Decorator Design Pattern in Java with Example

kashish2008

Read Discuss Courses Practice Video

Decorator design pattern allows us to dynamically add functionality and behavior to an object without affecting the behavior of other existing objects within the same class. We use inheritance to extend the behavior of the class. This takes place at compile-time, and all the instances of that class get the extended behavior.

- Decorator patterns allow a user to add new functionality to an existing object without altering its structure. So, there is no change to the original class.
- The decorator design pattern is a structural pattern, which provides a wrapper to the existing class.
- The decorator design pattern uses abstract classes or interfaces with the composition to implement the wrapper.
- Decorator design patterns create decorator classes, which wrap the original class and supply additional functionality by keeping the class methods' signature unchanged.
- Decorator design patterns are most frequently used for applying single responsibility principles since we divide the functionality into classes with unique areas of concern.

Remember: Certain key points are to be taken into consideration that are as follows:

1. *Decorator design pattern is useful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the amount of choices are more.*
2. *The disadvantage of decorator design pattern is that it uses plenty of similar kind of objects (decorators)*
3. *Decorator pattern is used a lot in Java IO classes, like FileReader, BufferedReader, etc.*

Procedure:

1. Create an interface.

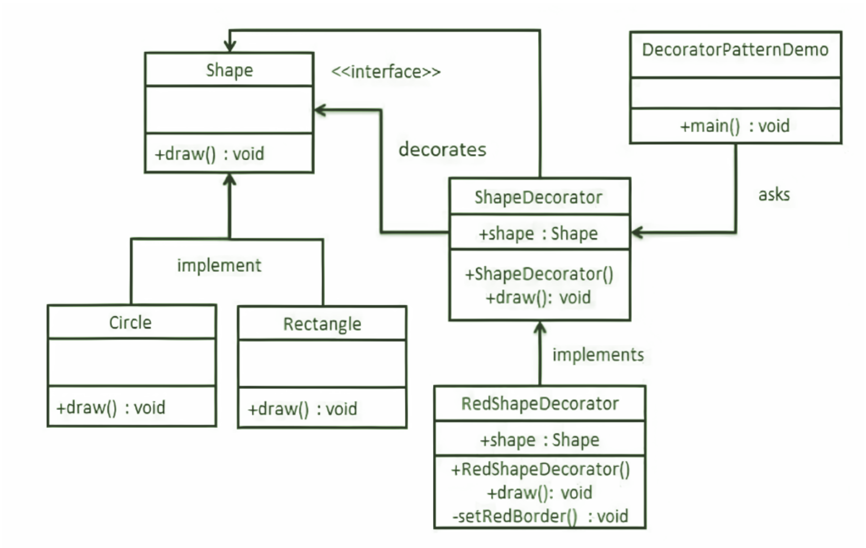
3. Create an abstract decorator class implementing the above same interface.
4. Create a concrete decorator class extending the above abstract decorator class.
5. Now use the concrete decorator class created above to decorate interface objects.
6. Lastly, verify the output

Implementation:

We're going to create a Shape interface and concrete classes implementing the Shape interface. We will then create an abstract decorator class ShapeDecorator implementing the Shape interface and having the Shape object as its instance variable.

1. 'Shape' is the name of the interface

3. 'ShapeDecorator' is our abstract decorator class implementing the same 'Shape' interface.
4. RedShapeDecorator is a concrete class implementing ShapeDecorator.
5. DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate Shape objects.



Step 1: Creating an interface named 'Shape'

Example

Java

```
// Interface named Shape
public interface Shape {

    // Method inside interface
    void draw();
}
```

Step 2: Create concrete classes implementing the same interface.

Rectangle.java and Circle.java are as follows

```java
// Class 1
// Class 1 will be implementing the Shape interface

// Rectangle.java
public class Rectangle implements Shape {

    // Overriding the method
    @Override public void draw()
    {
        // /Print statement to execute when
        // draw() method of this class is called
        // later on in the main() method
        System.out.println("Shape: Rectangle");
    }
}
```

Java

```java
// Circle.java
public class Circle implements Shape {

    @Override
    public void draw()
    {
        System.out.println("Shape: Circle");
    }
}
```

Step 3: Create an abstract decorator class implementing the Shape interface.

