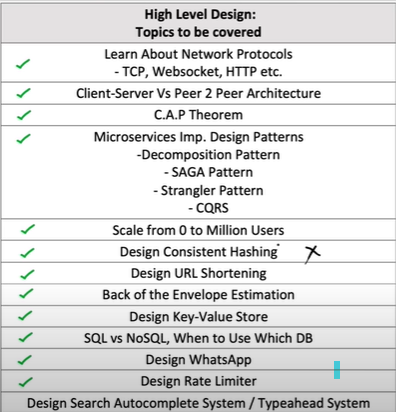
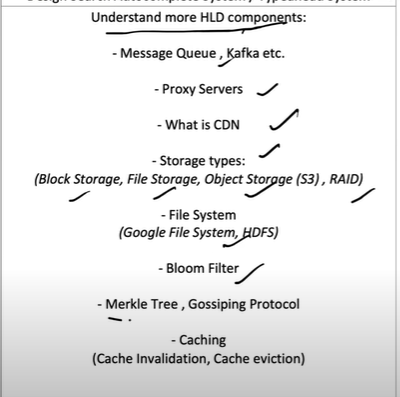
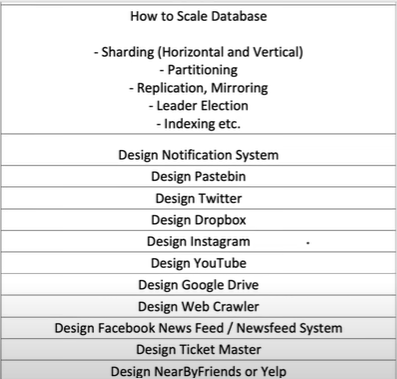
Concept && Coding: System Design RoadMap

|  |  |
| --- | --- |
| **Low Level Design:  Patterns To Be Covered** | **Low Level Design:  Popular Interview Question To Be Covered** |
| Strategy Pattern | S.O.L.I.D Principle |
| Observe Pattern | Design Notify- Me Button Functionality |
| Decorator Pattern | Design Pizza Billing System |
| Factory Pattern | Design Parking Lot |
| Abstract Factory Pattern | Design Snake and Ladder Game |
| Chain of Responsibility Pattern | Design Elevator Pattern |
| Proxy Pattern | Design Car Rental System |
| Null Object Pattern | Design Logging System |
| State Pattern | Design Tic Tac Toe Game |
| Composite Pattern | Design BookMyShow And Concurrency Handling |
| Adaptor Pattern | Design Vending machine |
| Singleton Pattern | Design ATM |
| Builder Pattern | Design Chess Game |
| Prototype Pattern | Design File System |
| Bridge Pattern | Design splitWise |







**Design Pattern**

**What are Design Patterns?**

Design patterns are reusable solutions to common problems in software design. They represent best practices used by experienced object-oriented software developers. Design patterns provide a standard terminology and are specific to particular scenarios.

