20__17-09-2022

Saturday, 17 September 2022 8:07 PM

Structural design pattern

Decorator design pattern

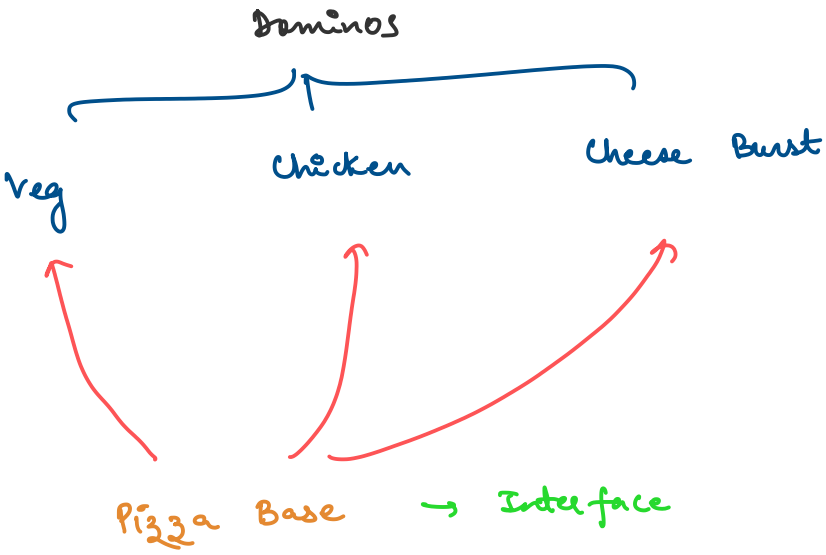
↳ To decorate an object at the runtime

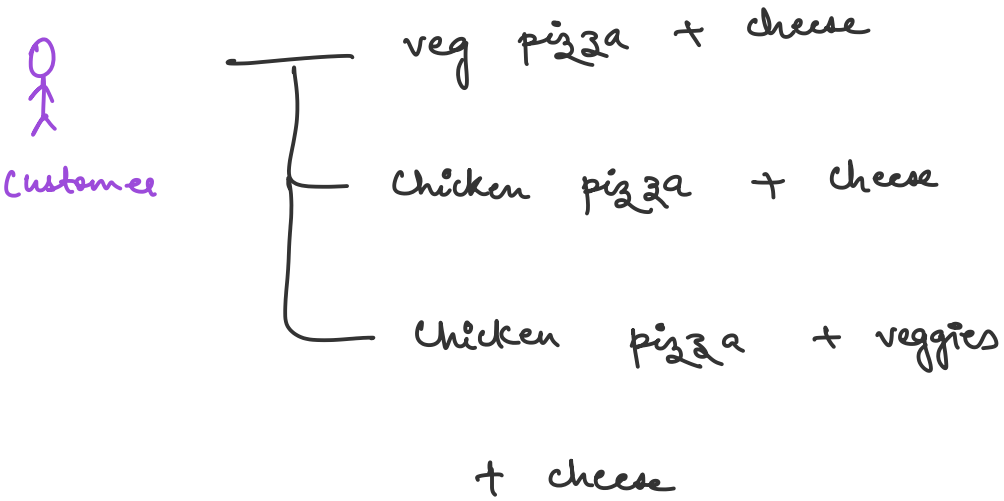
Eg- Decorator at home

↓

Giving instructions at the time of his decoration (Run time)

Eg





Builder design patteen

↳ objects

↳ we cannot change these objects later

↓

Addressing object complexity

at compile time

Decorator design pattern

↳ object

↳ which can be changed

based upon the

requirement.

