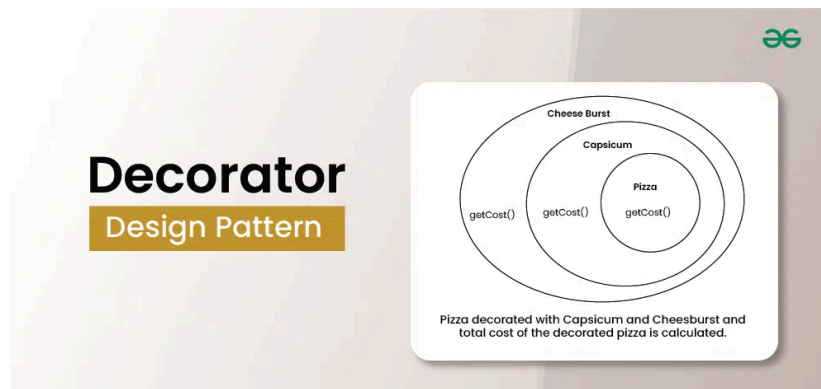




Decorator Design Pattern

Last Updated : 03 Jan, 2025

The Decorator Design Pattern is a [structural design pattern](#) that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It involves creating a set of decorator classes that are used to wrap concrete components.



Important Topics for Decorator Design Pattern

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

Got It !

- [Key Components of the Decorator Design Pattern](#)
- [Example of Decorator Design Pattern](#)
- [Advantages of the Decorator Design Pattern](#)
- [Disadvantages of the Decorator Design Pattern](#)

What is a Decorator Design Pattern?

The Decorator Design Pattern is a [structural design pattern](#) used in software development. It allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. This pattern is useful when you need to add functionality to objects in a flexible and reusable way.

Characteristics of the Decorator Pattern

- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.
- It follows the open/closed principle, as new decorators can be added without modifying existing code, making it a powerful tool for building modular and customizable software components.
- The Decorator Pattern is commonly used in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner, such as in text formatting, graphical user interfaces, or customization of products like coffee or ice cream.

Real-World Example of Decorator Design Pattern

Consider a video streaming platform where users can watch movies and TV shows. Each video content may have additional features or options available, such as subtitles, language preferences, video quality options, and audio enhancements.

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

- For example, a user might select the option to enable subtitles, change the language of the audio track, or adjust the video quality settings.
- Each of these options acts as a decorator that enhances the viewing experience without altering the underlying video content.
- By using the Decorator pattern, the streaming platform can dynamically apply these additional features to the video content based on user preferences, providing a customizable viewing experience.

Use Cases for the Decorator Pattern

Below are some of the use cases of Decorator Design Pattern:

- **Extending Functionality:** When you have a base component with basic functionality, but you need to add additional features or behaviors to it dynamically without altering its structure. Decorators allow you to add new responsibilities to objects at runtime.
- **Multiple Combinations of Features:** When you want to provide multiple combinations of features or options to an object. Decorators can be stacked and combined in different ways to create customized variations of objects, providing flexibility to users.
- **Legacy Code Integration:** When working with legacy code or third-party libraries where modifying the existing codebase is not feasible or desirable, decorators can be used to extend the functionality of existing objects without altering their implementation.
- **GUI Components:** In graphical user interface (GUI) development, decorators can be used to add additional visual effects, such as borders, shadows, or animations, to GUI components like buttons, panels, or windows.
- **Input/Output Streams:** Decorators are commonly used in input/output stream classes in languages like Java. They allow you to wrap streams with additional functionality such as buffering, compression,

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

- **Component Interface:** This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.
- **Concrete Component:** These are the basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.
- **Decorator:** This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.
- **Concrete Decorator:** These are the concrete classes that extend the Decorator class. They add specific behaviors or responsibilities to the Component. Each Concrete Decorator can add one or more behaviors to the Component.