**Key Characteristics of Design Patterns**

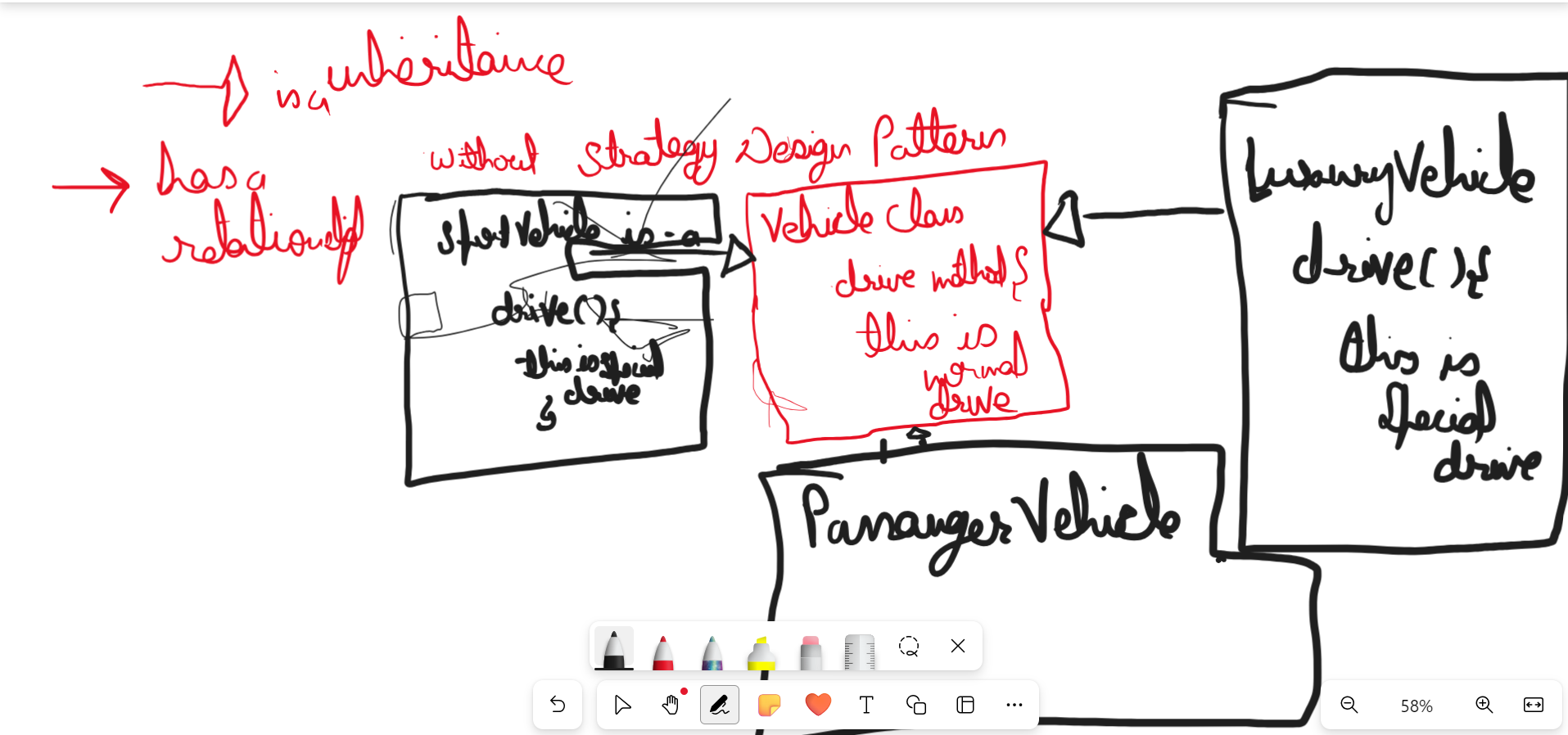
* **Reusability**: Patterns can be applied to different projects and problems, saving time and effort in solving similar issues.
* **Standardization**: They provide a shared language and understanding among developers, facilitating communication and collaboration.
* **Efficiency**: By using established patterns, developers can avoid reinventing the wheel, leading to faster and more reliable development.
* **Flexibility**: Patterns are abstract solutions that can be adapted to fit various contexts and requirements.

**Why Learn Design Patterns?**

There are several reasons to learn design patterns:

1. **Improve Code Quality**: Design patterns help in creating code that is easier to understand, maintain, and extend. They promote best practices and provide solutions that have been tested and proven effective.
2. **Enhance Problem-Solving Skills**: Learning design patterns equips developers with a book of standard solutions to common problems. This enables them to quickly and effectively address similar challenges in various projects.
3. **Promote Reusability and Efficiency:**By applying design patterns, developers can create reusable components that can be used across multiple projects. This reduces redundancy and saves development time.
4. **Learn from Experts: Explanation**: Design patterns are derived from the collective experience of skilled developers and architects. Learning these patterns allows developers to benefit from the wisdom and insights of industry experts.