↓

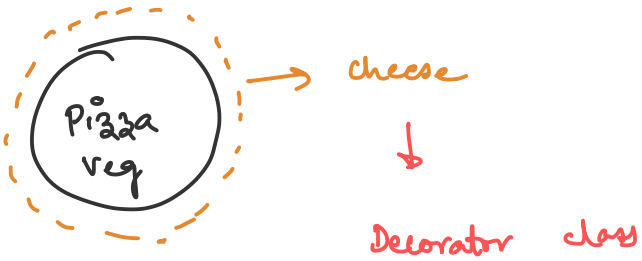
Addressing the complexity

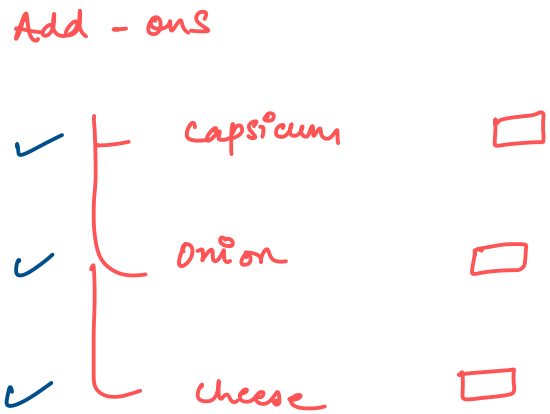
of object at runtime.

↓

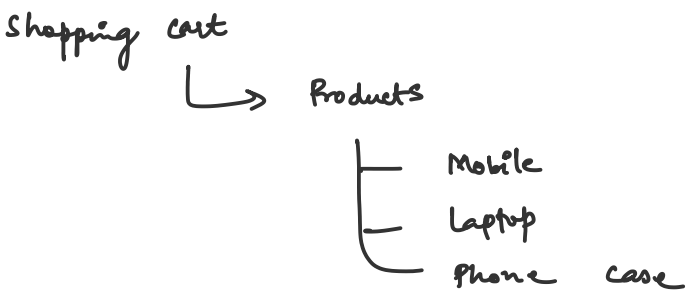
Decorating the object

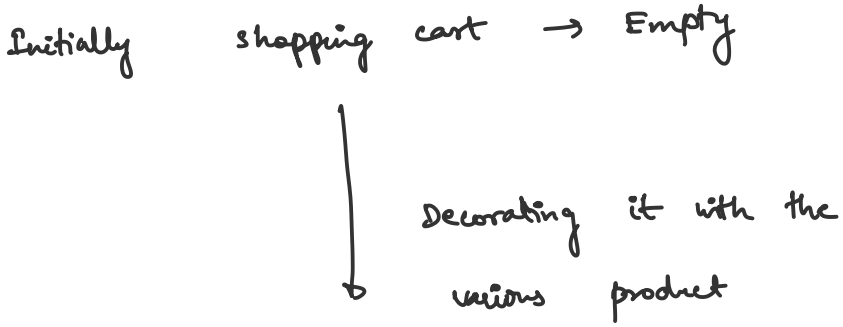
with the new parameters


Pizza
veg
Cheese
Decorator class


Add - ons
Capsicum
Onion
Cheese

Another eg -


Shopping cart
Products
Mobile
Laptop
Phone case



Definition

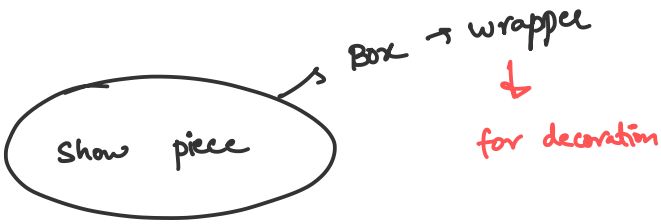
Decorator design pattern states that we need to attach additional responsibilities to an object dynamically.

Decorator provides a flexible alternative to sub-classing for extending flexibility.

