Design Pattern and Thinking Lab

Creational Design Pattern

Jaivik Jariwala

21BCP004 | Divison - 1 | Group - 1

Aim :

To Learn , Understand and Implement Creational Design Pattern

Introduction:

Creational Design Patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic idea is to hide the complexity of object creation and keep the system decoupled, creating objects that follow the Single Responsibility Principle and the Open-Closed Principle.

There are several types of Creational Design Patterns, including:

Singleton: Ensures that a class has only one instance while providing a global point of access to this instance.

Abstract Factory: Provides an interface for creating families of related objects without specifying their concrete classes.

Builder: Separates object construction from its representation, allowing the same construction process to create different representations.

Factory Method: Defines an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Prototype: A fully initialized instance to be copied or cloned.

Each of these design patterns serves a specific purpose and can be used in different scenarios depending on the specific needs of the system being developed.

Factory Method:

The factory method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. In other words, it allows you to define a generic interface for creating objects, but lets subclasses decide which class to instantiate.

Example-1

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark!");
    }
}

class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}

class AnimalFactory {
    public Animal getAnimal(String animalType) {
        if (animalType == null) {
            return null;
        }
        if (animalType.equalsIgnoreCase("DOG")) {
            return new Dog();
        } else if (animalType.equalsIgnoreCase("CAT")) {
            return new Cat();
        }
        return null;
    }
}
```

Example-2

```
interface Shape {
    void draw();
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
}
```

Abstract Factory Method:

The Abstract Factory pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern can be useful in cases where you need to create groups of objects that are related to each other in some way.

Example-1

```
interface Animal {
    void speak();
}

class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("Woof!");
    }
}

class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("Meow!");
    }
}

interface AnimalFactory {
    Animal createAnimal();
}

class DogFactory implements AnimalFactory {
    @Override
    public Animal createAnimal() {
        return new Dog();
    }
}

class CatFactory implements AnimalFactory {
    @Override
    public Animal createAnimal() {
        return new Cat();
    }
}

class Client {
    public static void main(String[] args) {
        AnimalFactory factory = null;
```

```
        String animalType = args[0];
        if (animalType.equalsIgnoreCase("dog")) {
            factory = new DogFactory();
        } else if (animalType.equalsIgnoreCase("cat")) {
            factory = new CatFactory();
        }
        Animal animal = factory.createAnimal();
        animal.speak();
    }
}
```

Example-2

```
interface Button {
    void paint();
}

class WindowsButton implements Button {
    @Override
    public void paint() {
        System.out.println("You have created WindowsButton.");
    }
}

class OSXButton implements Button {
    @Override
    public void paint() {
        System.out.println("You have created OSXButton.");
    }
}

interface GUIFactory {
    Button createButton();
}

class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}

class OSXFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new OSXButton();
    }
}

class Application {
    public void paint(GUIFactory factory) {
        Button button = factory.createButton();
        button.paint();
    }
}

class Client {
    public static void main(String[] args) {
```

```
        Application app = new Application();
        GUIFactory factory = null;
        String osName = System.getProperty("os.name").toLowerCase();
        if (osName.contains("windows")) {
            factory = new WindowsFactory();
        } else {
            factory = new OSXFactory();
        }
        app.paint(factory);
    }
}
```

Prototype Method:

The prototype method is a creational design pattern that is used in software engineering to create objects based on a blueprint of an existing object through cloning. The prototype pattern involves implementing a prototype interface, which specifies a clone method that creates a new object that is a copy of the original object. This is useful when the creation of a new object is expensive in terms of time and resources and when the user needs to create many instances of the same object.

Example-1

```
interface Animal extends Cloneable {
    Animal makeCopy();
}

class Sheep implements Animal {
    private String name;

    public Sheep(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public Animal makeCopy() {
        Sheep sheep = null;
        try {
            sheep = (Sheep) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return sheep;
    }

    @Override
    public String toString() {
        return "Sheep{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

Example-2

```
interface PrototypeCapable extends Cloneable {
    PrototypeCapable clone();
}

class Movie implements PrototypeCapable {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public PrototypeCapable clone() {
        Movie movie = null;
        try {
            movie = (Movie) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return movie;
    }
}
```

Builder Method:

The builder pattern is a creational design pattern that is used to separate the construction of a complex object from its representation. The idea behind this pattern is to have a builder class that takes care of constructing an object step by step, instead of having the client code construct the object directly.