**Strategy Design Pattern**

****

**The Problem with this approach is every child class is need to override the driver method from the Vehicle class if they want the special drive.**

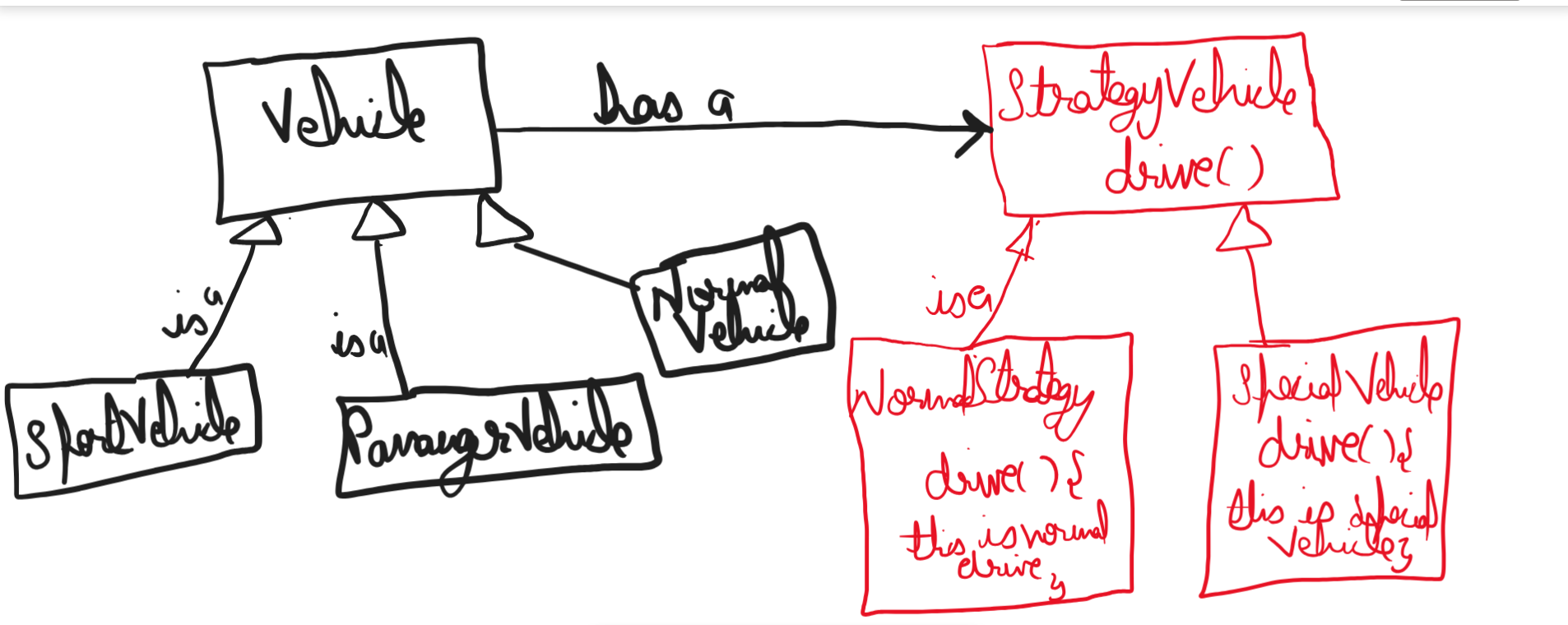
package com.automation.designPattern.withoutStrategyDP;  
  
public class Vehicle {  
  
 public Vehicle() {  
 }  
  
 public void driver(){  
 System.*out*.println("The is normal driver strategy");  
  
 }  
}

package com.automation.designPattern.withoutStrategyDP;  
  
public class SportVehicle extends Vehicle{  
  
 public void driver(){  
 System.*out*.println("The is special driver strategy");  
  
 }  
}

package com.automation.designPattern.withoutStrategyDP;  
  
public class PassangerVehicle extends Vehicle{  
}

package com.automation.designPattern.withoutStrategyDP;  
  
public class GoodsVehicle extends Vehicle {  
  
 public void driver(){  
 System.*out*.println("The is special driver strategy");  
  
