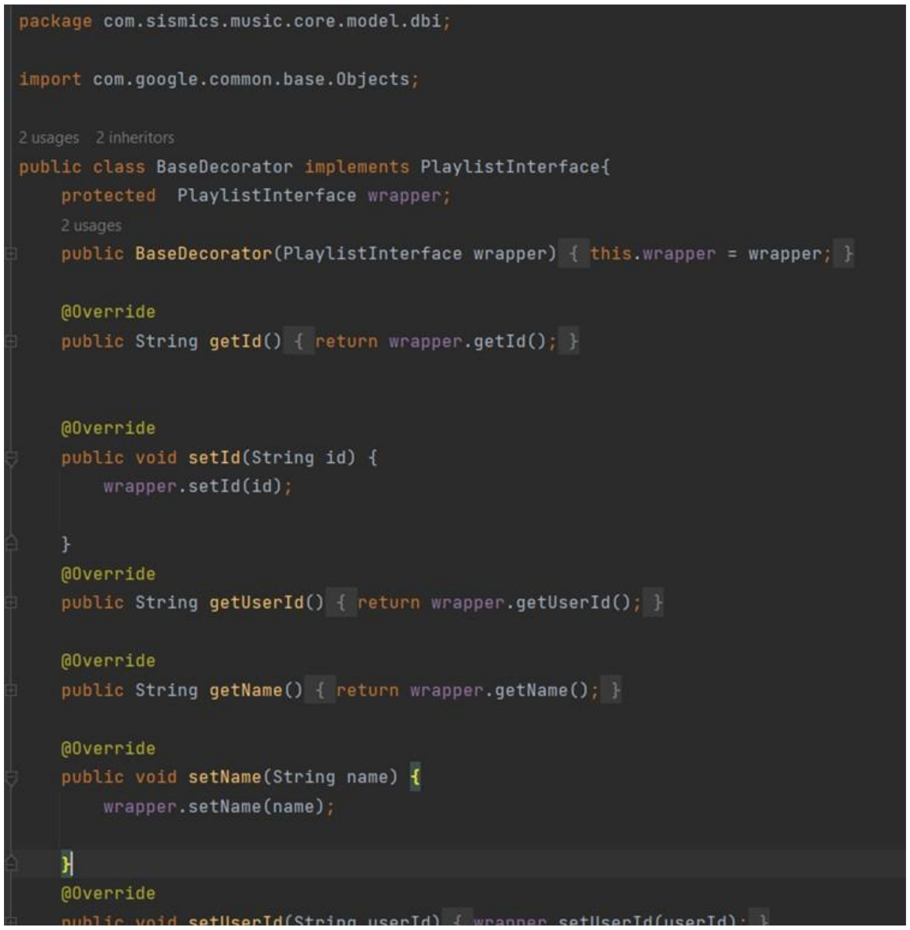
```java
package com.sismics.music.core.model.dbi;

6 usages   4 implementations
public interface PlaylistInterface {

    2 implementations
    public String getId();
    2 implementations
    public void setId(String id);
    2 implementations
    public String getUserId();
    2 implementations
    public String getName();
    2 implementations
    public void setName(String name);
    2 implementations
    public void setUserId(String userId);

}
```