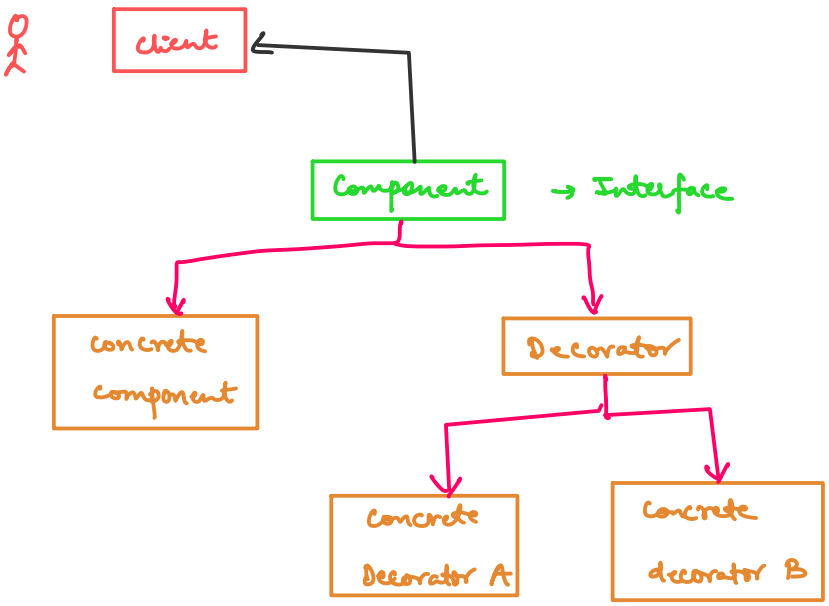
In other words, the decorator pattern uses composition instead of inheritance to extend the functionality of an

object at runtime.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure.



This design pattern acts as a wrapper to the existing classes and thats why it is also called as wrapper.



Client

↓

Customer

↓

using our appn | for whom appn is designed

↓

Client will be using the component interface to interact with the objects.

Component

↓

Interface

↓

Defines interface for the objects that can have responsibilities added to them dynamically.

Concrete component

↓

Defines an object to which additional responsibilities can be attached.

Decorator

↓

Decorator will be maintaining a reference to a component object and defines an interface that conforms to the component's interface.

Concrete decorator

↓

Adds responsibilities to the component.

Advantages —

1. It provides greater flexibility than static inheritance.

2. It enhances the extendability of the object because changes are made by coding new classes.

3. It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Implementation guidelines

We need to use decorator design pattern when—

1. When you want to add responsibilities

to an object that you may want to change in future.

2. Extending functionality by sub classing is no longer practical.

3. Class definition may be hidden or unavailable for sub-classing.

4. When you want to be transparent and want to add responsibilities to objects without affecting the other objects.

Steps –

1. Create an interface.

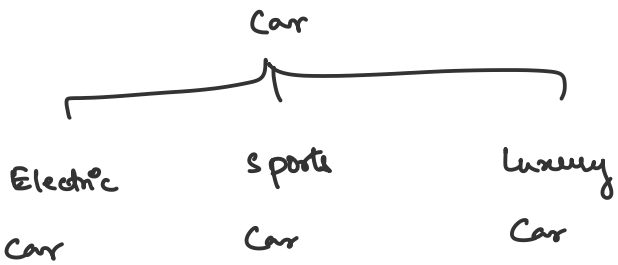
2. Create concrete component to implement