 }  
}

**WithStrategyDesign Pattern**

****

package com.automation.designPattern.withStrategyDP.strategy;  
  
public interface StrategyDrive {  
  
 public void drive();  
}

package com.automation.designPattern.withStrategyDP.strategy;  
  
public class SpecialStrategy implements StrategyDrive{  
 @Override  
 public void drive() {  
 System.*out*.println("This is special drive strategy");  
 }  
}

package com.automation.designPattern.withStrategyDP.strategy;  
  
public class NormalStrategy implements StrategyDrive{  
 @Override  
 public void drive() {  
 System.*out*.println("This is normal drive strategy");  
 }  
}

package com.automation.designPattern.withStrategyDP;  
  
import com.automation.designPattern.withStrategyDP.strategy.StrategyDrive;  
  
public class Vehicle {  
  
 StrategyDrive strategyDrive;  
  
 public Vehicle(StrategyDrive strategyDrive) {  
 this.strategyDrive = strategyDrive;  
 }  
  
 public void drive(){  
 strategyDrive.drive();  
 }  
}

package com.automation.designPattern.withStrategyDP;  
  
import com.automation.designPattern.withStrategyDP.strategy.SpecialStrategy;  
import com.automation.designPattern.withStrategyDP.strategy.StrategyDrive;  
  
public class SportVehicle extends Vehicle {  
  
  
 public SportVehicle() {  
 super(new SpecialStrategy());  
 }  
}

package com.automation.designPattern.withStrategyDP;  
  
import com.automation.designPattern.withStrategyDP.strategy.NormalStrategy;  
import com.automation.designPattern.withStrategyDP.strategy.StrategyDrive;  
  
public class PassangerVehicle extends Vehicle {  
 public PassangerVehicle() {  
 super(new NormalStrategy());  
 }  
}

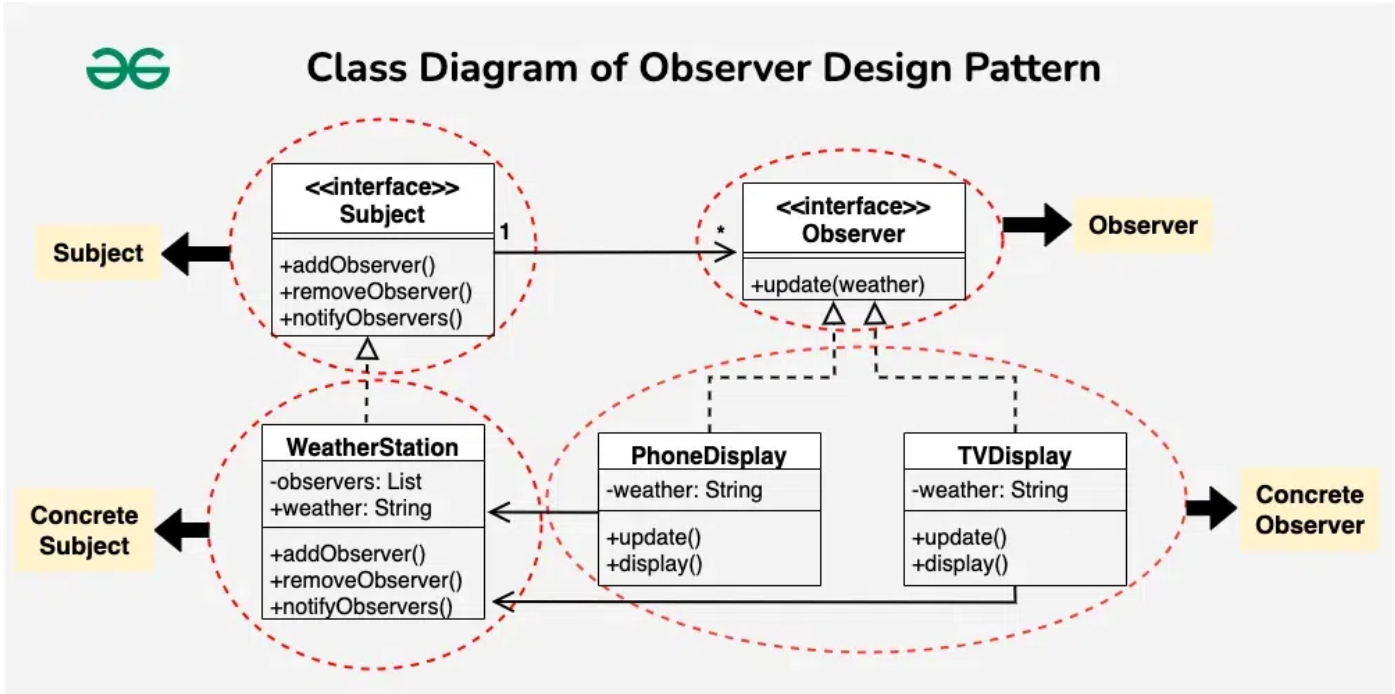
package com.automation.designPattern.withStrategyDP;  
  
