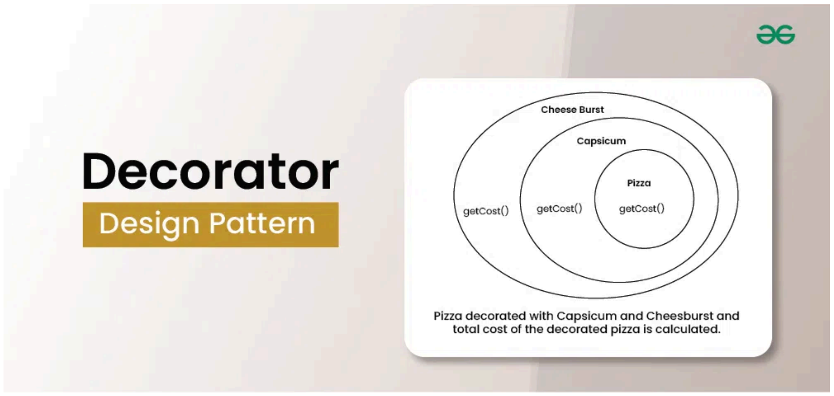
Decorator Design Pattern

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.


Decorator
Design Pattern
Cheese Burst
Capsicum
Pizza
getCost()
getCost()
getCost()
Pizza decorated with Capsicum and Cheesburst and total cost of the decorated pizza is calculated.

Important Topics for 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.*

- 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,

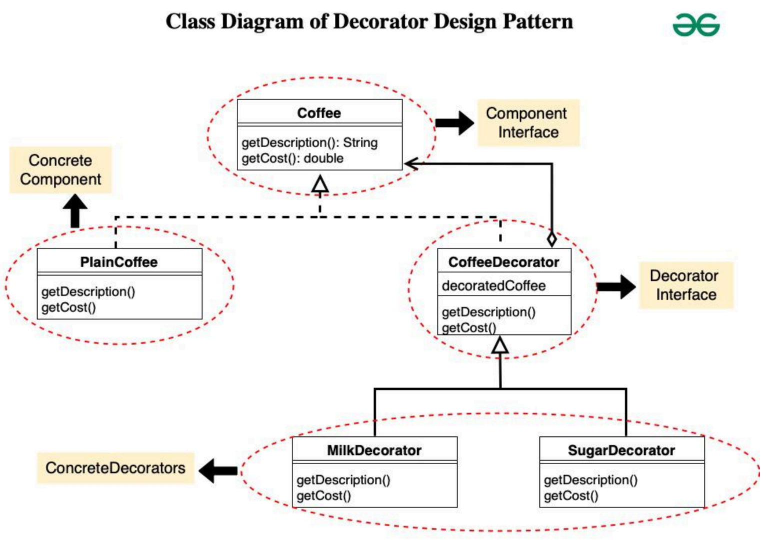
- **Component Interface:** This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.
- **Concrete Component:** These are the basic objects or classes that implement the Component 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:

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.

```
// Coffee.java
```

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.

```java
// PlainCoffee.java
public class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

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.

```
public abstract class CoffeeDecorator
implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee
decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public String getDescription() {
        return
decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

4. ConcreteDecorators(MilkDecorator,SugarDecorator)

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

```java
// MilkDecorator.java
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

// SugarDecorator.java
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
```

Complete Code of the above problem statement:

Below is the complete code of the above problem statement:

```java
// Coffee.java
public interface Coffee {
    String getDescription();
    double getCost();
}

// PlainCoffee.java
public class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
```

```java

// CoffeeDecorator.java
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}

// MilkDecorator.java
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

```

```java
}

// SugarDecorator.java
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        // Plain Coffee
        Coffee coffee = new PlainCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: $" + coffee.getCost());

        // Coffee with Milk
        Coffee milkCoffee = new MilkDecorator(new PlainCoffee());
        System.out.println("\nDescription: " + milkCoffee.getDescription());
        System.out.println("Cost: $" +
        Coffee sugarMilkCoffee = new SugarDecorator(new MilkDecorator(new PlainCoffee()));
        System.out.println("\nDescription: " + sugarMilkCoffee.getDescription());
        System.out.println("Cost: $" + sugarMilkCoffee.getCost());
    }
}
```

```
Description: Plain Coffee
Cost: $2.0

Description: Plain Coffee, Milk
Cost: $2.5

Description: Plain Coffee, Milk, Sugar
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

- **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,

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

9 min read

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

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. Behvioural Design Patterns