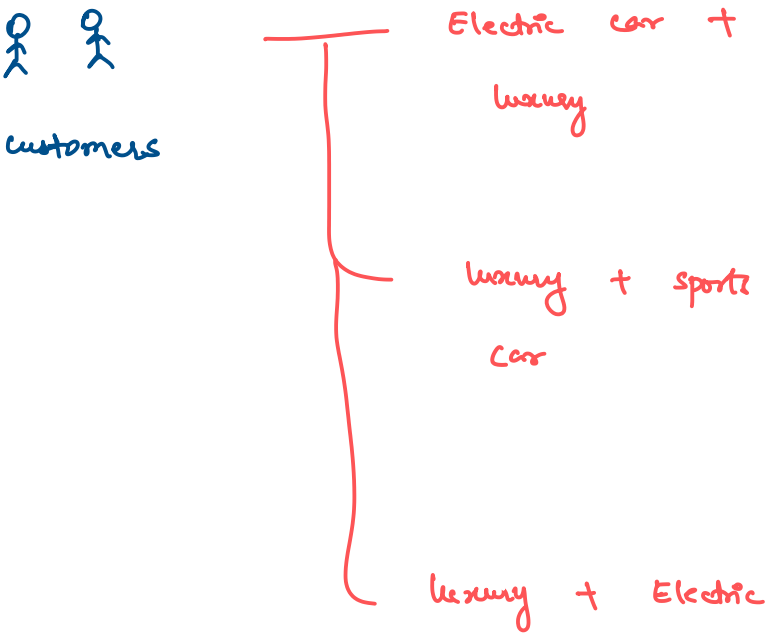
the interface.

3. Create a decorator class to decorate the object.

4. Create client class.

Eg





+ Sports Car

Basic car

↳ Adding extra layer

↓

Decorator class.

Luxury car → Luxury Car Decorator

Electric " → Electric " "

Sports " → Sports " "

```
package DecoratorDesignPattern;

public interface ICar {
   public void manufactureCar();
}
```

```
package DecoratorDesignPattern;

public class BasicCar implements ICar {
   @Override
   public void manufactureCar() {
      System.out.println("Manufacturing basic car.");
   }
}
```

```
package DecoratorDesignPattern;

//Basic car decorator
public class CarDecorator implements ICar {

   protected ICar car;

   public CarDecorator(ICar car) {
      this.car = car;
   }

   @Override
   public void manufactureCar() {
      this.car.manufactureCar();
   }
}
```

```
package DecoratorDesignPattern;

public class ElectricCarDecorator extends CarDecorator {
   public ElectricCarDecorator(ICar car) {
      super(car);

   }

   @Override
   public void manufactureCar() {
      //first basic car will be created
      super.manufactureCar();
      System.out.println("Including the features of electric car.");
   }
}
```

```
package DecoratorDesignPattern;
```

```
public class LuxuryCarDecorator extends CarDecorator {
   public LuxuryCarDecorator(ICar car) {
      super(car);
   }

   @Override
   public void manufactureCar() {
      //first basic car will be created
      super.manufactureCar();
      System.out.println("Including the features of luxury car.");
   }
}
```

```
package DecoratorDesignPattern;

public class SportsCarDecorator extends CarDecorator {
   public SportsCarDecorator(ICar car) {
      super(car);
   }

   @Override
   public void manufactureCar() {
      //first basic car will be created
      super.manufactureCar();
      System.out.println("Including the features of sports car.");
   }
}
```

```
package DecoratorDesignPattern;

public class Program {
   public static void main(String[] args) {
      //Demand for sports car
      ICar sportsCar = new SportsCarDecorator(new BasicCar());
      sportsCar.manufactureCar();
      System.out.println();
```

```
        //Demand for electric car
        ICar electricCar = new ElectricCarDecorator(new BasicCar());
        electricCar.manufactureCar();
        System.out.println();

        //Demand for sports electric car
        ICar sportsElectricCar = new ElectricCarDecorator(new SportsCarDecorator(new BasicCar(
        sportsElectricCar.manufactureCar();
        System.out.println();

        //Demand for luxury sports electric car
        ICar luxurySportsElectricCar = new ElectricCarDecorator(new SportsCarDecorator(new
LuxuryCarDecorator(new BasicCar())));
        luxurySportsElectricCar.manufactureCar();
    }
}
```

Output:

```
Manufacturing basic car.
Including the features of sports car.

Manufacturing basic car.
Including the features of electric car.

Manufacturing basic car.
Including the features of sports car.
Including the features of electric car.

Manufacturing basic car.

Including the features of luxury car.
Including the features of sports car.
Including the features of electric car.
```