import com.automation.designPattern.withStrategyDP.strategy.NormalStrategy;  
import com.automation.designPattern.withStrategyDP.strategy.SpecialStrategy;  
import com.automation.designPattern.withStrategyDP.strategy.StrategyDrive;  
  
public class GoodsVehicle extends Vehicle {  
  
 public GoodsVehicle() {  
 super(new SpecialStrategy());  
 }  
}

package com.automation.designPattern.withStrategyDP;  
  
import com.automation.designPattern.withStrategyDP.strategy.StrategyDrive;  
import cucumber.api.java.tr.Ve;  
  
public class MainVehicleClass {  
  
 public static void main(String[] args) {  
  
 Vehicle drive= new PassangerVehicle();  
 drive.drive();  
 }  
}

**Observer Design Pattern, Walmart Design**

**Has a**

**-----Is a--------**

****

package com.automation.designPattern.ObserverDP.observer;  
  
public interface Observer {  
  
 void update(String weather);  
}

package com.automation.designPattern.ObserverDP.observer;  
  
public class LaptopDisplay implements Observer{  
  
 public String weather;  
 @Override  
 public void update(String weather) {  
 this.weather=weather;  
 display();  
 }  
  
 public void display(){  
 System.*out*.println("Laptop Display: Weather updated - " + weather);  
 }  
}

package com.automation.designPattern.ObserverDP.observer;  
  
public class PhoneDisplay implements Observer{  
  
 public String weather;  
 @Override  
 public void update(String weather) {  
 this.weather=weather;  
 display();  
 }  
  
 public void display(){  
 System.*out*.println("Phone Display: Weather updated - " + weather);  
 }  
}

package com.automation.designPattern.ObserverDP.subject;  
  
import com.automation.designPattern.ObserverDP.observer.Observer;  
  
public interface Subject {  
  
 void addObserver(Observer observer);  
  
 void remove(Observer observer);  
  
 void notifyObservers();  
}

package com.automation.designPattern.ObserverDP.subject;  
  
import com.automation.designPattern.ObserverDP.observer.Observer;  
  
import java.sql.Array;  
import java.util.ArrayList;  
import java.util.List;  
  
public class WeatherStation implements Subject{  
  
 List<Observer> observerList= new ArrayList();  
  
 public String weather;  
 @Override  
 public void addObserver(Observer observer) {  
 observerList.add(observer);  
 }  
  
 @Override  
 public void remove(Observer observer) {  
 observerList.remove(observer);  
 }  
  
 @Override  
 public void notifyObservers() {  
  
 for(Observer observer: observerList){  
 observer.update(weather);  
 }  
 }  
  
 public void setWeather(String newWeather){  
 this.weather=newWeather;  
 notifyObservers();  
 }  
}

package com.automation.designPattern.ObserverDP.subject;  
  
import com.automation.designPattern.ObserverDP.observer.Observer;  
import com.automation.designPattern.ObserverDP.observer.PhoneDisplay;  
  
public class WeatherApp {  
  
 public static void main(String[] args) {  
  
 WeatherStation weatherStation = new WeatherStation();  
  
 Observer observer = new PhoneDisplay();  
 weatherStation.addObserver(observer);  
 weatherStation.setWeather("Rainy");  
 }  
}

**Decorator Design Pattern**

The Decorator design pattern is a [structural pattern](https://www.geeksforgeeks.org/structural-design-patterns/) used in object-oriented programming to add new functionality to objects dynamically without altering their structure.

* 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.

