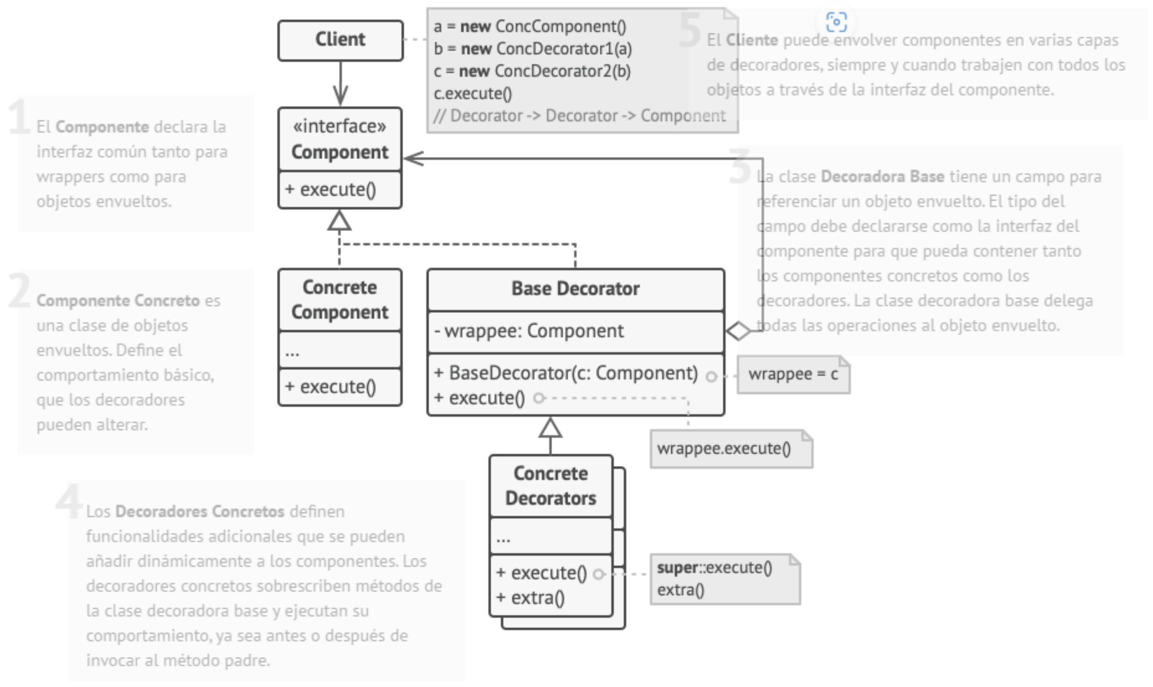
DECORATOR

MIGUEL ANGEL RAMIREZ JUAREZ
JAVA ACADEMY

AUGUST/ 22/ 2024

Decoration Pattern in Java?

The **Decorator** pattern is a structural design pattern used in object-oriented programming to extend the functionalities of objects in a flexible and reusable way. Instead of modifying the original object or creating a large number of subclasses to add new features, the Decorator pattern allows you to "decorate" or "wrap" objects with additional behavior dynamically.


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
5 El Cliente puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente.
1 El Componente declara la interfaz común tanto para wrappers como para objetos envueltos.
«interface» Component
+ execute()
3 La clase Decoradora Base tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores. La clase decoradora base delega todas las operaciones al objeto envuelto.
2 Componente Concreto es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.
Concrete Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
4 Los Decoradores Concretos definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes. Los decoradores concretos sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.
Concrete Decorators
...
+ execute()
+ extra()
super::execute()
extra()

Goal

The goal of this task was to implement the Decorator pattern in Java to enhance the flexibility and extensibility of a subscription system. The Decorator pattern allows for dynamic addition of functionalities to objects without altering the existing object code. Unit tests were also implemented to ensure the system functions correctly and achieves a minimum of 85% code coverage.

Problem Description

The Decorator pattern was used to extend the functionality of a basic subscription in a subscription system. Decorators were implemented to add features such as offline downloads, premium content, and ad removal. The implementation included:

1. **Basic Subscription**: The base class providing a standard subscription with a fixed cost.
2. **Decorators**: Classes extending the basic subscription to add additional features and adjust the cost.

Implementation

- **Subscription**: Interface defining methods getDescription() and getCost().
- **BasicSubscription**: Implementation of the basic subscription.
- **SubscriptionDecorator**: Abstract class implementing the Subscription interface, delegating functionality to a decorated Subscription object.
- **OfflineDownloadDecorator, PremiumContentDecorator, NoAdsDecorator**: Concrete classes extending SubscriptionDecorator to add specific features.

Unit Testing

Unit tests were designed to verify the correct operation of the Decorator pattern:

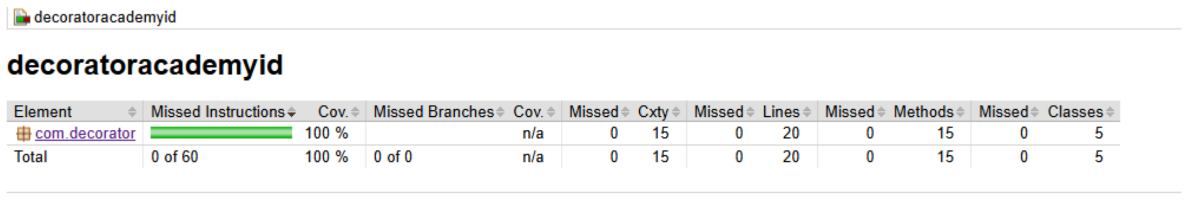
1. **Individual Decorator Tests**: Check that each decorator adds the expected description and cost.
2. **Main Method Test**: Captures and verifies the output of the main method to ensure the format and content are correct.

Issues Encountered and Solutions:

- **String Comparison Issue**: Comparison between expected and actual output failed due to differences in spaces and line breaks. This was resolved by using trim() and ensuring that the output format exactly matched the expected output.

Test Execution

Tests were executed using Maven and the system-rules library to capture standard output. Code coverage was verified with the jacoco-maven-plugin, ensuring a minimum coverage of 85%.


decoratoracademyid

decoratoracademyid

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.decorator	🟩	100 %		n/a	0	15	0	20	0	15	0	5
Total	0 of 60	100 %	0 of 0	n/a	0	15	0	20	0	15	0	5

Conclusion

The implementation of the Decorator pattern and unit tests ensured that the subscription system was flexible and extensible. The decorators allowed for adding additional functionalities without modifying existing code, and the unit tests validated the system's behavior. Capturing and comparing standard output was effectively resolved to validate the main method's functionality.