Figure 1 Interface component



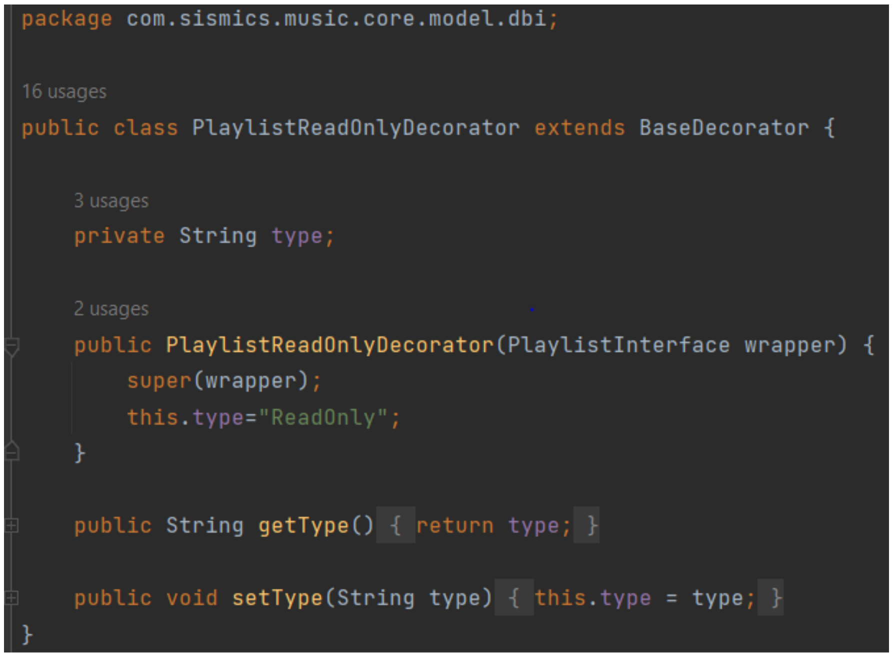
```java
public class Playlist implements PlaylistInterface{
    /**
     * Playlist ID.
     */
    7 usages
    private String id;

    public Playlist() {
    }

    /**
     * User ID.
     */
    5 usages
    private String userId;

    /**
     * Playlist name.
     */
    4 usages
    private String name;

    public Playlist(String id) { this.id = id; }

    public Playlist(String id, String userId) {
        this.id = id;
        this.userId = userId;
    }
    public Playlist(String id, String userId,String name) {
        this.id = id;
        this.userId = userId;
```

Figure 2 Concrete Component



```java
package com.sismics.music.core.model.dbi;

import com.google.common.base.Objects;

2 usages  2 inheritors
public class BaseDecorator implements PlaylistInterface{
    protected  PlaylistInterface wrapper;
    2 usages
    public BaseDecorator(PlaylistInterface wrapper) { this.wrapper = wrapper; }

    @Override
    public String getId() { return wrapper.getId(); }


    @Override
    public void setId(String id) {
        wrapper.setId(id);

    }
    @Override
    public String getUserId() { return wrapper.getUserId(); }

    @Override
    public String getName() { return wrapper.getName(); }

    @Override
    public void setName(String name) {
        wrapper.setName(name);

    }
    @Override
    public void setUserId(String userId) { wrapper.setUserId(userId); }
```

Figure 3 Base Decorator



```java
package com.sismics.music.core.model.dbi;

16 usages
public class PlaylistReadOnlyDecorator extends BaseDecorator {

    3 usages
    private String type;

    2 usages
    public PlaylistReadOnlyDecorator(PlaylistInterface wrapper) {
        super(wrapper);
        this.type="ReadOnly";
    }

    public String getType() { return type; }

    public void setType(String type) { this.type = type; }
}
```

