Part A
Class B Tech CSE 2nd Year **Sub: Design Pattern Lab**
Aim: Write a program to implement the Decorator Design Pattern
Prerequisite: Basics of Object Oriented Programming
Outcome: To impart knowledge of Design Pattern techniques
Theory: Explain about the Decorator Design Pattern.

Part B
Steps: Write procedure/code here.
Output: Paste Output here.
Observation & Learning: Give your observation and learning about the experiment.
Conclusion: Give your conclusion for the experiment.

Decorator Design Pattern

The Decorator Design Pattern is a structural pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It's useful when you need to add new functionality to objects at runtime, or when you want to keep the system open for extension but closed for modification.

Key Components:

1. **Component**: This is the base interface or abstract class defining the methods that will be implemented by concrete components and decorators.
2. **Concrete Component**: This is the base class implementing the Component interface. It defines the basic behavior to which additional responsibilities can be added.
3. **Decorator**: This is an abstract class extending the Component interface. It has a reference to a Component object and implements the Component interface itself. Decorators have the same interface as the components they are wrapping, allowing them to be used interchangeably.
4. **Concrete Decorators**: These are subclasses of the Decorator class. They add additional functionality to the component by

overriding its methods and calling the wrapped component's methods as needed.

Benefits:

- **Flexibility**: Decorators allow behavior to be added or removed at runtime, providing a flexible way to extend the functionality of objects.
- **Open-Closed Principle**: The Decorator pattern promotes the open-closed principle, allowing classes to be open for extension but closed for modification. New functionality can be added by creating new decorators without modifying existing code.
- **Single Responsibility Principle**: Each decorator class has a single responsibility, making the code easier to understand, maintain, and extend.

Code:

```
// Car interface
interface Car {
    String getDescription();

    double getCost();
}

// Concrete component - BasicCar
class BasicCar implements Car {
    @Override
    public String getDescription() {
        return "Basic Car";
    }

    @Override
    public double getCost() {
        return 20000;
    }
}

// Decorator - CarDecorator
abstract class CarDecorator implements Car {
    protected Car car;

    public CarDecorator(Car car) {
```

```
        this.car = car;
    }
}

// Concrete Decorator - AlloyWheels
class AlloyWheels extends CarDecorator {
    public AlloyWheels(Car car) {
        super(car);
    }

    @Override
    public String getDescription() {
        return car.getDescription() + ", Alloy Wheels";
    }

    @Override
    public double getCost() {
        return car.getCost() + 1000;
    }
}

// Concrete Decorator - LeatherSeats
class LeatherSeats extends CarDecorator {
    public LeatherSeats(Car car) {
        super(car);
    }

    @Override
    public String getDescription() {
        return car.getDescription() + ", Leather Seats";
    }

    @Override
    public double getCost() {
        return car.getCost() + 2000;
    }
}

// Client code
public class Pr10_Decorator {
    public static void main(String[] args) {
        // Creating a basic car
        Car myCar = new BasicCar();
        System.out.println("Description: " + myCar.getDescription());
        System.out.println("Cost: " + myCar.getCost());

        // Adding alloy wheels
        Car myCarWithAlloyWheels = new AlloyWheels(myCar);
```

```
        System.out.println("Description: " +
myCarWithAlloyWheels.getDescription());
        System.out.println("Cost: " + myCarWithAlloyWheels.getCost());

        // Adding leather seats
        Car myCarWithLeatherSeats = new LeatherSeats(myCarWithAlloyWheels);
        System.out.println("Description: " +
myCarWithLeatherSeats.getDescription());
        System.out.println("Cost: " + myCarWithLeatherSeats.getCost());
    }
}
```

Output:

```
Description: Basic Car
Cost: 20000.0
Description: Basic Car, Alloy Wheels
Cost: 21000.0
Description: Basic Car, Alloy Wheels, Leather Seats
Cost: 23000.0
```

Observation & Learning:

1. **Adding Features Dynamically**: We saw how easy it was to add new features to objects on the fly. By using decorators, we could wrap objects with additional functionalities without changing their core structure.
2. **Flexibility and Modular Design**: The pattern showed us how to create flexible and modular code. We could mix and match decorators to create different combinations of behaviors, making our code adaptable to changing requirements.
3. **Reuse and Clarity**: Decorators proved to be reusable across different objects, which helped keep our code concise and clear. Each decorator had a single job, making it easy to understand and maintain.

Conclusion:

In the end, the Decorator pattern gave us a simple and powerful way to enhance the functionality of objects without making them complicated. It encouraged us to keep our code modular, flexible, and easy to understand. We'll definitely keep it in mind for future projects to make our code more versatile and maintainable.