Example of Decorator Design Pattern

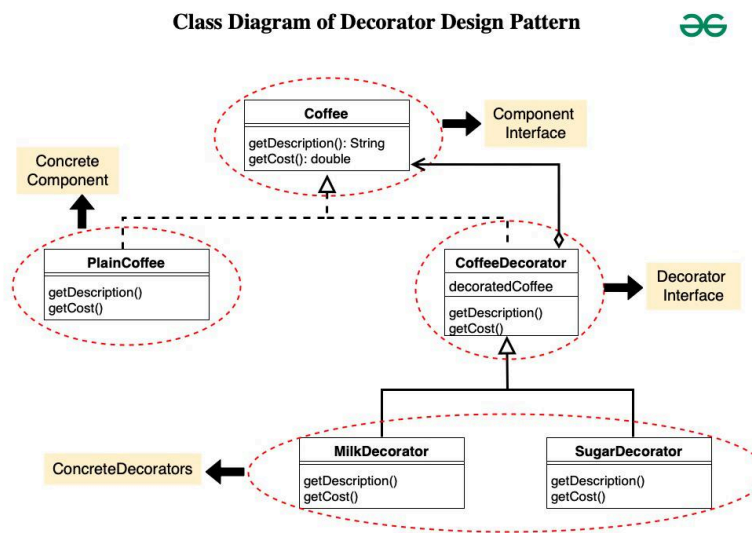
Below is the problem statement to understand the Decorator Design Pattern:

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

implement a system where we can dynamically add these add-ons to a coffee order without modifying the coffee classes themselves.

Using the Decorator Pattern allows us to add optional features (add-ons) to coffee orders dynamically without altering the core coffee classes. This promotes code flexibility, scalability, and maintainability as new add-ons can be easily introduced and combined with different types of coffee orders.



Lets Breakdown the code into component wise code:

1. Component Interface(Coffee)

- This is the interface `Coffee` representing the component.
- It declares two methods `getDescription()` and `getCost()` which must be implemented by concrete components and decorators.

```
1 // Coffee.java
```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

2. ConcreteComponent(PlainCoffee)

- PlainCoffee is a concrete class implementing the Coffee interface.
- It provides the description and cost of plain coffee by implementing the getDescription() and getCost() methods.

```
1  // PlainCoffee.java
2  public class PlainCoffee implements Coffee {
3      @Override
4      public String getDescription() {
5          return "Plain Coffee";
6      }
7
8      @Override
9      public double getCost() {
10         return 2.0;
11     }
12 }
```

3. Decorator(CoffeeDecorator)

- CoffeeDecorator is an abstract class implementing the Coffee interface.
- It maintains a reference to the decorated Coffee object.
- The getDescription() and getCost() methods are implemented to delegate to the decorated coffee object.

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

```

2  public abstract class CoffeeDecorator
   implements Coffee {
3      protected Coffee decoratedCoffee;
4
5      public CoffeeDecorator(Coffee
decoratedCoffee) {
6          this.decoratedCoffee = decoratedCoffee;
7      }
8
9      @Override
10     public String getDescription() {
11         return
decoratedCoffee.getDescription();
12     }
13
14     @Override
15     public double getCost() {
16         return decoratedCoffee.getCost();
17     }
18 }

```

4. ConcreteDecorators(MilkDecorator,SugarDecorator)

- MilkDecorator and SugarDecorator are concrete decorators extending CoffeeDecorator.
- They override getDescription() to add the respective decorator

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.



```
1 // MilkDecorator.java
2 public class MilkDecorator extends
CoffeeDecorator {
3     public MilkDecorator(Coffee
decoratedCoffee) {
4         super(decoratedCoffee);
5     }
6
7     @Override
8     public String getDescription() {
9         return decoratedCoffee.getDescription()
+ ", Milk";
10    }
11
12    @Override
13    public double getCost() {
14        return decoratedCoffee.getCost() + 0.5;
15    }
16 }
17
18 // SugarDecorator.java
19 public class SugarDecorator extends
CoffeeDecorator {
20     public SugarDecorator(Coffee
decoratedCoffee) {
21         super(decoratedCoffee);
22     }
23
24     @Override
25     public String getDescription() {
26         return decoratedCoffee.getDescription()
+ ", Sugar";
27    }
28
29    @Override
30    public double getCost() {
```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