Example

Java

```java
// Class 2
// Abstract class
// ShapeDecorator.java
```

```
    // Method 1
    // Abstract class method
    public ShapeDecorator(Shape decoratedShape)
    {
        // This keywordd refers to current object itself
        this.decoratedShape = decoratedShape;
    }

    // Method 2 - draw()
    // Outside abstract class
    public void draw() { decoratedShape.draw(); }
}
```

Step 4: Create a concrete decorator class extending the ShapeDecorator class.

Example

Java

```
// Class 3
// Concrete class extending the abstract class
// RedShapeDecorator.java
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratedShape)
    {
        super(decoratedShape);
    }

    @Override public void draw()
    {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape)
    {
```

Step 5: Using the RedShapeDecorator to decorate Shape objects.

Example

Java

```java
// DecoratorPatternDemo.java

// Class
// Main class
public class DecoratorPatternDemo {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of Shape interface
        // inside the main() method
        Shape circle = new Circle();

        Shape redCircle
            = new RedShapeDecorator(new Circle());

        Shape redRectangle
            = new RedShapeDecorator(new Rectangle());

        // Display message
        System.out.println("Circle with normal border");

        // Calling the draw method over the
        // object calls as created in
        // above classes

        // Call 1
        circle.draw();

        // Display message
        System.out.println("\nCircle of red border");

        // Call 2
        redCircle.draw();
        // Call 3
        redRectangle.draw();
    }
}
```

Step 6: Verifying the output

Output:

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

Output explanation:

Glancing at the decorator design pattern one can conclude out that this is often a decent choice in the following cases where

- *When we wish to add, enhance or perhaps remove the behavior or state of objects.*
- *When we just want to modify the functionality of a single object of the class and leave others unchanged.*

1. Pattern pattern() method in Java with Examples
2. Difference Between State and Strategy Design Pattern in Java
3. Design Patterns in Java - Iterator Pattern
4. Singleton Design Pattern | Implementation
5. Singleton Design Pattern | Introduction
6. Object Pool Design Pattern
7. Spring - When to Use Factory Design Pattern Instead of Dependency Injection
8. MVC (Model View Controller) Architecture Pattern in Android with Example
9. MVP (Model View Presenter) Architecture Pattern in Android with Example
10. Material Design EditText in Android with Example

Related Tutorials

1. Spring MVC Tutorial
2. Spring Tutorial
3. Spring Boot Tutorial
4. Java 8 Tutorial
5. Introduction to Monotonic Stack - Data Structure and Algorithm Tutorials

Collections in Java

Article Contributed By :

kashish2008
kashish2008

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : adityakumar129, surinderdawra388, twinshu_parmar, sagartomar9927, swatiomar09

Article Tags : Java-Design-Patterns, Picked, Technical Scripter 2020, Java, Technical Scripter

Practice Tags : Java

Improve Article Report Issue

In Media

Contact Us

Terms and Conditions

Privacy Policy

Copyright Policy

Third-Party Copyright Notices

Advertise with us

Python Backend LIVE

Android App Development

DevOps LIVE

DSA in JavaScript

Languages

Python

Java

C++

GoLang

SQL

R Language

Android Tutorial

Data Structures

Array

String

Linked List

Stack

Queue

Tree

Graph

Algorithms

Sorting

Searching

Greedy

Dynamic Programming

Pattern Searching

Recursion

Backtracking

Web Development

HTML

CSS

JavaScript

Bootstrap

ReactJS

AngularJS

NodeJS

Computer Science

GATE CS Notes

Python

Python Programming Examples

Software Engineering

Digital Logic Design

Engineering Maths

OpenCV Python Tutorial

Python Interview Question

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

Competitive Programming

Top DSA for CP

Top 50 Tree Problems

Top 50 Graph Problems

Top 50 Array Problems

Top 50 String Problems

Top 50 DP Problems

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

Scalability in SD

Databases in SD

High Level Design or HLD

Low Level Design or LLD

Top SD Interview Questions

Interview Corner

Company Preparation

Preparation for SDE

Company Interview Corner

Experienced Interview

GfG School

CBSE Notes for Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11