**Components of Decorator Method Design Pattern in Java**

Below are the components of decorator method design pattern in java:

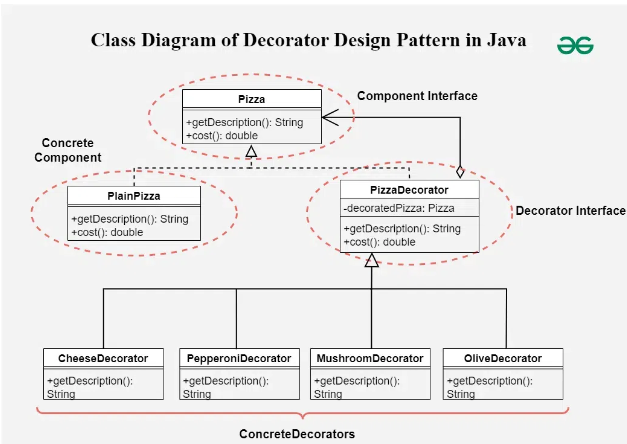
* **Component Interface:** An interface or abstract class that defines the core functionality. This is the base type for both concrete components and decorators.
* **Concrete Component:** A class that implements the Component interface and provides the basic behavior.
* **Decorator:** An abstract class that implements the Component interface and has a reference to a Component object. This class defines the interface for the decorators and includes a reference to a Component instance.
* **Concrete Decorators:** Classes that extend the Decorator class and add additional behavior to the Component.

**Example of Decorator Method Design Pattern in Java**

Below is the **problem statement** to understand decorator method design pattern in Java:

*Imagine a pizza shop where customers can customize their pizzas with various toppings like cheese, pepperoni, mushrooms, and olives. The goal is to create a flexible system that allows you to dynamically add any combination of toppings to a base pizza without modifying the existing pizza classes or creating numerous subclasses.*

* The Decorator pattern helps solve this problem by allowing you to extend the behavior of a base pizza object dynamically.
* You can create decorators for each topping, which will add its specific functionality to the base pizza.

****

package com.automation.designPattern.decoratorDP;  
  
public interface PizzaBase {  
  
 public double cost();  
  
 public String getDescription();  
}

package com.automation.designPattern.decoratorDP;  
  
public class PlainPizzaBase implements PizzaBase{  
 @Override  
 public double cost() {  
 return 4.5;  
 }  
  
 @Override  
 public String getDescription() {  
 return "This is Base Pizza";  
 }  
}

package com.automation.designPattern.decoratorDP;  
  
public class PizzaDecorator implements PizzaBase{  
  
 PizzaBase pizzaBase;  
  
 public PizzaDecorator(PizzaBase pizzaBase) {  
 this.pizzaBase = pizzaBase;  
 }  
  
 @Override  
 public double cost() {  
 return pizzaBase.cost();  
 }  
  
 @Override  
 public String getDescription() {  
 return pizzaBase.getDescription();  
 }  
}

package com.automation.designPattern.decoratorDP;  
  
public class ExtraCheeseDecorator extends PizzaDecorator{  
 public ExtraCheeseDecorator(PizzaBase pizzaBase) {  
 super(pizzaBase);  
 }  
  
 @Override  
 public double cost() {  
 return pizzaBase.cost() + 10.5;  
 }  
  
 @Override  
 public String getDescription() {  
 return pizzaBase.getDescription() + "Extraa cheese";  
 }  
}

package com.automation.designPattern.decoratorDP;  
  
public class PepperoniDecorator extends PizzaDecorator {  
  
 public PepperoniDecorator(PizzaBase pizzaBase) {  
 super(pizzaBase);  
 }  
  
 @Override  
 public double cost() {  
 return pizzaBase.cost() + 18.5;  
 }  
  
 @Override  
 public String getDescription() {  
 return pizzaBase.getDescription() + "Pepperoni";  
 }  
}