Complete Code of the above problem statement:

Below is the complete code of the above problem statement:

```
1  // Coffee.java
2  public interface Coffee {
3      String getDescription();
4      double getCost();
5  }
6
7  // PlainCoffee.java
8  public class PlainCoffee implements Coffee {
9      @Override
10     public String getDescription() {
11         return "Plain Coffee";
12     }
13
14     @Override
```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

```

19
20 // CoffeeDecorator.java
21 public abstract class CoffeeDecorator
    implements Coffee {
22     protected Coffee decoratedCoffee;
23
24     public CoffeeDecorator(Coffee
    decoratedCoffee) {
25         this.decoratedCoffee = decoratedCoffee;
26     }
27
28     @Override
29     public String getDescription() {
30         return
    decoratedCoffee.getDescription();
31     }
32
33     @Override
34     public double getCost() {
35         return decoratedCoffee.getCost();
36     }
37 }
38
39 // MilkDecorator.java
40 public class MilkDecorator extends
    CoffeeDecorator {
41     public MilkDecorator(Coffee
    decoratedCoffee) {
42         super(decoratedCoffee);
43     }
44
45     @Override
46     public String getDescription() {
47         return decoratedCoffee.getDescription()
    + ", Milk";
48     }
49

```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

```

54     }
55
56     // SugarDecorator.java
57     public class SugarDecorator extends
CoffeeDecorator {
58         public SugarDecorator(Coffee
decoratedCoffee) {
59             super(decoratedCoffee);
60         }
61
62         @Override
63         public String getDescription() {
64             return decoratedCoffee.getDescription()
+ ", Sugar";
65         }
66
67         @Override
68         public double getCost() {
69             return decoratedCoffee.getCost() + 0.2;
70         }
71     }
72
73     // Main.java
74     public class Main {
75         public static void main(String[] args) {
76             // Plain Coffee
77             Coffee coffee = new PlainCoffee();
78             System.out.println("Description: " +
coffee.getDescription());
79             System.out.println("Cost: $" +
coffee.getCost());
80
81             // Coffee with Milk
82             Coffee milkCoffee = new
MilkDecorator(new PlainCoffee());
83             System.out.println("\nDescription: " +
milkCoffee.getDescription());
84             System.out.println("Cost: $" +
milkCoffee.getCost());

```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our

[Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

```
87         Coffee sugarMilkCoffee = new
SugarDecorator(new MilkDecorator(new
PlainCoffee()));
88         System.out.println("\nDescription: " +
sugarMilkCoffee.getDescription());
89         System.out.println("Cost: $" +
sugarMilkCoffee.getCost());
90     }
91 }
```

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.



```
1  Description: Plain Coffee
2  Cost: $2.0
3
4  Description: Plain Coffee, Milk
5  Cost: $2.5
6
7  Description: Plain Coffee, Milk, Sugar
8  Cost: $2.7
```

Advantages of the Decorator Design Pattern

Here are some of the advantages of the decorator pattern:

- **Open-Closed Principle:** The decorator pattern follows the open-closed principle, which states that classes should be open for extension but

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

- **Flexibility:** It allows you to add or remove responsibilities (i.e., behaviors) from objects at runtime. This flexibility makes it easy to create complex object structures with varying combinations of behaviors.
- **Reusable Code:** Decorators are reusable components. You can create a library of decorator classes and apply them to different objects and classes as needed, reducing code duplication.
- **Composition over Inheritance:** Unlike traditional inheritance, which can lead to a deep and inflexible class hierarchy, the decorator pattern uses composition. You can compose objects with different decorators to achieve the desired functionality, avoiding the drawbacks of inheritance, such as tight coupling and rigid hierarchies.
- **Dynamic Behavior Modification:** Decorators can be applied or removed at runtime, providing dynamic behavior modification for objects. This is particularly useful when you need to adapt an object's behavior based on changing requirements or user preferences.
- **Clear Code Structure:** The Decorator pattern promotes a clear and structured design, making it easier for developers to understand how different features and responsibilities are added to objects.

