COMMONLY USED

DESIGN PATTERNS

CREATIONAL PATTERNS:

Creational patterns focus on how objects are created, aiming to decouple the instantiation process from the code that uses these objects. This ensures flexibility and control over object creation. Examples include **Singleton**, **Factory**, and **Builder** patterns, which simplify object creation in complex situations without exposing underlying logic.

SINGLETON PATTERN:

- Ensures that a class has only one instance while providing a global point of access to it.
- Useful in scenarios like logging, caching, or database connections.
- The key is to make the constructor private and provide a static method to return the instance.

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

    private Singleton() { }

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

FACTORY PATTERN:

- Used when we want to create objects without specifying the exact class of the object that will be created.
- Useful in scenarios where the type of object needed can vary based on input.

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

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

public class ShapeFactory {
    public static Shape getShape(String shapeType) {
        if (shapeType.equals("Circle")) {
            return new Circle();
        }
        return null;
    }
}
```

BUILDER PATTERN:

- Used to construct a complex object step by step, allowing for more control and flexibility in object creation.
- Especially useful when an object needs to be created with multiple optional parameters.
- The pattern decouples the object creation logic from its representation.

```java
public class House {
    private String foundation;
    private String structure;
    private String roof;
    private boolean hasGarage;

    private House(HouseBuilder builder) {
        this.foundation = builder.foundation;
        this.structure = builder.structure;
        this.roof = builder.roof;
        this.hasGarage = builder.hasGarage;
    }

    public static class HouseBuilder {
        private String foundation;
        private String structure;
        private String roof;
        private boolean hasGarage;

        public HouseBuilder setFoundation(String foundation) {
            this.foundation = foundation;
            return this;
        }

        public HouseBuilder setStructure(String structure) {
            this.structure = structure;
            return this;
        }
```

BUILDER PATTERN:

```
    public HouseBuilder setRoof(String roof) {
        this.roof = roof;
        return this;
    }

    public HouseBuilder setGarage(boolean hasGarage) {
        this.hasGarage = hasGarage;
        return this;
    }

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

@Override
public String toString() {
    return "House [foundation=" + foundation + ", structure=" + structure +
            ", roof=" + roof + ", hasGarage=" + hasGarage + "]";
    }
}
```

BUILDER PATTERN:

```java
public class Main {
    public static void main(String[] args) {
        House house = new House.HouseBuilder()
            .setFoundation("Concrete")
            .setStructure("Wood")
            .setRoof("Tile")
            .setGarage(true)
            .build();

        System.out.println(house);
    }
}
```

STRUCTURAL PATTERNS:

Structural patterns deal with the composition of objects and classes to form larger structures. These patterns focus on simplifying relationships between objects to improve flexibility and efficiency. Examples include **Adapter**, **Decorator**, and **Facade** patterns, which help organize systems with minimal coupling.

ADAPTER PATTERN:

- Allows incompatible interfaces to work together.
- Acts as a bridge between two interfaces, converting one interface into another that a client expects.

```java
// Existing class
public class VgaDisplay {
    public void displayOnVga() {
        System.out.println("Displaying via VGA");
    }
}

// Adapter interface
public interface Hdmi {
    void displayOnHdmi();
}

// Adapter class
public class VgaToHdmiAdapter implements Hdmi {
    private VgaDisplay vgaDisplay;

    public VgaToHdmiAdapter(VgaDisplay vgaDisplay) {
        this.vgaDisplay = vgaDisplay;
    }

    @Override
    public void displayOnHdmi() {
        vgaDisplay.displayOnVga(); // Converting VGA to HDMI
    }
}
```

ADAPTER PATTERN:

```
// Client code
public class Main {
    public static void main(String[] args) {
        VgaDisplay vgaDisplay = new VgaDisplay();
        Hdmi hdmiDisplay = new VgaToHdmiAdapter(vgaDisplay);
        hdmiDisplay.displayOnHdmi();
    }
}
```

DECORATOR PATTERN:

- Allows behavior to be added to an object dynamically, without modifying its code.
- Useful when you want to extend the functionalities of classes in a flexible way.

```java
public interface Car {
    void assemble();
}

public class BasicCar implements Car {
    public void assemble() {
        System.out.println("Basic Car.");
    }
}

public class CarDecorator implements Car {
    protected Car car;

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

    public void assemble() {
        this.car.assemble();
    }
}
```

DECORATOR PATTERN:

```
public class SportsCar extends CarDecorator {
    public SportsCar(Car car) {
        super(car);
    }

    public void assemble() {
        super.assemble();
        System.out.println("Adding features of Sports Car.");
    }
}
```

FACADE PATTERN:

- Provides a simplified interface to a complex subsystem.
- Hides the complexity and provides a higher-level interface that’s easier to use.

```java
// Subsystem classes
public class Engine {
    public void start() {
        System.out.println("Engine started.");
    }
}

public class AirConditioning {
    public void on() {
        System.out.println("Air conditioning on.");
    }
}

public class Radio {
    public void playMusic() {
        System.out.println("Playing music.");
    }
}
```

FACADE PATTERN:

```
// Facade class
public class CarFacade {
    private Engine engine;
    private AirConditioning ac;
    private Radio radio;

    public CarFacade() {
        this.engine = new Engine();
        this.ac = new AirConditioning();
        this.radio = new Radio();
    }

    public void drive() {
        engine.start();
        ac.on();
        radio.playMusic();
        System.out.println("Driving...");
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        CarFacade car = new CarFacade();
        car.drive();
    }
}
```

BEHAVIORAL PATTERNS:

Behavioral patterns define how objects interact and communicate with one another. They focus on delegation, responsibility, and how behavior can be distributed across objects. Examples include **Strategy**, **Observer**, and **Command** patterns, which enhance flexibility and scalability in object collaboration.

STRATEGY PATTERN:

- Allows defining a family of algorithms, encapsulating each one, and making them interchangeable.
- Useful when you need different algorithms or behaviors depending on the situation, such as different payment methods or sorting strategies.

```
public interface PaymentStrategy {
    void pay(int amount);
}

public class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid with credit card: " + amount);
    }
}

public class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid with PayPal: " + amount);
    }
}
```

STRATEGY PATTERN:

```
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

OBSERVER PATTERN:

- Allows an object (subject) to notify other objects (observers) about changes without being tightly coupled.
- Often used in event handling systems.

```java
public interface Observer {
    void update(String message);
}

public class ConcreteObserver implements Observer {
    public void update(String message) {
        System.out.println("Received update: " + message);
    }
}

public class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

COMMAND PATTERN:

- Encapsulates a request as an object, allowing for parameterization and queuing of requests.
- Supports undo/redo operations and can decouple the invoker from the logic that processes the request.

```
// Command interface
public interface Command {
    void execute();
}

// Concrete command class
public class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}
```

COMMAND PATTERN:

```
// Receiver class
public class Light {
    public void turnOn() {
        System.out.println("The light is on.");
    }
}

// Invoker class
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

COMMAND PATTERN:

```
// Client code
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOn);
        remote.pressButton(); // Light turns on
    }
}
```