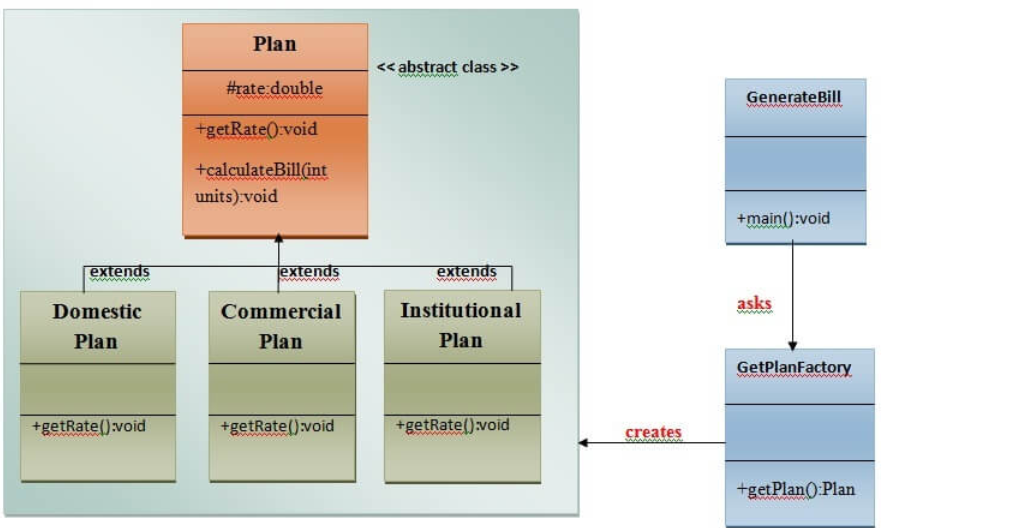
package com.automation.designPattern.decoratorDP;  
  
public class MainClass {  
  
 public static void main(String[] args) {  
  
  
 PizzaBase pizza= new PlainPizzaBase();  
 System.*out*.println(pizza.getDescription()+ "$" + pizza.cost());  
  
 pizza = new ExtraCheeseDecorator(pizza);  
 System.*out*.println(pizza.getDescription() + " $" + pizza.cost());  
 }  
}

**Factory Design Pattern**

1. **Factory Design pattern is a creational design pattern**
2. **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

#### Advantage of Factory Design Pattern

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

****

package com.automation.designPattern.factoryDP;  
  
public abstract class Plain {  
  
 protected double rate;  
 abstract void getRate();  
  
 public void calculateBill(int units){  
 System.*out*.println(units\*rate);  
 }  
  
}

package com.automation.designPattern.factoryDP;  
  
public class InstitutionalPlan extends Plain{  
 @Override  
 void getRate() {  
 rate=3.5;  
 }  
}

package com.automation.designPattern.factoryDP;  
  
public class DomesticPlan extends Plain{  
 @Override  
 void getRate() {  
 rate=5.5;  
 }  
}

package com.automation.designPattern.factoryDP;  
  
public class CommercialPlan extends Plain{  
 @Override  
 void getRate() {  
 rate=7.50;  
 }  
}

package com.automation.designPattern.factoryDP.factoryclass;  
  
import com.automation.designPattern.factoryDP.CommercialPlan;  
import com.automation.designPattern.factoryDP.DomesticPlan;  
import com.automation.designPattern.factoryDP.InstitutionalPlan;  
import com.automation.designPattern.factoryDP.Plain;  
  
public class PlainFactory {  
  
 public Plain getPlain(String plain){  
 if(plain==null){  
 return null;  
 } else if (plain.equals("DOMESTIC")) {  
 return new DomesticPlan();  
 } else if (plain.equals("COMMERCIAL")) {  
 return new CommercialPlan();  
 } else {  
 return new InstitutionalPlan();  
 }  
  
  
 }  
}

package com.automation.designPattern.factoryDP.factoryclass;  
  
import com.automation.designPattern.factoryDP.CommercialPlan;  
import com.automation.designPattern.factoryDP.DomesticPlan;  
import com.automation.designPattern.factoryDP.InstitutionalPlan;  
import com.automation.designPattern.factoryDP.Plain;  
  
public class PlainFactory {  
  
 public Plain getPlain(String plain){  
 if(plain==null){  
 return null;  
 } else if (plain.equals("DOMESTIC")) {  
 return new DomesticPlan();  
 } else if (plain.equals("COMMERCIAL")) {  
 return new CommercialPlan();  
 } else {  
 return new InstitutionalPlan();  
 }  
  
  
 }  
}