Figure 4 Concrete Decorator

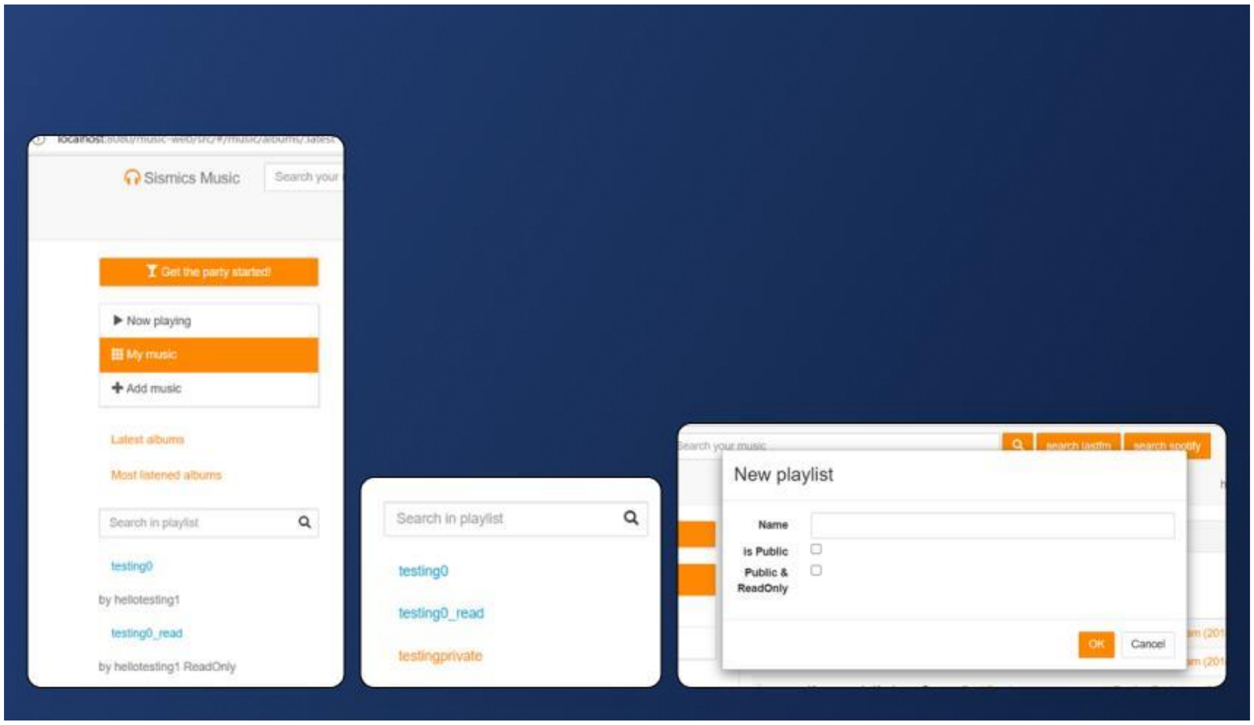
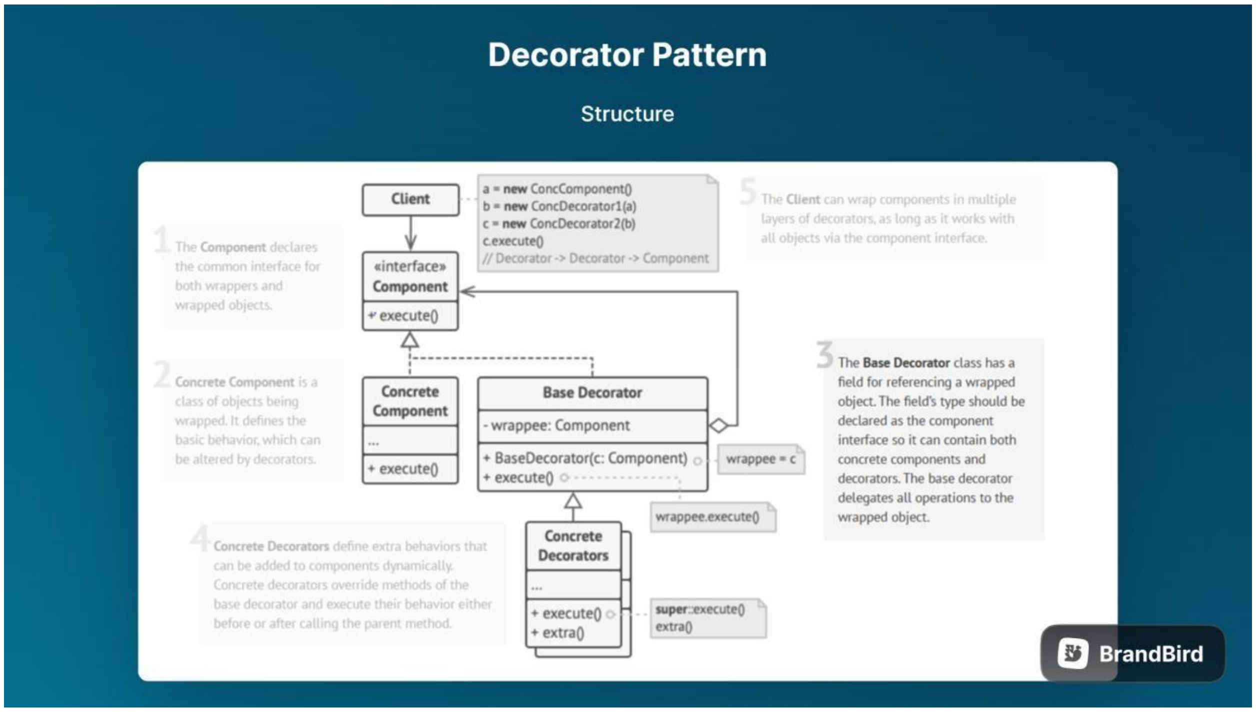


Figure 5 Demo of creation and display of playlists.



```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3 The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.