Disadvantages of the Decorator Design Pattern

Here are some of the disadvantages of the Decorator pattern:

- **Complexity:** As you add more decorators to an object, the code can become more complex and harder to understand. The nesting of decorators can make the codebase difficult to navigate and debug, especially when there are many decorators involved.
- **Increased Number of Classes:** When using the Decorator pattern, you often end up with a large number of small, specialized decorator classes. This can lead to a proliferation of classes in your codebase,

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

the correct order, it can lead to unexpected results. Managing the order of decorators can be challenging, especially in complex scenarios.

- **Potential for Overuse:** Because it's easy to add decorators to objects, there is a risk of overusing the Decorator pattern, making the codebase unnecessarily complex. It's important to use decorators judiciously and only when they genuinely add value to the design.
- **Limited Support in Some Languages:** Some programming languages may not provide convenient support for implementing decorators. Implementing the pattern can be more verbose and less intuitive in such languages.

Comment

More info

Advertise with us

Next Article

Facade Method Design Pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented...

11 min read

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our

[Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships...

7 min read

Types of Structural Patterns

Adapter Design Pattern

One structural design pattern that enables the usage of an existing class's interface as an additional interface is the adapter design pattern. To make two incompatible interfaces function together, it...

8 min read

Bridge Design Pattern

The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern. There are 2 parts in Bridge design pattern : AbstractionImplementationThis...

4 min read

Composite Method | Software Design Pattern

Composite Pattern is a structural design pattern that allows you to compose objects into tree

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. I...

9 min read

Facade Method Design Pattern

Facade Method Design Pattern is a part of the Gang of Four design patterns and it is categorized under Structural design patterns. Before we go into the details, visualize a structure. The house is the...

8 min read

Flyweight Design Pattern

The Flyweight design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to...

10 min read

Proxy Design Pattern

The Proxy Design Pattern a structural design pattern is a way to use a placeholder object to control access to another object. Instead of interacting directly with the main object, the client talks to the...

9 min read

3. Behavioural Design Patterns



Corporate & Communications

Address:

A-143, 7th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar
Pradesh (201305)

Registered Address:

K 061, Tower K, Gulshan Vivante
Apartment, Sector 137, Noida,

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.



Advertise with us

Company

About Us
Legal
Privacy Policy
In Media
Contact Us
Advertise with us
GFG Corporate Solution
Placement Training Program
GeeksforGeeks Community

DSA

Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep Jain
All Cheat Sheets

Web Technologies

HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
Bootstrap
Web Design

Computer Science

Operating Systems
Computer Networks

Languages

Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial
Tutorials Archive

Data Science & ML

Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP
Deep Learning

Python Tutorial

Python Programming Examples
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question
Django

DevOps

Git
Linux

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.

Software Development

Software Testing

System Design

High Level Design

Low Level Design

UML Diagrams

Interview Guide

Design Patterns

OOAD

System Design Bootcamp

Interview Questions

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Commerce

World GK

GCP

DevOps Roadmap

Interview Preparation

Competitive Programming

Top DS or Algo for CP

Company-Wise Recruitment Process

Company-Wise Preparation

Aptitude Preparation

Puzzles

GeeksforGeeks Videos

DSA

Python

Java

C++

Web Development

Data Science

CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved

We use cookies to ensure you have the best browsing experience on our website.

By using our site, you acknowledge that you have read and understood our

[Cookie Policy](#) & [Privacy Policy](#). Cookies are not collected in the GeeksforGeeks mobile applications.