Example-1

```
class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;

    private Computer(ComputerBuilder builder) {
        this.CPU = builder.CPU;
        this.GPU = builder.GPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    public static class ComputerBuilder {
        private String CPU;
        private String GPU;
        private int RAM;
        private int storage;

        public ComputerBuilder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public ComputerBuilder setGPU(String GPU) {
            this.GPU = GPU;
            return this;
        }

        public ComputerBuilder setRAM(int RAM) {
            this.RAM = RAM;
            return this;
        }

        public ComputerBuilder setStorage(int storage) {
            this.storage = storage;
```

```
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    public String getCPU() {
        return CPU;
    }

    public String getGPU() {
        return GPU;
    }

    public int getRAM() {
        return RAM;
    }

    public int getStorage() {
        return storage;
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", GPU=" + GPU + ", RAM=" + RAM + ",
storage=" + storage + "]";
    }
}

class BuilderExample {
    public static void main(String[] args) {
        Computer computer = new Computer.ComputerBuilder().setCPU("Intel Core
i7").setGPU("Nvidia RTX 3080").setRAM(16)
                .setStorage(1000).build();
        System.out.println(computer);
    }
}
```

Example-2

```
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) {
        this.dough = dough;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public void setTopping(String topping) {
        this.topping = topping;
    }

    @Override
    public String toString() {
        return "Pizza [dough=" + dough + ", sauce=" + sauce + ", topping=" +
topping + "]";
    }
}

class PizzaBuilder {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public PizzaBuilder setDough(String dough) {
        this.dough = dough;
        return this;
    }

    public PizzaBuilder setSauce(String sauce) {
        this.sauce = sauce;
        return this;
    }
```

```
    public PizzaBuilder setTopping(String topping) {
        this.topping = topping;
        return this;
    }

    public Pizza build() {
        Pizza pizza = new Pizza();
        pizza.setDough(dough);
        pizza.setSauce(sauce);
        pizza.setTopping(topping);
        return pizza;
    }
}

class Cook {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pizzaBuilder) {
        this.pizzaBuilder = pizzaBuilder;
    }

    public Pizza getPizza() {
        return pizzaBuilder.build();
    }

    public void constructPizza() {
        pizzaBuilder.setDough("cross");
        pizzaBuilder.setSauce("mild");
        pizzaBuilder.setTopping("ham+pineapple");
    }
}

class BuilderExample {
    public static void main(String[] args) {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new PizzaBuilder();
        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();

        Pizza pizza = cook.getPizza();
        System.out.println("Pizza built: " + pizza);
    }
}
```

Singleton Method:

The Singleton pattern is a creational design pattern that ensures a class has only one instance, while providing a global point of access to this instance. The Singleton pattern can be useful in cases where you need to make sure that a class has only one instance throughout the lifetime of your application.

Example-1

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}


public class SingletonExample {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

Example-2

```
public class ChatServer {
    private static ChatServer instance;

    private ChatServer() {}

    public static ChatServer getInstance() {
        if (instance == null) {
            instance = new ChatServer();
        }
        return instance;
    }

    public void startServer() {
        // code to start the chat server
    }

    public void stopServer() {
        // code to stop the chat server
    }
}
```

GitHub:

https://github.com/Jaivik-Jariwala/Design-Pattern-with-JAVA-

References:

- *Javatpoint*
- *Chatgpt*